Masterarbeit

Extensible and Versatile Energy Measurement Framework for CPUs

Thomas Oberhauser

21. Mai 2024

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer:	Prof. Dr.-Ing. Horst Schirmeier
Gutachter:	Dr.-Ing. Nils Asmussen
Betreuende Mitarbeiter:	Jan Bierbaum
	Till Smejkal

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

Aufgabenstellung für die Anfertigung einer Master-Arbeit

Studiengang:	Master
Studienrichtung:	Informatik (2010)
Name:	Thomas Oberhauser
Matrikelnummer:	4767653
Titel:	Extensible and Versatile Energy Measurement Framework for CPUs

Besides their normal mode of operation, modern CPUs feature so-called idle or sleep states. In these states, the CPU does not execute instructions but employs various power-saving mechanisms. The CPU enters these idle states only when requested to do so by software, normally by the operating system. Transitioning to an idle state and back to the normal operating mode requires time, the length of which depends on the specific idle state and CPU. Even though the exact properties — energy consumption and transition latency — of a CPU's idle states are highly important for its energy-optimal operation, they are not documented.

An existing Linux kernel module that autonomously measures the aforementioned properties of two generations of Intel CPUs (Haswell and Skylake) serves as the basis for this Master's thesis. This implementation uses the instruction pair `monitor`/`mwait` to transition the CPU to a given idle state and Intel's Running Average Power Limits (RAPL) feature to measure energy consumption. As a side effect of relying on the High-Precision Event Timer (HPET) to wake up cores after the idle phase, the kernel module currently requires a custom Linux kernel.

The primary goal of this thesis is to extend the available infrastructure into a generic, versatile framework able to measure the energy and wake-up latency properties of a wide range of CPUs. A recommended first generalisation step is to modularise the existing implementation. This change should establish well-defined interfaces for triggering state transitions and energy measurement so that `monitor`/`mwait`, HPET, and RAPL become easily replaceable by others mechanisms. Using these interfaces the student should enable the framework to measure wake-latency in addition to energy consumption. He should then add support for AMD x86 CPUs and another CPU architecture, e.g. Arm or RISC-V. As only modern x86 CPUs come with integrated power measurement (RAPL), this step requires support for external metering infrastructure along with proper synchronisation.

As time permits, the student may also pursue any of the following secondary goals:

- Analyse the difference between waking cores from idle states via inter-processor interrupts (IPIs) and by writing to the `monitor`ed memory section when using `monitor`/`mwait`.
- Adapt the framework to run on an unmodified Linux kernel.
- Add full support for „Short-Time RAPL" and explore the viability of short measurement intervals while preserving reliable results.
- Add support for and measure the properties of a heterogeneous CPU.

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

Gutachter:	Dr.-Ing. Nils Asmussen
Betreuer:	Jan Bierbaum Till Smejkal
Ausgehändigt am:	20. Dezember 2023
Einzureichen am:	22. Mai 2024

Prof. Dr.-Ing. Horst Schirmeier
Betreuender Hochschullehrer

Horst Benjamin Schirmeier

Digital unterschrieben von Horst Benjamin Schirmeier
Datum: 2023.12.05 12:47:53 +01'00'

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 21. Mai 2024

Thomas Oberhauser

Abstract

Minimizing energy consumption is crucial, from mobile devices to datacenters. An important processor feature in this regard are "sleep states". They allow to temporarily deactivate idle cores, lowering their energy consumption. Reactivating a core from a sleep state takes time, however, which can impact performance. Modern processors usually support a variety of sleep states, each taking different measures to save energy. Sleep states taking more drastic measures save more energy, but need more time for reactivation. The operating system has to navigate this energy-performance trade-off when selecting the most appropriate sleep state once a core becomes idle. To make an optimal decision, it needs accurate data on the energy consumption and wake-up latency of each sleep state. The data commonly used for this task is of questionable quality, however.

This thesis presents a framework capable of measuring these sleep state characteristics, allowing to gather first-hand, trustworthy data instead. The framework builds upon an already existing implementation which so far measured the energy consumption of sleep states on two Intel processors. In this thesis, I add wake-up latency measurements. Additionally, I extend the framework to work on AMD processors. I also add support for ARM, for which I introduce external energy measurements to the framework. I test the implementation on four different systems, using an Intel Core i7-4790, an Intel Core i7-6700K, an AMD Ryzen 5 5600G and an ARM Cortex-A76, respectively. Results show that the framework works well across manufacturers and architectures, providing precise and reproducible values for both sleep state energy consumption and wake-up latencies.

Contents

List of Figures

List of Tables

List of Acronyms

ACPI	Advanced Configuration and Power Interface. 41, 55, 56
APIC	Advanced Programmable Interrupt Controller. 3, 7, 8, 27, 30, XV
CC-state	Core C-state. 4, 5, 9, 12, 42, 43, 45, 55, 57
CPU	Central Processing Unit. 1, 3, 6, 8, 13, 14, 31, 35, 38, 46, 54, 59, 60
DVFS	Dynamic Voltage and Frequency Scaling. 1, 14, 35
GT	Generic Timer. 3, 12, 22, 24, 28, 52
HPET	High Precision Event Timer. 3, 7–10, 12, 18, 19, 22, 24, 25, 27–29, 31, 32, 36, 42–45, 48, 49, 54, 55
IF	Interrupt Flag. 5, 6, 9, 10, 28–31
IOAPIC	I/O Advanced Programmable Interrupt Controller (APIC). 7, 8, 10, 24, 30, 31, 44
IPI	Inter-Processor Interrupt. 8, 10, 30, 47–49, 56, 60
LAPIC	Local Advanced Programmable Interrupt Controller (APIC). 8, 24, 25, 28–31, 44, 54, 55
MSR	Machine Specific Register. 6, 7, 12, 19, 20, 29, 36, 42, 45, 46, 57
NMI	Non-Maskable-Interrupt. 6, 9, 10, 24, 27–29, 31, 41, 49, 54, 55
NTP	Network Time Protocol. 23, 24
OS	Operating System. 1, 2, 35, 41
OSPM	Operating System-directed configuration and Power Management. 41
PC-state	Package C-state. 4, 12, 13, 23, 36, 39, 42, 43, 45, 46, 48, 49, 55–57, 60

PIC	Programmable Interrupt Controller. 8
PIT	Programmable Interval Timer. 7
RAPL	Running Average Power Limit. 3, 6, 9, 20, 23, 46, 47, 50, 60
RTC	Real-Time Clock. 7
SC	System Counter. 12, 13, 22, 32, 36, 52
SMI	System Management Interrupt. 6
SMT	Simultaneous Multithreading. 4, 5, 9
SR	System Register. 24
TSC	Timestamp Counter. 7, 8, 13, 19, 25, 29, 32, 36

1 Introduction

Energy consumption of computation is a crucial consideration in many domains. Be it to prolong the battery live of laptops and smartphones, to minimize electricity and cooling costs in data centers, or simply to reduce the footprint in regard to climate change, saving energy can be critical. Processor manufacturers are not ignorant of this fact and have added various energy-saving features over the years, e.g. DVFS, which allows throttling the CPU's frequency.

Modern CPUs usually contain multiple cores which can execute code independently of each other. It is common, however, that not all of them have a useful task to work on at any given time. As a result, a core might spend significant durations executing some idle loop, wasting energy. Fortunately, for this situation, too, manufacturers have devised an energy-saving feature in the form of sleep states. In a nutshell, sleep states allow to temporarily deactivate cores, making idling more energy-efficient. These savings do not come for free, however. To save energy this way means flushing caches, stopping clocks, or even removing voltage from the core altogether. When the core's computation power is eventually needed again, all these measures have to be reversed. Only then can the computer provide its maximum performance again. The processor needs some time for this reactivation, the wake-up latency.

The question resulting from these facts is simple: When a core becomes idle, should it enter a sleep state, saving energy? Or should it stay awake in some idle loop, remaining instantly available and not sacrificing performance? The operating system (OS) is responsible for answering this question. On modern CPUs the answer is not a simple, binary "yes" or "no", however. Instead, these CPUs usually offer a variety of sleep states, the Intel Core i7-6700K having six distinct ones, for example. Each of these sleep states has different characteristics, taking measures of varying severity to save energy. Deeper sleep states — those taking more drastic measures — save more energy, but lead to a higher wake-up latency. The OS has to navigate this trade-off between wake-up latency and energy savings, constantly deciding if it can afford to enter a deeper sleep state, or if the latency's impact on performance would become to great.

In Linux the "cpuidle governor" is in charge of this decision. It chooses whether to enter a sleep state and selects the best possible one. The cpuidle governor usually bases its decision on the expected idle duration, which it estimates using heuristics[1]. If a long idle duration is likely, a deep sleep state is preferable as the saved energy over time outweighs the latency cost. On the other hand, if heuristics predict a wake-up event within the near future, the governor should select a less deep sleep state, balancing smaller energy-savings with a shorter wake-up latency. For the cpuidle governor to select the optimal sleep state for the situation, it needs accurate data. This of course

[1] The "menu" cpuidle governor is an example for this, see `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/cpuidle/governors/menu.c?h=v6.2.9#n29`

includes its internal heuristics, but more importantly for this thesis, it needs to know the energy savings and wake-up latency of each sleep state as precisely as possible. Only then is the governor able to weigh these characteristics against each other in an informed manner. This is especially important as absolute power savings have been observed to quickly diminish with each deeper sleep state, while wake-up latencies keep increasing disproportionately [20, p. 6]. Taking this small potential for additional energy savings into account, a deeper sleep state might often not be worth the possible cost to performance.

Currently, the sleep state characteristics used by the cpuidle governor can come from a variety of sources. For Intel, as an example, they are hard-coded into the kernel source code[2]. For other processors the OS might also read them from special tables containing system information [21, pp. 481–483] that system firmware provides in memory [21, p. lxiv]. Wherever this data comes from, it is hard to know how accurate it truly is. Additionally, due to binning during the manufacturing process — aka the "silicon lottery" — processors of the same model are not necessarily identical at silicon level [15, p. 17]. Because of this, even for processors of the same model actual sleep state characteristics might differ. To ensure that the cpuidle governor can make optimal decisions it would thus be preferable to gather trustworthy, first-hand data from the actual processor the OS is working with.

This is exactly where the research conducted for this thesis comes in. The goal is to develop a framework which allows measuring sleep state characteristics for a wide variety of processors. In previous research I already built a framework capable of measuring the energy consumption of sleep states and tested it on two Intel processors. In this thesis, I will extend this existing implementation to measure wake-up latencies as well, which will allow to assess the trade-off between energy-savings and wake-up latencies. Apart from this, I will also increase the range of supported processors to include AMD, another manufacturer, and ARM, a completely separate architecture.

This thesis' structure is as follows: Chapter 2 will go over the knowledge required for understanding the rest of this thesis. In doing this, I will also detail how the already existing framework works, as it acts as the basis for the later extensions. Lastly, Chapter 2 will also describe related research. In Chapter 3, I will discuss how to best add the various required capabilities to the framework, while Chapter 4 will go over some details of the resultant implementation. In Chapter 5, I will present the results of the measurements I conducted using this implementation. Finally, Chapter 6 will give an overview of what I achieved during this thesis and where more research could be done on this topic.

[2] See `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/idle/intel_idle.c?h=v6.2.9#n233`

2 Technical Background

To ease the understanding of later chapters, the following sections will give an introduction into the technical concepts which are most relevant for this thesis. These are:

1. C-states, which are the sleep states of x86.
2. The `monitor`/`mwait` instructions, which are the main way to enter C-states.
3. RAPL, an x86 feature allowing processor-internal energy measurements.
4. Further relevant concepts on x86, such as the HPET and the APIC.
5. My previous implementation of the framework, which allows energy measurements on Intel. This is the basis for the extensions I add in this thesis.
6. Relevant concepts on ARM, like the `wfi` instruction and the Generic Timer.

Afterwards, I will present related work and discuss the relevancy of this thesis in the surrounding field of research.

One of the central concepts in this thesis is the energy-saving feature which allows temporarily deactivating cores when they become idle. As there is no consensus on the name of this processor feature, x86 calling them "C-states", for example, I will simply refer to them as "sleep states" throughout this thesis. Furthermore, I will call the time needed for reactivation when leaving a sleep state the "wake-up latency". Lastly, when comparing sleep states, I will refer to a sleep state which takes more drastic energy saving measures as a "deeper" sleep state. This means that a deeper sleep state will generally save more energy while having a higher wake-up latency than a less deep sleep state.

2.1 C-states

Modern x86 CPUs support a variety of energy-saving measures. One prominent example in this regard is their implementation of sleep states, the so-called "C-states". They refer to various energy-saving states that single cores as well as the entire package can enter when there is no work to be done. Which C-states are available differs between specific processor models, with only C0 and C1 being required by the *Advanced Configuration and Power Interface (ACPI) Specification* [21]. Within the range of available C-states, C0 is not actually a sleep state, as it refers to the core being fully functional and actively executing code. C1 to Cn, however, are "[...] processor sleeping states [...]" [21, p. 473], wherein a core does not execute any instructions. Generally, the higher the number of a C-state, the deeper a sleep state it is, having both a higher wake-up latency and energy savings. This leads to a trade-off between performance and energy consumption when selecting a C-state. [21, pp. 473–475]

Example: Intel Core i7-6700K

As an example, the Intel Core i7-6700K, one of the processors I measured previously, can put its cores in any of the Core C-states (CC-states) described in Table 2.1. As can be seen, higher CC-state numbers generally mean more aggressive energy saving measures, ranging from shortly halting a core to removing its voltage entirely. A CC-state is generally entered upon a core explicitly requesting it, e.g. by using `monitor/mwait`.

If Simultaneous Multithreading (SMT) is enabled, energy saving measures for the core cannot take place until all hardware threads have requested to enter a CC-state. Once this has happened, the CC-state actually entered by the core will correspond to the numerically lowest, least deep requested CC-state on the thread level.

Core C-state	**Description**
C0	Active
C1	Core halted
C1E	Core halted, low frequency, low voltage
C3	L1/L2 caches flushed, clocks stopped
C6, C7, C8	Core state saved, core voltage removed

Table 2.1: Core C-states supported by the Intel Core i7-6700K [11, pp. 63–72]

If each core of a package is in Core C-state CCn or deeper, the entire package can enter Package C-state (PC-state) PCn and apply further energy saving measures. The PC-states supported by the Intel Core i7-6700K are listed in Table 2.2, though in a slightly simplified way, as in those states Uncore components are also increasingly affected. These details were omitted here. As with the CC-states, numerically higher PC-states turn off progressively more components to save additional energy.

Package C-state	**Description**
C0	Active
C2	Temporary state before C3
C3	Clocks to each core shut off
C6	Base clock shut off
C7	L3 may be flushed (voltage removed if completely flushed)
C8	L3 must be flushed

Table 2.2: Package C-states supported by the Intel Core i7-6700K [11, pp. 63–72]

Unlike with CC-states, there is no way to explicitly request to enter a PC-state. Instead, hardware independently decides when and if to enter a PC-state, providing no specific guarantees. This means that, even if the “most awake” core of a package is in CCn, the package does not necessarily enter PCn, even if the processor supports it. [11, pp. 63–72]

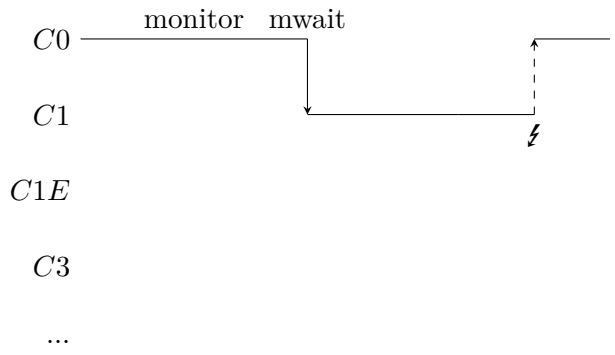


Figure 2.1: General usage of `monitor/mwait` to request entering a Core C-state. ↯ represents a wake-up event, such as an interrupt. The shown C-states are based on those supported by the Intel Core i7-6700K.

2.2 monitor/mwait

As was already mentioned in the previous section, a core (or rather all its hardware threads in case of SMT) needs to explicitly request a Core C-state for it to be entered. The most prominent and modern way to do this is by using the `monitor/mwait` instruction pair. As investigating the behavior and impact of these two instructions was the original idea behind this research, this section focuses on introducing the main properties of `monitor/mwait`. Since Intel seems to have adopted `monitor/mwait` more broadly, this section will mainly be based on Intel's documentation. AMD's implementation of `monitor/mwait` acts the same in all relevant aspects, however [2, p. 414] [2, p. 420].

Figure 2.1 shows the general usage procedure for the `monitor/mwait` instruction pair. The first step is to execute the `monitor` instruction, passing it a memory address. This arms the "address monitoring hardware" on a region in memory that surrounds this address. [13, ch. 4 p. 33] Once this is done, the `mwait` instruction is executed. Its purpose is to request to enter a sleep state until some event occurs. By passing a hint to `mwait` it is further possible to indicate the specific Core C-state that should be entered. [13, ch. 4 p. 160-161] Both the `monitor` and the `mwait` instructions are privileged. [13, ch. 4 p. 33] [13, ch. 4 p. 160]

Once a sleep state has been entered through this procedure, there is a variety of events that cause the core to leave it again:

- A store operation to the address range defined by `monitor`, triggering the address monitoring hardware.
- A hardware-generated interrupt, if it would be delivered to software. Crucially this allows to ignore it by clearing the Interrupt Flag (IF).[1]

[1] The documentation limits this statement by adding that "[i]mplementation-specific conditions may result in an interrupt causing the processor to exit the implementation-dependent-optimized state [$\widehat{=}$ sleep state; author's note] even if interrupts are masked [...]" [13, ch. 4 p. 160].

- A hardware-generated interrupt, even with cleared IF, if a specific optional extension was given when calling `mwait`, enabling "[...] masked interrupts as break events [...]" [13, ch. 4 p. 160].
- A Non-Maskable-Interrupt (NMI), regardless of the IF.
- Other hardware events such as SMI, debug or machine check exception, various signals.

Still, the documentation does not give any guarantees that a core will remain sleeping until one of these events occurs, explaining that "[o]ther implementation-dependent events [...]" [13, ch. 4 p. 160] can also cause the processor to leave the sleep state. [13, ch. 4 p. 160]

2.3 Running Average Power Limit (RAPL)

Another concept applied throughout this thesis is Running Average Power Limit (RAPL), an x86 processor feature present in many modern Intel and AMD processors. Originally introduced by Intel to allow enforcing power consumption limits, it also includes a "[p]ower metering interface providing energy consumption information" [14, ch. 15 p. 48] consisting of special Machine Specific Registers (MSRs). AMD started providing energy values through RAPL as well beginning with the introduction of the Zen architecture, using different MSRs however [18, p. 564]. On a processor with RAPL, reading the provided MSRs can provide insight into the power consumption of a CPU without needing any external measurement infrastructure.

RAPL only offers limited spacial and temporal granularity, however. On one hand, there are the different RAPL domains. For Intel, only the "Package" and "PP0" domains have to be supported. While "Package" measures the energy consumption of the entire processor, "PP0" allows restricting the measurement to the collective cores only, excluding Uncore components. It is not possible to get insight into the energy consumption of single cores on Intel processors [14, ch. 15 pp. 48–53]. For AMD, a similar package domain, as well as a core domain are supported. Unlike Intel's implementation, AMD's core domain does provide per-core energy consumption metrics [18, p. 564]. The temporal granularity, on the other hand, is limited by the update period of the energy counters which is roughly one millisecond [14, ch. 15 pp. 48–53] [18, p. 569].

For Intel, RAPL accuracy has been verified to be very high for processor generations from Haswell onwards, where the reported values started to be based on actual measurements instead of a modelling approach [9]. As for AMD, at least until Zen2, RAPL values are still based on a model. Because of this, accuracy comparable to Intel cannot be expected, which has also been confirmed experimentally [18].

2.4 Further Concepts on x86

This short section will introduce any further concepts and components that are relevant to understanding the implementation of the framework on x86. Figure 2.2 gives a rough overview of the location of these components in the system.

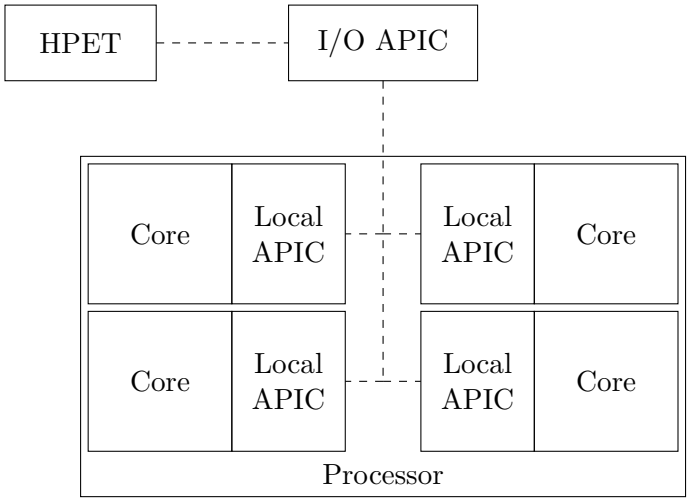


Figure 2.2: Overview of the location of relevant system components on x86 [14, ch. 11 p. 1] [1, p. 620] [12, p. 7]. The visualized processor is a generic example.

2.4.1 High Precision Event Timer (HPET)

Originally introduced to supersede the older PIT and RTC timers [12, p. 5], the High Precision Event Timer (HPET) is nowadays broadly supported on x86 systems. It is implemented using only a single, monotonically increasing counter register. Nonetheless, it can support multiple independent timers through so-called "comparators". Each comparator has an associated, externally accessible register, which it constantly checks against the counter. Once the counter matches the register, the timer fires an interrupt. Writing directly to a comparator's register to request a one-shot interrupt is always supported, periodic interrupt generation however is an optional feature. [12, p. 7]

As for precision, the *IA-PC HPET (High Precision Event Timers) Specification* [12] recommends a counter tick period of at most 100 ns ($\widehat{=}$ a minimum frequency of 10 MHz). For an interval longer than 1 ms, the frequency should not drift by more than ± 0.05 percent [12, p. 9].

2.4.2 Timestamp Counter (TSC)

The Timestamp Counter (TSC) is an efficient time-keeping mechanism on x86 as reading from it is very fast. This is because, internally, the TSC is a special Machine Specific Register (MSR) which increments monotonically. Originally, the TSC's frequency depended on the processor's frequency, as well as C-states. On modern processors, the TSC increments at a constant rate, however. This allows to use it for defining absolute timestamps and to easily calculate the time between different timestamps. [14, ch. 18 pp. 42–43] [1, pp. 422–423]

2.4.3 Advanced Programmable Interrupt Controller (APIC)

On modern x86 CPUs, the Advanced Programmable Interrupt Controller (APIC) is in charge of interrupt delivery, having replaced the older 8259 PIC. To integrate better into multi-processor systems, the APIC consists of one or more outward-facing I/O APICs (IOAPICs), and a set of per-core Local APICs (LAPICs). It is the IOAPIC's job in the chipset to receive any external interrupts and forward them to their assigned LAPICs as an interrupt message, where they can be signaled to the core and handled accordingly. In addition to this main purpose, there are further usages, like sending Inter-Processor Interrupts (IPIs). A variety of configuration registers further allows controlling details like the delivery mode of interrupts or interrupt masking. [14, ch. 11 pp. 1–3] [1, pp. 620–629]

An additional important feature contained in the APIC is the Local APIC Timer. It allows requesting timed interrupts from the per-core LAPIC [14, ch. 11 pp. 16–17] [1, pp. 629–631], speeding up accesses compared to external timers like the HPET and avoiding the need to set up interrupt routing. On newer Intel CPUs, interrupts can be requested at absolute timestamps that are compared to the TSC [14, ch. 11 pp. 17–18].

2.5 The Measurement Framework

As this thesis builds upon a measurement framework which I already created during earlier research, this section will explain how it works and present results obtained with the described methodology. This section will be mostly based on the paper "Sleep Well" [20], which incorporated my previous work on the subject.

The measurement framework is designed to run inside the Linux kernel. As such, most of the required code is written as a Linux kernel module. A small modification to the Linux kernel itself was necessary as well, for which the Linux kernel version 6.2.9 was used as a basis.

2.5.1 Measuring

The main design goals regarding the measurement process were to gain values as precise as possible while being able to run measurements in a fully automated fashion. An overview of the sequence of actions during a measurement is given in Figure 2.3.

In very general terms, each measurement starts with some general setup before the framework takes over all cores, preventing any interference from that point onward. After that, the actual measurement takes place, during which the framework puts the cores in the requested state and collects power values. Once this is finished, the framework returns control of the cores to the system and finalizes the measurement before publishing the results back to the user.

Phase 1: Setup

Inserting the kernel module starts the measurement process, beginning with some general setup steps. Most importantly, the IOAPIC is set up so that, later on, the wake-up

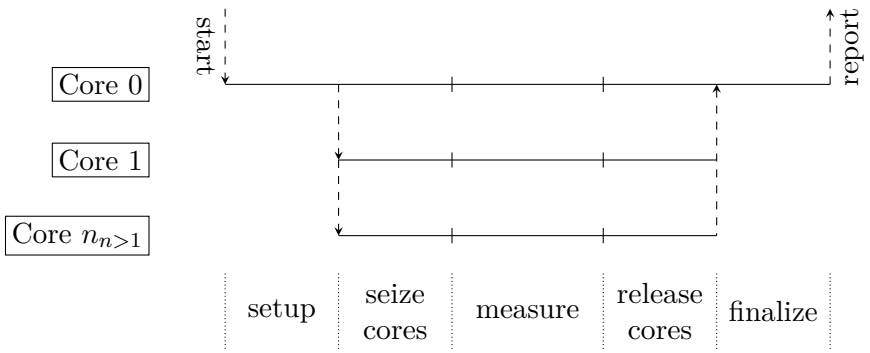


Figure 2.3: The measurement procedure in the original measurement framework.

interrupt that is generated by the HPET gets delivered to the correct core as a Non-Maskable-Interrupt (NMI). This core — the one that is woken up first at the end of the measurement — is referred to as the "leader" core[2]. Also at this point, the framework registers the interrupt handler for the wake-up interrupt. Apart from that, it might also need to activate some internal counters during this step which can give further insight into the processor's behavior during the measurement.

Phase 2: Seize Cores

Once the general setup is completed, the measurement algorithm needs to take control of all available cores. The intention of this is to control the system as completely as possible, creating an environment suitable for precise and reproducible measurements. Here, the framework uses the `on_each_cpu()`[3] interface of the Linux kernel, which allows to execute a function on all cores in the system.

The function that is called through this interface deactivates preemption to keep the scheduler from interfering after this point. Additionally, the Interrupt Flag (IF) is cleared on each core, disabling interrupts from causing wake-ups. Before the actual measurement period can start, the framework needs to make sure that all cores are ready. Because of this, it employs a synchronization mechanism at this point.

Phase 3: Measure

Once all cores have reached the synchronization barrier, the actual measurement phase begins. At this point, all relevant internal counters have their starting value read and saved. This includes the RAPL counters which the framework uses to measure the energy consumption. Then, the leader core requests the interrupt from the HPET that is responsible for ending the measurement later on.

After all this is done, each core executes the `monitor`/`mwait` instructions and requests whichever Core C-state the user indicated when starting the measurement. Since the

[2] In case of Simultaneous Multithreading (SMT), the leader core can also be a hardware thread. The measurement procedure stays the same with SMT, just with hardware threads instead of cores.

[3] `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/smp.h?h=v6.2.9#n69`

framework fully controls the system, all cores should remain in their assigned state for the entire measurement duration.

The "measure" phase ends once the interrupt requested from the HPET fires, waking the leader core. Crucially, it can do this while the IF is cleared on that core because it is delivered as an NMI. The leader core's responsibility is now to wake all other cores by sending them an Inter-Processor Interrupt (IPI). This IPI, too, is delivered as an NMI. All internal registers of which the start value was recorded have their end values read at this point, too.

Phase 4: Release Cores

In this short phase, each core transfers control back to the kernel by re-enabling preemption as well as interrupts. After this point, system operation should already mostly continue as before.

Phase 5: Finalize

This last phase has three responsibilities.

Firstly, it needs to evaluate the data gathered during the measurement. This consists mainly of comparing the collected start and end values of the internal counters that were collected earlier.

Secondly, the framework needs to reset everything that was reconfigured for the measurement to its original state. This is necessary to ensure continued system functionality. It includes restoring the previous state of the IOAPIC and deregistering the wake-up interrupt's interrupt handler.

Lastly, the finalized results need to be reported to the user. This happens by setting up a suitable directory structure in the sysfs[4].

2.5.2 Results

Using the outlined measuring procedure, I already measured two different systems, one with an Intel Core i7-4790 and one with an Intel Core i7-6700K. The results, which were already incorporated into "Sleep Well" [20], are shown in Figures 2.4 and 2.5, respectively. Please note that "C0" in this case refers to requesting to enter C0 when executing `monitor/mwait`. As C0 is no actual sleep state, the `mwait` instruction completes almost immediately. Because of this, `monitor/mwait` is run in a loop until the measurement finishes.

Looking at both figures, which show 10 separate measurement values per C-state as well as their mean, it is apparent that the observed power values are extremely stable for all measured C-states. Also, the results look plausible overall, with deeper C-states needing less energy. All in all, the presented approach to measuring C-state energy consumption seems to work quite well.

[4] The sysfs is a special filesystem in Linux that can be used to provide file-like interfaces to kernel-internal data or functions.

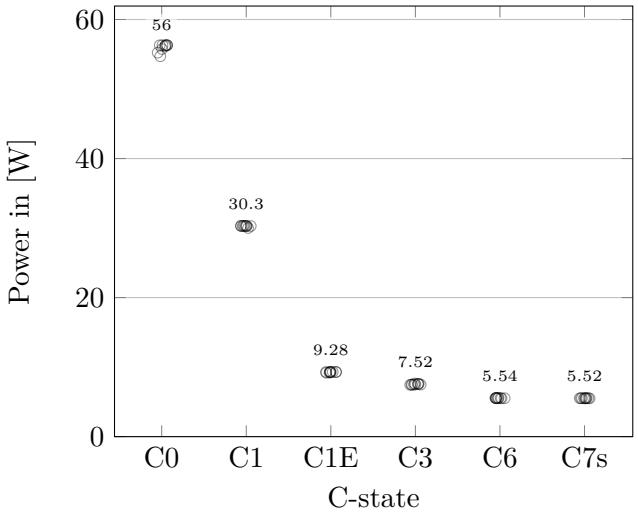


Figure 2.4: Processor energy consumption of different C-states supported by the Intel Core i7-4790 as measured for "Sleep Well" [20].

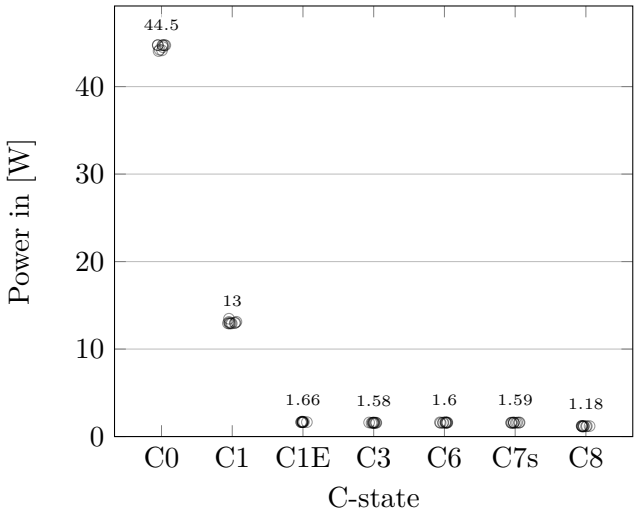


Figure 2.5: Processor energy consumption of different C-states supported by the Intel Core i7-6700K as measured for "Sleep Well" [20].

A slight issue with the results, however, is that no Package C-states deeper than PC2 were ever entered during the measurements. This was verified by looking at special MSRs provided by Intel which allow tracking the time spent in specific Core and Package C-states. Because of this, no conclusions can be drawn about any further power savings potentially possible through those deep PC-states.

2.6 ARM

As this thesis will introduce support for ARM to the measurement framework, I will use some ARM-specific features later on. This section gives an overview of all necessary concepts. Specifically, as versions of ARM can differ, this section presents features of the ARMv8-A architecture.

2.6.1 `wfi`

As C-states and `monitor`/`mwait` are x86-specific concepts, they are not applicable when working with ARM. Instead, ARM provides the `wfi` ($\widehat{=}$ wait for interrupt) and `wfe` ($\widehat{=}$ wait for event) instructions, which act very similar to each other, both allowing to enter sleep states. In this section, I will focus on the `wfi` instruction. Similar to `mwait`, `wfi` allows to enter a "[...] low-power state" [4, p. 6119] until some wake-up event occurs. This event is usually an interrupt, as the name implies. However, similar to `mwait`, the core can also leave the sleep state due to a variety of other reasons, including "[a]n IMPLEMENTATION DEFINED wake event that is architecturally permitted to occur at any time" [4, p. 6119]. In general, the *Arm Architecture Reference Manual for A-profile architecture* [4] does not make many strict requirements for the implementation of `wfi`. The same is true for the sleep states themselves, which are not defined beyond their effects on memory coherency and the architectural state. [4, pp. 6119–6120]

Example: Cortex-A76

This is why it makes sense to directly look at an example. For this I chose the Cortex-A76 which is built into the Raspberry Pi 5, one of the test systems used for this thesis. The *Arm® Cortex®-A76 Core Technical Reference Manual* [5] simply states that it disables "[...] most of the clocks in the core [...]" [5, p. 38] on executing `wfi`, not giving too much detail, either [5, pp. 38–39]. Additionally, after a timeout expires inside normal `wfi` mode, a core may enter "Core dynamic retention mode", where the core's clock is gated outside the core. Unlike the different Core C-states on x86, it can not be directly requested, however. [5, p. 42]

2.6.2 Generic Timer

Instead of x86-specific timers like the HPET, ARM has its own timers, too. The most important one with regard to this thesis is the Generic Timer. It provides each core with multiple timers, each one having a programmable register containing a "[c]omparator value" [6, p. 11]. Once the System Counter (SC) reaches this value, an interrupt fires.

[6, pp. 10–13] The SC is similar to the TSC on x86. It is a counter which increments at a constant rate [4, p. 6832], allowing to define absolute timestamps against it.

As for the accuracy of the SC, ARM recommends a drift of at most ±10 s per 24 h. This is not a strict requirement, however. For recent CPUs, the SC operates at a fixed frequency of 1 GHz, though for older models this frequency is typically lower at 1 to 50 MHz. [4, p. 6832]

2.7 Related Work

While there is some literature on analyzing the properties of sleep states, the approach to tightly control the environment used in this thesis seems to be unique. Nonetheless, this section will discuss other ideas on how to measure the energy consumption and wake-up latencies of sleep states.

2.7.1 User-space Measurements

In "Wake-up latencies for processor idle states on current x86 processors" [19], Schöne, Molka, and Werner introduced a method to measure the wake-up latencies of C-states on x86 processors. They modified the Linux kernel to introduce a new interface to the sysfs, allowing to explicitly trigger a specific core to wake up. The core processing the interaction with this interface (the "caller") would take a timestamp before waking up the requested other core (the "callee"). Upon reaching C0, the callee would then take another timestamp and also remember the C-state it woke up from. Calculating the wake-up latency from the difference of these two timestamps, and repeating this process many times, would then allow to draw conclusions about the wake-up latency associated with specific C-states.

In subsequent work by Ilsche et al. [10], this approach was improved upon to not be dependent on a modified kernel anymore. Caller and callee were transformed into threads of a user-space application, and all required data was collected through trace points instead. Overall, the process remained the same, however.

Using this approach, sleep state wake-up latencies were calculated for — among others — Intels Haswell [9] and Skylake [17] microarchitectures, as well as AMDs Zen2 microarchitecture [18].

Comparison to this Thesis

When it comes to measuring the wake-up latencies of C-states, the approach by Schöne, Molka, and Werner [19] and Ilsche et al. [10] does have some important advantages.

Firstly, it uses the sleep states exactly as the Linux kernel does. This is a positive attribute, as writing an entirely new C-state entering routine as in this thesis has proven tricky at times. There appear to be situations that hinder cores from entering a prolonged sleep or prohibit certain Package C-states. When reusing the implementation of the kernel, it can be expected that difficulties like these have already been dealt with. Or, in case they still appear, it can be reasonably concluded that the fault may lie with the

kernel, not the measuring method. This is harder to argue in the case of the framework in this thesis.

Another advantage is the smaller interference with the system, especially with the newer implementation that does not require a custom kernel. The measurement framework presented in this thesis reconfigures interrupt routing as well as system timers, basically rendering the system inoperable for the duration of the measurement. Due to potential problems like lost interrupts, this may have negative effects on system stability even after the measurement has ended. Before broadly deploying it, this risk would have to be considered, addressed, and solutions properly tested. In contrast, with the approach by Ilsche et al. [10], system stability should not pose a problem, as it stays fully operable.

This advantage goes hand in hand with a disadvantage though. If the system is not fully controlled on kernel-level, you can not rule out interference or even reliably enter a specific C-state. To remedy this, more measurements are conducted (e.g. 400 per state in "Wake-up latencies for processor idle states on current x86 processors" [19, p. 223]), but even so, the results are inherently limited in their stability and precision. Also, more measurements mean a more time-consuming measuring process overall. While for this thesis' framework it is entirely plausible to integrate measuring into e.g. the boot process, it is harder to imagine for the approach by Schöne, Molka, and Werner [19] and Ilsche et al. [10].

A last aspect to discuss are the power measurements. Schöne, Molka, and Werner [19] estimated power savings using a relatively specialized external measuring device, but not much detail is given on how much time these measurements took or if they were automated in any way. Ilsche et al. [10], on the other hand, used a "[...] custom DC device, which provides a resolution of 2 µs" [10, p. 713], even allowing to watch C-state transitions in real-time. Both these approaches are not feasible or practical to apply broadly, which probably was also never the authors' goal. Nevertheless, this is another advantage of the measurement framework presented in this thesis. Since it does not rely on any external measurement infrastructure on x86, and can be configured to run with external measuring devices with low temporal granularity, it should be generally easier to use.

2.7.2 Further Literature

There are further papers about measuring different related aspects of CPUs. Daud et al. [7], for example, measured the energy consumption of embedded processors for different processor utilization percentages. Mazouz et al. [16], on the other hand, measured the switching latencies between different DVFS frequencies.

However, there does not seem to be further comparable literature focussing specifically on sleep states.

2.7.3 Conclusion

While there is other literature about this topic, the approaches taken by different groups are quite distinct when compared to the idea behind this thesis. Specifically, fully controlling the system at kernel-level is not done by anyone else. This aspect

however is what allows doing measurements quicker and with higher precision and stability. Furthermore, especially when it comes to power measurements, this framework's comparably low requirements regarding special hardware should make it easier to apply to new systems, potentially enabling the collection of many more data points. The high level of automation only furthers this argument.

All in all, the approach taken in this thesis is novel and offers significant advantages over existing solutions, making it a worthwhile addition to this field of research.

3 Design

This chapter will present the objectives for improving the existing measurement framework, which was described in Section 2.5. In summary, these objectives are:

1. Adding wake-up latency measurements: I should enable the framework to measure wake-up latencies in addition to power consumption. As a first step I will base this on the timer interrupt that ends the measurement on x86.

2. Measuring wake-up latencies when triggering `monitor`: There are multiple ways to cause a core to wake up on x86. Apart from the timer interrupt, the framework should also give insight into wake-up latencies when triggering the address monitoring hardware set up by `monitor`.

3. Supporting AMD: So far, the existing framework only measured systems with Intel processors. I should now add support for AMD processors, too.

4. Modularizing the framework: To enable adding support for further architectures apart from x86, I should modularize the framework. This modularization should facilitate reusing architecture-agnostic code while allowing to easily swap architecture-specific code.

5. Supporting ARM: Based on the modularized framework, I should add support for ARM processors.

6. External Measurements: As ARM does not offer comparable internal power measurements to x86, external power measurements are necessary to achieve a fully capable framework on ARM. Because of this, I need to add support for integrating power measurements from external measurement devices.

7. Supporting an unmodified kernel: The existing framework requires a custom kernel to conduct measurements. I should overcome this requirement, allowing to use an unmodified kernel.

For each goal, I will explain the challenges and discuss multiple solutions where possible.

3.1 Measuring Wake-up Latencies

This section will discuss how wake-up latencies can be measured within the existing measurement process as described in Section 2.5.1. As a first step, I will only consider x86. The content of this section was already incorporated into the paper “Sleep Well” [20]

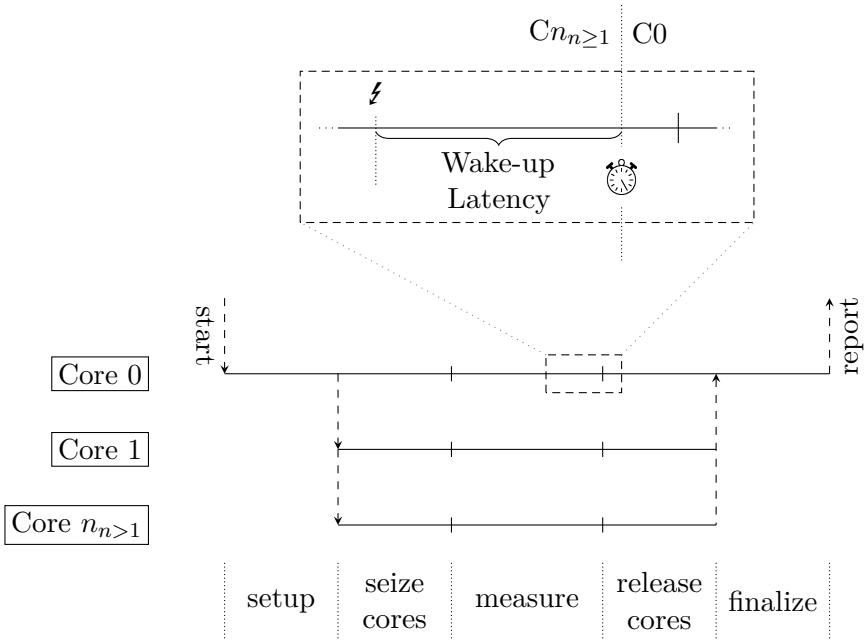


Figure 3.1: The concept behind integrating wake-up latency measurements into the measuring procedure. Core 0 is the leader core. ↯ represents the wake-up interrupt from the HPET. ⏲ stands for the timestamp that is taken at the end of the wake-up period, when C0 (active state) is reentered.

previously, of which I am a co-author. The reason for its inclusion was that it fit well with the other content of the paper and that I finished this part of the thesis early enough. I also already created an implementation of these latency measurements for "Sleep Well" [20] and collected wake-up latency values for two systems. I will present these results in Section 5.1.

When looking at the measurement process employed by the existing framework — as described in Section 2.5.1 — the latency to be measured occurs right at the end of the "measure" phase. It is the time between the generation of the wake-up interrupt and the first instruction to be executed after waking up, as this corresponds to the time a processor core needs to regain full functionality from a sleep state.

In more technical detail, towards the end of the "measure" phase, at a point in time that was set previously, the High Precision Event Timer (HPET) will generate a wake-up interrupt that will be delivered to one selected core, the leader core. The arrival of this interrupt triggers the leader core to begin exiting its sleep state, and shortly thereafter resume the execution of instructions. The entry point for code execution will be the interrupt handler that the framework registered during the "setup" phase.

The main idea behind measuring the wake-up latency at this point is that it can be calculated by taking the difference between the points in time of the generation of the wake-up interrupt and the resumption of code execution on the leader core. As the framework itself is responsible for setting the timestamp of the wake-up interrupt in the HPET, the first of those is already known. The second relevant timestamp can be

collected easily as well, as execution is resumed in code under the framework's control — the registered interrupt handler — where a timestamp can be taken. All these timestamps are in regard to the internal counter of the HPET. Figure 3.1 shows an overview of this measurement process for wake-up latencies.

3.2 Measuring Wake-up Latencies when triggering `monitor`

All latencies measured using the already presented methodology are in regard to a timer interrupt by the HPET as the wake-up event. However, as Section 2.2 described, there are other events, too, which can cause a core to exit a sleep state that was entered by `monitor`/`mwait`. One of these is triggering the address monitoring hardware that `monitor` set up. This section will describe how wake-up latencies can be measured for this wake-up event.

The overall procedure for calculating the wake-up latency on writing to `monitor`ed memory is similar to the one when the interrupt from the HPET causes the wake-up. We still need two timestamps, one when the event triggering the wake-up occurs, and the other when the targeted core becomes fully operational again. However, where the triggering event was the interrupt by the HPET before, it is now the leader core writing to a `monitor`ed memory address. And where previously only the leader core was woken, the targeted cores are now all cores except the leader.

Previously, the timestamp of the wake-up interrupt had already been known, as the framework explicitly programmed it into the HPET. Now, the leader core takes this first timestamp right before it writes to the memory `monitor`ed by the other cores. The second timestamp came from the wake-up interrupt's handler before, since it was the entry point for code execution after the wake-up. A write to `monitor`ed memory does not cause a handler to be executed, however. Instead, `mwait` simply exits and code execution resumes with the next instruction. Thus, the framework takes the second timestamp by inserting the associated code right after `mwait`. As the HPET is not involved in this process, taking timestamps in regard to its internal counter makes less sense than before. Instead, the measurement framework takes these timestamps by reading the TSC, which is quicker to access. Simply taking the difference of the two collected timestamps gives us the wake-up latency for a wake-up triggered by writing to `monitor`ed memory. Because the second timestamp is taken by each woken core individually, a single execution of this procedure yields distinct values for this metric for each core except the leader.

3.3 AMD

Since AMD's processor architecture is generally compatible with Intel's, running the framework on AMD processors does not require too many adjustments. Nonetheless, there are some conflicts that need to be addressed, namely (1) different Machine Specific Registers, (2) a new C-state entry mechanism and (3) a separate cpuidle driver. In this section, I will first give more details about each of these differences and then outline possible solutions.

3.3.1 Differences between Intel and AMD

Firstly, there are some differences in the Machine Specific Registers (MSRs) used by the framework. Because of this, Intel-specific MSRs need to be exchanged with their AMD equivalents, or removed if no such equivalent exists. An example for this is Running Average Power Limit (RAPL), since — as was mentioned in Section 2.3 — different MSRs need to be accessed in order to read the required energy values.

Secondly, while the Intel processors supported so far can enter any C-state by using `monitor`/`mwait`, this is not the case for the Ryzen 5 5600G on which I tested the support for AMD. Instead, some C-states need to be requested by reading from specific I/O ports. To allow measuring all available C-states, this deviating entry mechanism needs to be supported as well.

A final, minor difference is that so far the framework detected the sleep states to be measured by reading information provided by the "intel_idle" cpuidle driver, which is Intel-specific software not used on AMD systems. As such, the framework needs to be able to understand the information provided by the more generic "acpi_idle" driver as well.

3.3.2 Addressing the Differences

As the required adjustments to the existing implementation are rather minor, it does not make much sense to implement an entirely separate program flow just for AMD. Instead, extending the existing implementation to address these differences seems more sensible, least of all to avoid unnecessary code duplication.

Since the support for addressing differences 1 and 2 needs to be built into the kernel module, another decision has to be made: Should the code that supports these AMD-specific details always be compiled in and then selected at run-time? Or should it already be selected at compile-time, allowing the user to produce an Intel- as well as an AMD-specific binary? As it would be detrimental for the ease-of-use of the framework to have to decide between binaries, I ultimately selected the first option.

Using the `cpuid` instruction, it is possible to detect the processor's manufacturer during run-time. Based on this information, the framework can conditionally read from manufacturer-specific MSRs. This already deals with difference 1.

As for difference 2, implementing the I/O read entry mechanism does not pose any special challenges. It is not even necessary to conditionally disable this entry mechanism on Intel processors. The rationale for this stems from Intel documentation, which states that "[...] software may make C-state requests using the legacy method of I/O reads [...]" [11, p. 68]. So even if "[...] I/O reads are converted within the processor to the equivalent MWAIT C-state request" [11, p. 68], and Intel's C-states seem to be generally only requested through `monitor`/`mwait` nowadays, I/O reads still have some meaning on Intel processors. Ultimately, while the preferred entry mechanism should still be the more modern `monitor`/`mwait`, it does not hurt to allow I/O reads for requesting C-states on Intel, too.

Lastly, detecting the available sleep states based on the cpuidle information in the sysfs happens in a script in user-space. Extending this script, it is no issue to parse the

information from the “acpi_idle” driver and hand it to the kernel module. This deals with difference 3.

3.4 Modularization

As a prerequisite for supporting other processor architectures, properly modularizing the framework is recommendable. The objective is to be able to easily exchange the architecture-specific code fragments for others while reusing the generic parts of the code.

I originally implemented the framework with only x86 processors in mind. Naturally this made it unnecessary to cleanly separate the x86-specific code from code that may be reused for other architectures. This went as far as putting all source code of the kernel module into a single file. All in all, the existent implementation of the framework is simply not modular.

This section will first go over how the switching between architecture-specific code works internally. Afterwards, I will describe the most interesting parts of the interface between the generic and the architecture-specific code.

3.4.1 Switching between Architectures

Unlike for AMD, where we had the option of either selecting AMD-specific code at run- or compile-time, this question does not pose itself here. It is obvious that a separate binary will have to be built anyway, as we will be dealing with completely different architectures. The question is rather how switching between different architecture-specific code during compilation should be implemented.

The approach I ultimately went with is quite similar to how the Linux kernel deals with the same issue. Effectively, each architecture has a separate directory containing its specific source code files. Within these files the architecture-specific definitions of symbols are provided. The selection of the architecture happens inside the Makefile that orchestrates the compilation process. The Makefile simply uses a different architecture-specific directory during compilation and links the architecture-specific code with the generic parts of the kernel module.

3.4.2 Interfaces for architecture-specific Code

As the main steps of measuring are the same across architectures, the overall procedure is controlled from the generic code. This for example includes starting multiple measurements in sequence or conditionally repeating erroneous measurements. Some low-level steps can be done from architecture-agnostic code, too, for example disabling preemption or synchronizing all cores before the “measure” phase.

Within this generic procedure, a number of “hooks” are inserted. These are calls to function symbols which the source code files in the respective architecture-specific directory define. Using these hooks allows inserting architecture-specific code at various points of the generic measuring procedure. Here are the most important hooks used in the framework:

- Various `prepare_*()` hooks for doing setup at multiple points of the measuring procedure, e.g. right at the start or before each individual measurement. Each of these hooks has an associated `cleanup_*()` hook.
- The `do_system_specific_sleep()` hook, where each architecture inserts the appropriate code to enter its sleep states.
- The `setup_wakeup()` and `setup_leader_wakeup()` hooks, which allow requesting the timed event (usually an interrupt) in charge of ending the measurement. The two varieties exist to enable either only waking the leader core, which then wakes the other cores by some different mechanism, or having a timed event per core. In the first case `setup_wakeup()` does nothing. In the second case, `setup_leader_wakeup()` simply calls `setup_wakeup()` internally.
- The `publish_measurement_results()` and `cleanup_measurement_results()` hooks, which add the measurement results to or remove them from the sysfs. Architecture-specific code is necessary at this point as the collected data differs between architectures, affecting the published directory structure.

By defining all hooks appropriately, it is possible to quickly adapt the generic measurement procedure to a new architecture. Note that not all hooks are strictly necessary for each architecture. It might be perfectly reasonable to define some hooks as an empty function.

3.5 ARM

Using this modularization technique, I added support for another processor architecture. For this I chose ARM, or more specifically, as versions of ARM can significantly differ, the ARMv8-A architecture. This section will describe the two main adjustments necessary to make the framework function on this architecture.

Firstly, the mechanism for entering sleep states needs to be adapted. While on x86 the implementation of `do_system_specific_sleep()` uses `monitor`/`mwait` or I/O ports, these mechanisms are now swapped out for `wfi`. I described the `wfi` instruction in Section 2.6.1.

The second major change is regarding how to generate the timed wake-up event in charge of ending the measurement. As the HPET is unavailable on ARM, I used the Generic Timer (GT) instead; see Section 2.6.2. The GT allows to trigger an interrupt once the periodically up-counting System Counter (SC) equals or exceeds a programmable register. Since this is very alike to how the HPET functions, the methodology for calculating the wake-up latency from x86 can be reused. Please consult Section 3.1 for details, replacing the HPET and its internal counter with the GT and SC. The only other difference is that code execution upon a wake-up does not resume inside an interrupt handler, but directly behind `wfi`. Thus, I moved the code taking the second timestamp there; see Section 4.1 for details. As the GT is a per-core device, I decided not to have a distinct leader core for this implementation. Instead, each core requests its own wake-up interrupt from its GT. The advantage of this is that it simplifies the code as the

distinction between leader and non-leader cores is no longer necessary. Additionally, each single measurement can gather more data points regarding the wake-up latency, as each core produces a meaningful value. Nevertheless, this approach could lead to problems if something akin to Package C-states (PC-states) were present on the processor to be measured. This is because only the first core of the package to wake up deals with the additional wake-up latency of the PC-state. If you are interested in this additional latency, you might need to control which core wakes up first. This was not an issue on the system I tested for this thesis, however.

With these adjustments I managed to establish basic functionality of the framework on ARM. It is able to put the processor into specific states — e.g. all cores sleeping with `wfi` — for arbitrary durations of time. Additionally, it can calculate the wake-up latencies upon exiting these states. I was able to achieve this without introducing code directly into the Linux kernel, which is why no custom kernel is required to run the framework on ARM.

3.6 External Measurements

Another issue when supporting ARM is the lack of a processor-internal power measuring feature. While on x86 the framework can simply use Running Average Power Limit (RAPL) as a source for power measurements, other processor architectures usually lack comparable capabilities. To gain insight into the energy consumption of other architectures it becomes thus necessary to also add support for using external power measuring infrastructure in conjunction with the framework. This section will describe the challenges and solutions in doing this.

It is easy to plug the measured system into a power measuring device during a measurement run and gain power values that way. Instead, the main challenge comes from associating power values with individual measurements. Usually when working with external measuring devices, you end up with some kind of log consisting of power values associated with timestamps. The naive solution is to simply have the framework on the measured device emit the timestamps associated with the start and end of each individual measurement. You could then compare these time intervals with the timestamps in the power log and match power values to measurements that way. However, while the time intervals of the measurements are defined respective to the internal clock of the measured system, the timestamps of the power log are not. Instead, they use the time of the system that collected the power values. These two clocks are not necessarily synchronized. For longer durations of individual measurements it may be sufficient to explicitly synchronize these clocks beforehand, using the Network Time Protocol (NTP) for example. For very short durations the accuracy of these synchronization mechanisms might not be sufficient, however.

The solution I decided on was to generate a unique pattern in the energy consumption of the measured system at the start of each measurement run. As the framework generates this signal pattern on the measured system, it can emit a start timestamp with regard to the internal clock of the measured system. The framework can then later detect the power pattern in the power log, giving you the start time of the pattern

regarding the time source of the power log. Using this pair of values, all timestamps of the power log can then easily be associated with their respective timestamps on the measured system. The only requirement is that both time sources run at the same speed, without significant drift in regard to each other. For the systems used in this thesis and measurement durations of at most 1 minute, drift seems unlikely to be an issue. This approach to synchronizing times has multiple advantages. Firstly, its accuracy is solely dependent on the sample rate of the measuring device. This means that there can hardly be a mismatch where measurement results become unusable because of the comparably low accuracy of the time synchronization mechanism. Secondly, there might be situations where synchronization protocols like NTP can not be used, for example due to incompatibility with the power measuring device. It should always be possible to generate a power pattern, on the other hand. And lastly, while this approach does work on absolute timestamps, they are not strictly necessary. As long as the timestamps in the power log correctly show the progression of time in regard to each other they can also be associated with the timestamps on the measured system.

3.7 Supporting an unmodified Kernel

The existing implementation of the framework — as presented in Section 2.5 — depends on some changes to the source code of the Linux kernel, making the installation of a custom kernel a requirement for its usage. Overcoming this requirement would be preferable. As the implementation for ARM already managed to avoid introducing any dependencies on custom code in the kernel, the framework can already run on an unmodified kernel on ARM. On x86, however, a custom kernel is still necessary. The objective is thus to remove the need for a custom kernel on x86, too.

The responsibility of the custom kernel code on x86 is to allow setting up the timed wake-up interrupt. The modifications enable kernel module code to program the HPET to generate the interrupt. They also allow reconfiguring the I/O APIC (IOAPIC) to route the wake-up interrupt to the leader core and deliver it as an NMI. These steps were not possible using preexisting interfaces in the Linux kernel as those generally do not allow a kernel module enough control. On ARM, adding similar custom interfaces to the kernel was not necessary. This is mainly because ARM's implementation used the core-adjacent Generic Timer (GT), which can be configured through per-core System Registers, completely bypassing the kernel. Additionally, setting up interrupt routing was unnecessary on ARM, as interrupts by the GT are always routed to the associated core. On x86, something similar is possible using the Local APIC (LAPIC) Timer, which I described in Section 2.4.3. This is because the Linux kernel allows kernel modules to directly write to the LAPIC's memory-mapped configuration registers, giving the framework the necessary control without needing custom interfaces. Like for the GT, there also is no need to set up interrupt routing, as LAPIC Timer interrupts always target the associated core. Thus, replacing the HPET with the LAPIC Timer makes the custom kernel code regarding the IOAPIC obsolete as well. This allows to completely get rid of the changes to the kernel code and run the framework on an unmodified kernel for x86, too. The process for measuring wake-up latencies remains largely the same as in

Section 3.1, simply replacing the HPET and its internal counter with the LAPIC Timer and the TSC. The only other difference is that, like on ARM, code execution upon a wake-up does not resume inside an interrupt handler. Instead, I placed the code taking the second timestamp directly after `mwait`; see Section 4.1.

4 Implementation

This chapter deals with my implementation of the presented design. I will go into detail on a selection of its most interesting aspects, namely:

1. How I maximized the accuracy of the wake-up latency measurements
2. Details about supporting an unmodified kernel on x86
3. How I controlled the system during measurements to gain accurate power values
4. The "POLL" workload, which the framework uses when measuring active cores
5. The measuring procedure when using an external power measuring device

Please note that I will occasionally refer to a separate version of the framework. This is because the changes made for abolishing the custom kernel on x86 were incompatible with the AMD processor used in this thesis; see Section 4.2. As such, I could not integrate them into the main implementation without dropping AMD support, leading to a separate version besides the main one. In the rest of this thesis, I will always make clear when I am talking about the version supporting an unmodified kernel on x86 instead of the main implementation.

4.1 Accurate Wake-up Latencies

Section 3.1 described the concept behind integrating wake-up latency measurements into the existing measurement procedure. In a nutshell, the wake-up interrupt by the HPET causes the leader core to leave its sleep state and enter the associated interrupt handler. Here, the framework takes a timestamp. Comparing this timestamp with the time we programmed the interrupt to fire then allows calculating the wake-up latency. There are some factors introducing inaccuracy, however. First, we should consider the interrupt routing overhead. As the HPET is located in the chipset, any interrupt it generates needs to pass through the Advanced Programmable Interrupt Controller (APIC) to reach the target processor core; see Figure 2.2. This probably takes time. Another factor is the time spent by the Linux kernel before it actually hands control to the interrupt handler of the framework. Registering the wake-up interrupt's handler does not happen directly with the hardware. Instead, as the wake-up interrupt is delivered as a Non-Maskable-Interrupt (NMI), the `register_nmi_handler()`[1] interface of the Linux kernel acts as an intermediary, appending the framework's handler to a list of

[1] `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/arch/x86/include/asm/nmi.h?h=v6.2.9#n47`

other NMI handlers. Upon an NMI, Linux consecutively executes these handlers, which would add overhead if our handler came last. For this reason, the framework passes the `NMI_FLAG_FIRST` flag to `register_nmi_handler()`, requesting the kernel to execute our handler first. This should reduce the overhead. Nonetheless, there is still code that is executed by the kernel before it hands control to the framework's NMI handler, where we can take the timestamp. The delay introduced by this kernel code increases the measured wake-up latencies. Section 5.5 will later confirm that overhead from interrupt routing and interrupt handling inside the kernel is noticeable, as minimizing that overhead by switching to the Local APIC Timer significantly lowers the observed wake-up latencies.

Since the overhead from interrupt routing and handling is noticeable, fundamentally eliminating it would be preferable. As was already hinted at, exchanging the HPET for the Local APIC (LAPIC) Timer comes close to achieving this. Section 3.7 already described that having the LAPIC Timer generate the wake-up interrupt is the main change that enables the framework to run on an unmodified kernel. The location of the LAPIC very close to the core — see Figure 2.2 — already minimizes the overhead from interrupt routing. Because the LAPIC Timer can not generate NMIs [14, ch. 11 p. 13], though, other changes to the framework become necessary. Namely, normal interrupts — which are deactivated during the measurement by clearing the Interrupt Flag (IF) — had so far not been enabled to wake the leader core. I changed this by using the optional extension for `mwait` that enables masked interrupts as wake-up events; see Section 2.2. Furthermore, the wake-up interrupt's handler could no longer be an NMI handler. Instead, I could have simply registered a handler for the normal timer interrupt. There was a more interesting, second option, however. It resulted from still having the IF cleared during the measurement, while also using a normal interrupt to wake the leader core. In this situation the leader core wakes up from the interrupt, but can't enter a handler for it as the IF prevents this. Instead, code execution on reentering C0 resumes directly after `mwait`, avoiding any kernel overhead for code we place here. Because of this, I ultimately chose this second option for the version that supports an unmodified kernel on x86, placing the timestamp code after `mwait`. Section 5.5 will show that this version of the framework produces the wake-up latency measurements with the highest accuracy for Intel in this thesis.

When it comes to ARM, overhead in the wake-up latency measurements is a minor issue. Similar to the x86 implementation without a custom kernel, the wake-up interrupt originates from the core-adjacent Generic Timer. Thus, interrupt routing should be very quick. Also, during the measurement on ARM, the framework sets a number of flags that act comparably to the IF on x86, disabling interrupt handling. These flags do not keep cores from waking up on an interrupt, though. Because of this, code execution upon receiving the wake-up interrupt resumes directly after `wfi`, where we take the timestamp, excluding any kernel overhead. Section 5.4 will show that measured wake-up latencies on ARM are minuscule, which suggests that any overhead has to be tiny, too.

4.2 Unmodified Kernel on x86

As Section 3.7 described, the main change that allows the framework to run on an unmodified Linux kernel on x86 is to replace the High Precision Event Timer (HPET) with the Local APIC (LAPIC) Timer as the source of the wake-up interrupt. On the two Intel processors I tested for this thesis, an Intel Core i7-4790 and an Intel Core i7-6700K, this was straightforward. The reason is that these processors support "TSC-deadline mode", a feature that I already mentioned in Section 2.4.3. TSC-deadline mode allows to write an absolute timestamp into a Machine Specific Register. The LAPIC Timer then fires an interrupt once the TSC reaches this timestamp. [14, ch. 11 pp. 17–18] Since this is very similar to how the HPET functions — as described in Section 2.4.1 — it can be seamlessly replaced by the LAPIC Timer. Section 5.5 will show that the version without custom kernel on x86 works well on both tested Intel processors.

The AMD processor I tested, a Ryzen 5 5600G, does not support TSC-deadline mode, however. As the LAPIC Timer's documentation in the *AMD64 Architecture Programmer's Manual, Volume 2: System Programming* [1] does also not make any mention of a comparable feature, it may even be fully Intel-specific for the moment. While it should be possible to emulate TSC-deadline mode using other operating modes of the LAPIC Timer and overcome the need for a custom kernel on AMD, too, I did not do so for this thesis. Because of this, integrating the changes for abolishing the custom kernel into the main version of the framework would have meant dropping support for AMD. This is why I kept the version without need for a modified kernel on x86 separate.

4.3 Controlling the System

To gain accurate energy consumption values, the measured system needs to spend as much of the measurement period as possible in the state we want to measure. With respect to measuring a sleep state this mainly means for all cores to stay in that sleep state without waking up. I previously referred to this as "controlling the system". The implementation phase revealed significant differences regarding the wake-up behavior between different manufacturers and architectures, which I will discuss in this section.

4.3.1 Intel

Based on the two Intel processors I used during implementation, an Intel Core i7-4790 and an Intel Core i7-6700K, Intel seems to have a very uniform implementation of C-states, as they can all be entered using the `monitor`/`mwait` instructions. As such, when the Interrupt Flag (IF) is cleared, an interrupt will generally not wake a core; see Section 2.2. A Non-Maskable-Interrupt (NMI) will, however. This allows to use an approach comparable to a whitelist. We simply block all unwanted interrupts by clearing the IF, while explicitly sending the wake-up interrupt as an NMI. This approach is relatively easy to implement and has been shown to work very well, e.g. in "Sleep Well" [20].

4.3.2 AMD and ARM

In contrast, the AMD processor used in this thesis, a Ryzen 5 5600G, does not use `monitor/mwait` to enter every C-state. Instead, you request deeper C-states by reading from designated I/O ports. While the behavior of `monitor/mwait` on AMD is the same as for Intel, this is not the case for C-states entered by I/O port. Here, the IF only decides whether to enter an interrupt handler, but does not avoid the wake-up [3, p. 70][2].

On ARM the behavior is similar to the I/O-based C-states on AMD. An interrupt will wake a core, even when interrupts are deactivated [4, p. 6119].

As a consequence of these differences, the whitelisting approach on Intel is impossible to use on ARM and only partially applicable to AMD. Instead, we need to keep each individual interrupt from reaching the core, resulting in an approach more akin to blacklisting.

On x86, the Advanced Programmable Interrupt Controller (APIC) manages interrupts, so this is where we need to block unwanted interrupts during measurements. The preparation of the I/O APIC (IOAPIC) in the customized part of the kernel already includes masking all interrupts except the wake-up interrupt. Thus, any remaining interrupts would need to come from some Local APIC (LAPIC). Consequently, I also masked all local interrupts on each LAPIC by adjusting the associated memory-mapped configuration registers. This rules out interrupts like from the LAPIC Timer. The last relevant source of interrupts are Inter-Processor Interrupts (IPIs), but as we control all cores during the measurement, there is no core in the system that could generate an unexpected IPI. Thus, these measures rule out all possible interrupts on x86.

On ARM, I took similar measures. As the goal was to make do without customizing the Linux kernel, interfaces usable from a kernel module were the preferred solution. Fortunately, these do exist in the `disable_irq()`[3] and `disable_percpu_irq()`[4] functions. These interfaces allow temporarily disabling interrupts until a matching `enable_(percpu_)irq()` is called. However, under some circumstances Linux uses a lazy approach to disabling the interrupt, only internally marking it as disabled but leaving the hardware untouched until an actual interrupt occurs. This would not help with preventing wake-ups. To certainly avoid this lazy approach, I told the kernel not to use it by setting the `IRQ_DISABLE_UNLAZY` flag for each interrupt. Using these interfaces, I then disabled all interrupts on ARM.

Nonetheless, even with these measures in place, there were still too many wake-ups to gain useful measurement values. Because of the general complexity of computers and the lack of traceability for wake-ups, controlling them proved tedious at times. Even small changes to mostly unrelated code, like marking a function as `inline`, could suddenly drastically increase the number of observed wake-ups and reduce the quality of the measured values. I observed this behavior on both the AMD and the ARM systems I

[2] The cited documentation is relatively old and technically not applicable to the Ryzen 5 5600G. AMD's official documentation hub did not offer more recent documentation. I observed the relevant statements made by the documentation to still be valid, however, as Section 5.3 will confirm.

[3] e.g. `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/kernel/irq/manage.c?h=v6.2.9#n728` for Linux kernel version 6.2.9.

[4] e.g. `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/kernel/irq/manage.c?h=v6.2.9#n2430` for Linux kernel version 6.2.9.

used for testing. The actual causes of these drastic changes in wake-ups are unfortunately still a mystery. However, through mostly trial and error, I could reduce wake-ups to an acceptable level. The various measurements in Chapter 5 will show that wake-ups were ultimately not an issue, as measured values overwhelmingly were very stable. Still, I can not claim to completely understand every detail of the wake-up behavior of these CPUs.

4.3.3 Unmodified Kernel

For the version of the framework that supports an unmodified kernel on x86, the whitelist approach used on Intel so far is not possible. This is because the LAPIC Timer replaces the HPET; see Section 3.7. The LAPIC Timer can not generate NMIs [14, ch. 11 p. 13]. Because of this, the leader core uses the optional extension to `mwait` that allows it to wake up on normal interrupts, even when the IF is cleared; see Section 2.2. This effectively makes the wake-up behavior of the leader core the same as that of the I/O-based C-states on AMD. Consequently, the leader core may wake up more often than before as any interrupt can now cause it to leave its sleep state. To minimize wake-ups, interrupts have to be kept from reaching the leader core, as was already done for AMD and ARM. While I was able to reuse the code masking the LAPIC interrupts introduced for AMD, the code for masking interrupts in the IOAPIC was part of the abolished customizations to the Linux kernel. Using the `disable_irq()` interface like on ARM, it was possible to suitably replace this code, however. As Section 5.5 will show, these measures are sufficient to keep wake-ups to a minimum.

4.4 The “POLL” Workload

When it comes to measuring energy consumption, it has to be clear what we actually measure. Otherwise, we can not learn much from the results. For sleep states this is easy, as they put the core into an inactive state where it is not executing instructions. Because of this, sleep states fully define the state of the core. On x86 this applies to all deeper C-states starting from C1. In C0, however, the core is actively executing code. Accordingly, depending on what the actual workload is, the energy consumption associated with C0 can vastly differ. In the paper “Introducing FIRESTARTER” [8] for example, Hackenberg et al. compared various workloads for maximizing energy consumption. Despite this shared goal, actual energy consumption still changed significantly between workloads, as utilization of components varied. [8, pp. 4–7] As such, when measuring energy consumption for C0 to get a better frame of reference for the other values, we need to clearly define the executed workload. In “Sleep Well” [20], the workload for C0 was executing `monitor/mwait` while requesting `mwait` to enter C0 by using a special hint. This, however, caused the `mwait` instruction to quickly terminate, so effectively, the framework just executed `monitor/mwait` over and over again until the measurement finished. [20, p. 7] While it might be an interesting look into the behavior of `monitor/mwait` for this special hint, this workload does not make much practical sense overall, as this is nothing that should ever occur in the real world.

Accordingly, this section defines a simpler, more relevant workload which the framework will use during subsequent measurements. I will refer to this workload as “POLL”. On

x86, cores executing POLL enter a while-loop, where they repeatedly check a variable and then increment a counter that tracks how often they iterated through the loop. Cores leave this loop once the wake-up interrupt from the HPET fires, which causes the interrupt handler to rewrite the loop-condition variable.

On ARM, the same general problem is present, since we need some measurements of active cores to put the observed values for `wfi` into perspective. As the setup on ARM is somewhat different, the exact workload from x86 is not possible. This is because the framework does not register an interrupt handler that could change the polled variable. As a solution, instead of checking a variable, POLL repeatedly reads the System Counter (SC), exiting once it equals or exceeds the deadline where the measurement ends. For the version of the framework that runs on an unmodified kernel on x86, no interrupt handler is present either. Because of this, the framework employs the same solution as on ARM, repeatedly reading the TSC instead of the SC, however.

In the rest of this thesis, mentions of POLL and measurements of C0 will always refer to these specific workloads, unless explicitly stated otherwise.

4.5 External Power Measurements

To help understanding the integration of external power measurements as described in Section 3.6, I will outline the concrete procedure used by my implementation of the framework. Specifically, I will describe the measurement of a Raspberry Pi 5, as this is what I tested the external measurements on. In this section, I will repeatedly use the expressions “Controllbox” and “Measurebox”. They respectively refer to the computer from which the measurement is remotely controlled, and the system that is to be measured. Figure 4.1 shows an overview of the general procedure.

The external power measurements can come from any source, as long as they are in the form of a log of timestamps and their associated power values. The external measuring device I used for measuring the Raspberry Pi 5 was an ODROID Smart Power. This specific power measuring device does not have the capability to automatically collect power values into such a log. This isn’t a problem however, as there is a command-line tool[5] which allows to read the current measured power value from the ODROID Smart Power. Using this tool, I created a script which can automatically collect power values for any given timespan. This lead to a measurement setup where, after the main script orchestrating the measurement is initiated on the Controllbox, it starts the power logging script in the background. For the entire duration of the measurement, this background script will continuously log the measurements from the external power measuring device. After starting the power logging script, the Controllbox initiates the measurement procedure on the Measurebox. However, because we are using external measurements, the actual measurements are not the first action to take on the Measurebox. Instead, the framework needs to create a unique pattern in the energy consumption and save its start timestamp, as was described in Section 3.6. We will later use this information for associating values in the power log with individual measurements. The signal pattern in our case simply consists of first having all cores be active executing

[5] `https://github.com/mhaehnel/SmartDroid`

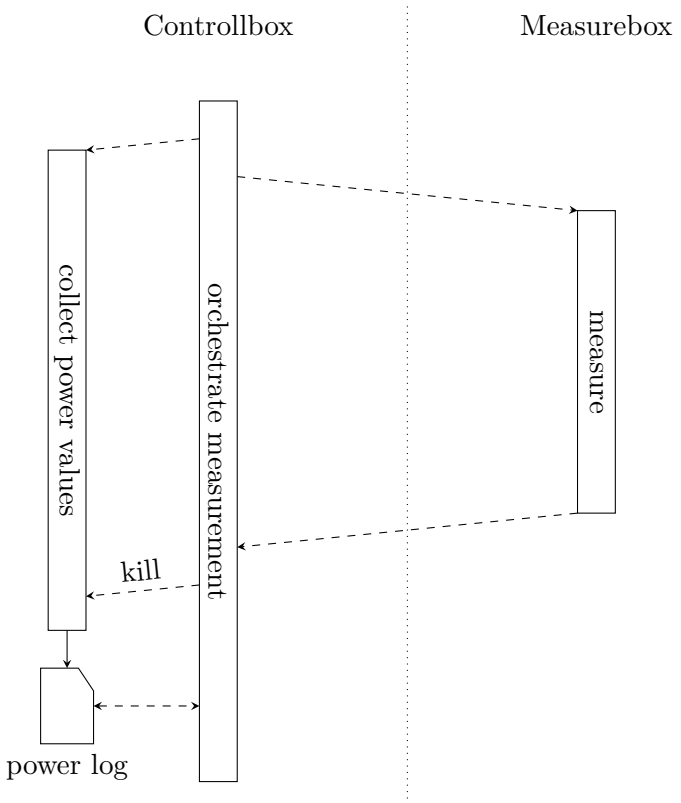


Figure 4.1: Overview of the measurement procedure when using external measurements.

the POLL workload and then having all cores sleep using `wfi` for 1 second, respectively. This is then repeated another time, just to have a more unique pattern. For this thesis, I did not conduct any external measurements on x86, but there is support for this as well, exchanging `wfi` for `mwait` with the hint for C1. Once the signal pattern has been generated, the actual measurements start, each saving its start and end timestamps. After the last measurement finishes and the measurement procedure on the Measurebox terminates, the main process on the Controllbox tells the background script to stop collecting power values and write its gathered power log to a file. The main script uses the "kill" command for this communication, sending a "SIGINT" signal to the background process. Using the file containing the power log and the data collected on the Measurebox, the Controllbox now starts post-processing. This last step first searches the generated signal pattern in the power log. Once it is found, the timestamp the Measurebox saved when starting the pattern can be associated with the start time of the pattern in the power log. Working forward from this point, power values from the log can then be associated with individual measurements by using their saved start and end timestamps. By the end of this step, each measurement should have one or multiple power values from the log associated with it. Taking the mean of these values then yields the final result.

5 Evaluation

To prove that my implementation of the framework works, I applied it to a variety of systems. These were:

1. Two systems using Intel CPUs, an Intel Core i7-4790 and an Intel Core i7-6700K, respectively.
2. A system using a Ryzen 5 5600G, which is a CPU by AMD.
3. A Raspberry Pi 5, which uses a Cortex-A76 processor of the ARM architecture.

On each of these systems, I measured sleep state energy consumption and wake-up latencies. On the x86 systems, I additionally gathered wake-up latency values for waking a core by triggering `monitor`. I will discuss all gathered values and use them to prove that my measurement framework works. I will also point out interesting observations from comparing different measurements.

Apart from these main measurements, I will also introduce the latency measurements I collected previously for the paper "Sleep Well" [20], for which I used an earlier version of the framework. I will contrast these values with the new ones, discussing possible reasons for the significant differences.

To prove the effectiveness of the changes I made to abolish the custom kernel on x86, I tested them on the two Intel systems. I will compare the wake-up latencies gained from these tests to those by the main version of the framework, substantiating the suspected reduction in overhead.

Lastly, I will compare the measured values to associated default values used by the measured systems' operating systems and discuss possible benefits of using the framework's values instead.

All measurements, excluding the ones that explicitly state differently, used the same version of the measurement framework[1]. To make measurements more comparable, I used the same kernel — same source code, same configuration — on all x86 systems. This only excludes the measurements from "Sleep Well" [20] and the measurements done on an unmodified kernel. The specific kernel used across all x86 systems is based on Linux kernel version 6.2.9 and contains slight modifications necessary for the framework[2]. During all measurements, to rule out interference from changing frequencies due to DVFS, I pinned all processor cores to the highest possible frequency using the "performance" frequency governor. By default, the framework does 10 consecutive measurements for

[1] The main version of the framework can be found under `https://github.com/obibabobi/MWAITmeasurements/tree/Masterarbeit`

[2] The associated modified kernel code can be found in the submodule "linuxMWAIT" within the git repository of the framework for each framework version that requires it.

each specific system state to be measured. Because of this, each separate result presented here, e.g. concerning a specific C-state, consists of 10 distinct measurement values. Each figure will show all these values, as well as their mean. Please also note that I rounded all shown measurement values to 3 significant figures. As the precision of the wake-up latencies depends on the frequency of the counters used in their calculation, Table 5.1 shows all relevant counter tick periods on the test systems.

	TSC	**HPET**
Intel Core i7-4790	$\approx 0.278\,\text{ns}$	$\approx 69.8\,\text{ns}$
Intel Core i7-6700K	$\approx 0.250\,\text{ns}$	$\approx 41.7\,\text{ns}$
Ryzen 5 5600G	$\approx 0.257\,\text{ns}$	$\approx 69.8\,\text{ns}$
	System Counter	
Cortex-A76	$\approx 18.5\,\text{ns}$	

Table 5.1: Tick periods of all relevant counters of the test systems. Systems are designated by their processor.

5.1 Latencies from "Sleep Well"

Using the approach presented in Section 3.1, I already gathered wake-up latencies for the Intel Core i7-4790 and Intel Core i7-6700K at an earlier point. These results were incorporated into the paper "Sleep Well" [20]. Section 2.5.2 already presented the associated power values from these measurements; see Figures 2.4 and 2.5. Note that, as I conducted these measurements quite a bit earlier than the others presented in this chapter, they did still use a different version of the framework and modified kernel[3]. Also, the kernel configuration used for these measurements was different.

Figures 5.1 and 5.2 show the gathered wake-up latencies. As with the power measurements, the observed latencies are very stable across measurements and seem plausible, with deeper C-states needing more time until the system is fully operational again.

The only unexpected aspect is the massive increase in latencies from the older Intel Core i7-4790, a Haswell processor, to the newer Intel Core i7-6700K of the Skylake microarchitecture. Even in C0, where close to the entire measured latency should come from the overhead of the interrupt, the time between the firing of the wake-up interrupt and the control flow reaching its interrupt handler increases from $\approx 3.23\,\mu\text{s}$ to $\approx 236\,\mu\text{s}$. "Sleep Well" [20] did not go into much detail on possible reasons, simply speculating that changes in the microarchitecture might have had detrimental effects. Taking new results from this thesis into account, the kernel seems a more likely cause, however; see Section 5.2.1. Also, as with the power measurements, Intel Machine Specific Registers (MSRs) imply that no deeper Package C-states than PC2 were entered. If these statistics are to be believed, the results do not give insight into the possible effects of these deeper PC-states.

[3] The version of the framework used in "Sleep Well" [20] can be found under `https://github.com/tud-os/sleepwell`

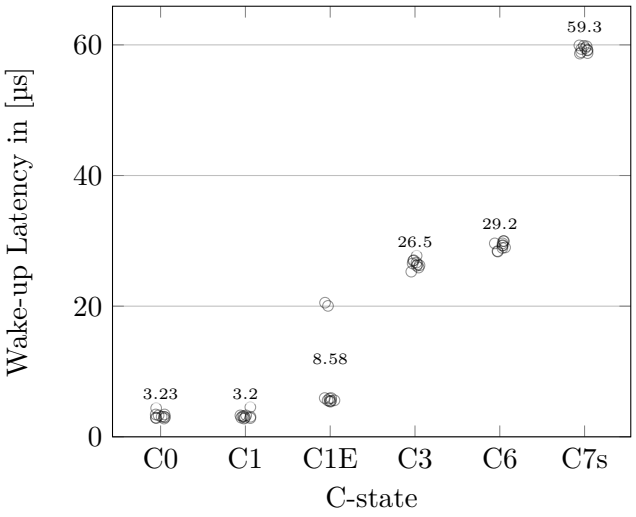


Figure 5.1: Wake-up latencies of different C-states supported by the Intel Core i7-4790 as measured for “Sleep Well” [20].

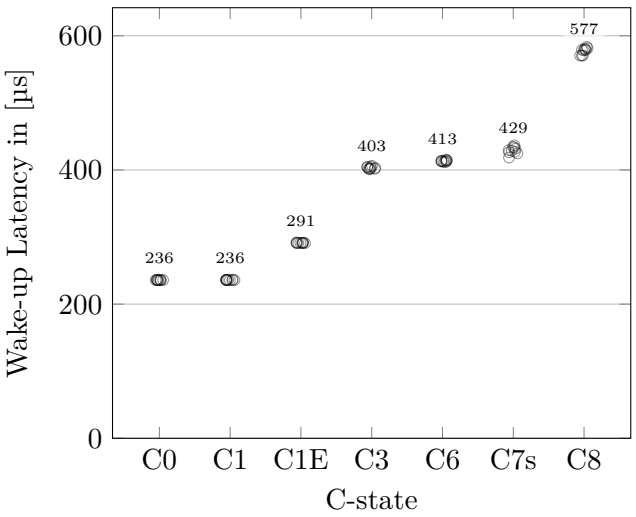


Figure 5.2: Wake-up latencies of different C-states supported by the Intel Core i7-6700K as measured for “Sleep Well” [20].

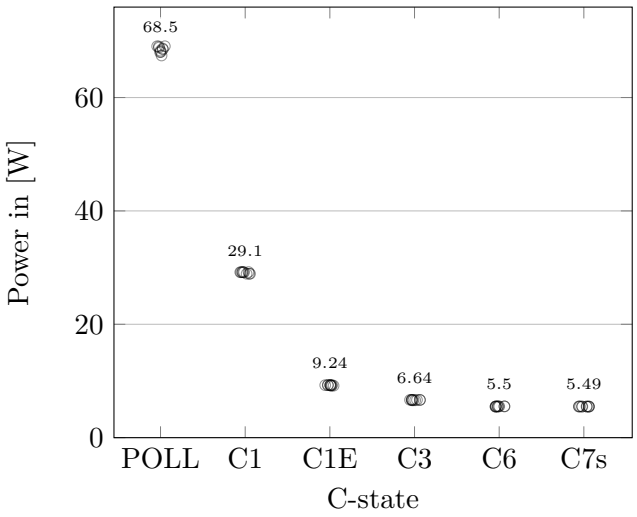


Figure 5.3: Processor energy consumption of different C-states supported by the Intel Core i7-4790. POLL workload shown for C0.

5.2 Intel

Using the implementation of the framework I created for this thesis, I remeasured the exact systems from "Sleep Well" [20]. The specific CPUs in question are again an Intel Core i7-4790 as well as an Intel Core i7-6700K. As the observed wake-up latencies were significantly different when compared to the earlier measurement in "Sleep Well" [20], I will first present the new results and discuss possible reasons for this change. After that, I will look into the latencies when waking cores by writing to their `monitor`ed memory region.

As for the power values gathered from these measurements, they were largely the same as earlier in Section 2.5.2, so I will not discuss them in as much detail. Nonetheless, for the sake of completeness, the new results are also plotted in Figures 5.3 and 5.4. One noticeable but expected difference concerns C0 (now named POLL), as the actual executed workload is different now. The energy consumption incurred by the POLL workload as described in Section 4.4 is $\approx 68.5\,\mathrm{W}$ and $\approx 52.7\,\mathrm{W}$ for the Intel Core i7-4790 and Intel Core i7-6700K, respectively. For the Intel Core i7-4790, the only other noticeable difference is that the measured energy consumption for C3 dropped from $\approx 7.52\,\mathrm{W}$ to $\approx 6.64\,\mathrm{W}$. For the Intel Core i7-6700K, energy consumption for C1E rose from $\approx 1.66\,\mathrm{W}$ to $\approx 1.91\,\mathrm{W}$, while for C3, C6 and C7s it dropped by $\approx 9\,\%$ to $\approx 1.44\,\mathrm{W}$. Finally, for C8, it decreased a bit more strongly from $\approx 1.18\,\mathrm{W}$ to $\approx 1.02\,\mathrm{W}$. The reason for these moderate changes is unclear. It may be found with the different kernel config that was used for these measurements. The next section will discuss this in more detail, as I will look into the more significant changes in observed wake-up latencies.

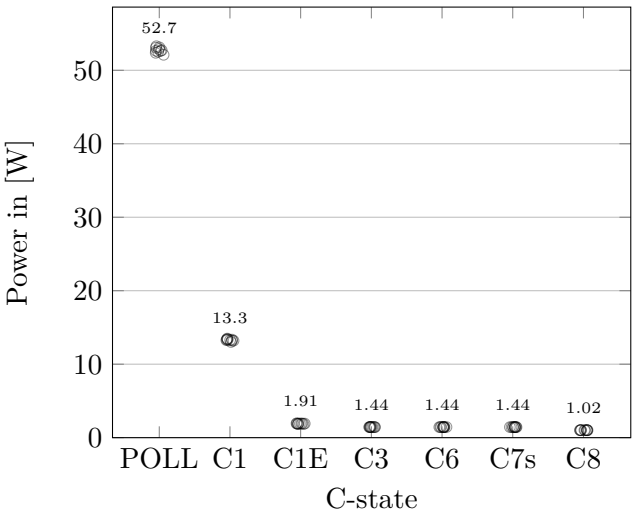


Figure 5.4: Processor energy consumption of different C-states supported by the Intel Core i7-6700K. POLL workload shown for C0.

5.2.1 Wake-up Latencies

To measure the presented wake-up latencies, I used the methodology described in Section 3.1. The results for the Intel Core i7-4790 and the Intel Core i7-6700K are visualized in Figures 5.5 and 5.6 respectively.

For the Intel Core i7-4790 results appear very similar to the ones observed previously in "Sleep Well" [20]. Compare Figure 5.1 for details. For example for C0, where the entire latency should come from the overhead of the wake-up interrupt, I observed a latency of $\approx 3.00\,\mu s$, compared to $\approx 3.23\,\mu s$ previously. Results for the other C-states are also similar, even being alike in the ramp-up effect of C1E's measurements. The only differences worth mentioning are outliers for C3 and C6 which I did not observe before. While the measured latency for C3 is nearly unchanged at $\approx 26.4\,\mu s$ for 8 of the 10 measurements, it is significantly lower for the other 2 measurements, at $\approx 20.0\,\mu s$ and $\approx 17.6\,\mu s$, respectively. For C6 the measured latency is $\approx 30.8\,\mu s$ for nine measurements, with one measurement reporting $\approx 22.8\,\mu s$, however. It is unclear why these outliers were not observed before, as no unexpected wake-ups occurred during either of these measurements, and other metrics like Core and Package C-state residencies remain unchanged. With a sample size of 10, it might just be chance, however. Another possible cause is the change in kernel between these measurements. When these outliers are ignored, the results remain practically the same compared to "Sleep Well" [20], with changes in measured latencies being within a few percent.

The remeasuring results for the Intel Core i7-6700K offer more grounds for an in-depth discussion. When comparing them to the previous results in Figure 5.2, the lower baseline becomes immediately obvious. Where I previously observed the interrupt overhead in C0 to be $\approx 236\,\mu s$, it is now a much more moderate $\approx 11.1\,\mu s$. For the other C-states, similarly drastic reductions in latency can be observed. For C1 latencies drop from $\approx 236\,\mu s$ to $\approx 11.1\,\mu s$ as well, while for C1E, the drop is even starker, going from $\approx 291\,\mu s$

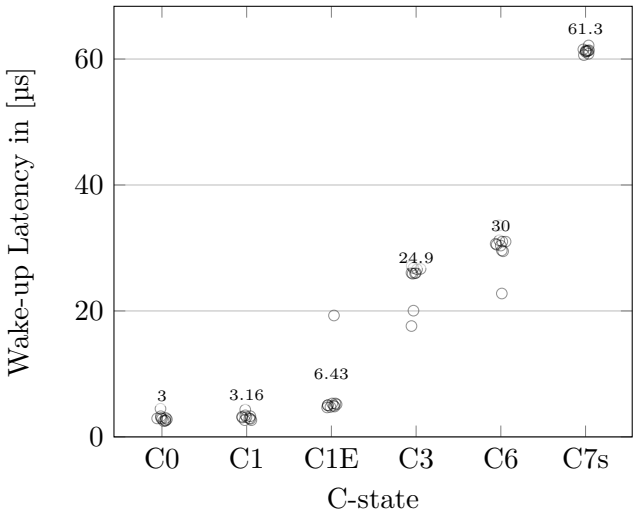


Figure 5.5: Wake-up latencies of different C-states supported by the Intel Core i7-4790.

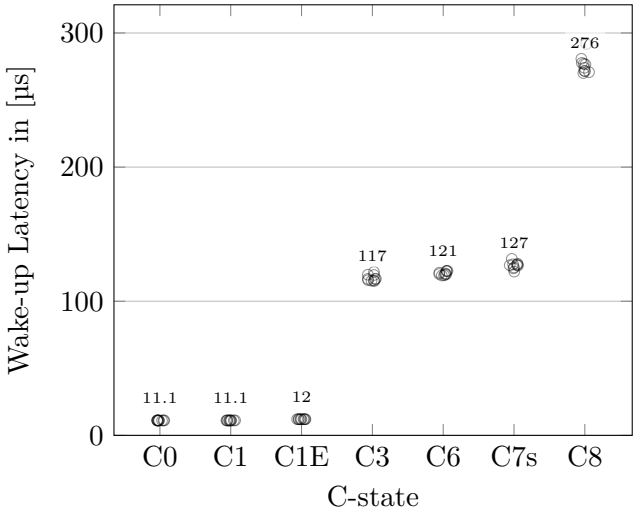


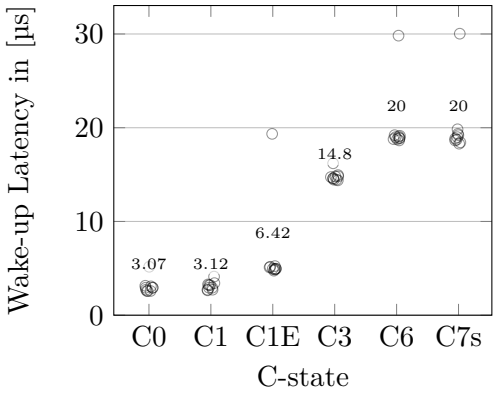
Figure 5.6: Wake-up latencies of different C-states supported by the Intel Core i7-6700K.

to ≈ 12.0 µs. Further changes are from ≈ 403 µs to ≈ 117 µs for C3, from ≈ 413 µs to ≈ 121 µs for C6, from ≈ 429 µs to ≈ 127 µs for C7s and from ≈ 577 µs to ≈ 276 µs for C8.

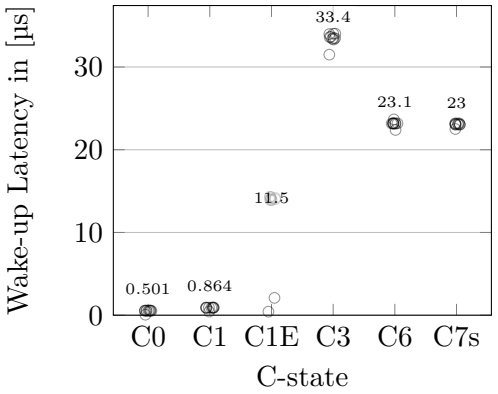
Keep in mind that I measured the exact same system as in "Sleep Well" [20], which rules out hardware differences as possible cause for these changes. Additionally, the kernel version is still the same, with the custom parts having only insignificantly changed[4]. What did change between these measurements, however, was the kernel configuration. Because of this, it seems likely that some option in the old configuration drastically increased the overhead of NMI handling in the kernel, leading to the inflated latency values observed previously. This underlines the necessity of doing such measurements using the same kernel to achieve comparable results. As such, it is the main reason why I kept the kernel as stable as possible across all x86 measurements in this chapter. However, it would be even better to eliminate interrupt handling overhead completely, removing the operating system as a variable when looking at the measured latencies. The proposed changes for supporting an unmodified kernel are promising in this regard, as I explained in Section 4.1. I will evaluate these changes in Section 5.5.

While it is impossible to say with certainty without analyzing the difference in both configurations and their resultant kernel code, it seems likely that the old configuration caused the kernel to execute some additional code upon receiving the NMI. This would delay handing control to the NMI handler registered by the framework where the timestamp deciding the latency is taken. However, every one of those latency drops apart from C1 is significantly greater than the simple decrease of interrupt handling overhead of ≈ 225 µs observed in C0. Starting with C3, an explanation is relatively easy to come up with, as those C-states flush the L1 and L2 caches; see Section 2.1. Code that is executed after the core wakes up from these deep C-states would take longer than for the core already being in C0, as these caches would need to be repopulated in the process. This also fits nicely with C1 showing the exact same change in latencies as C0, since C1 does not flush any caches. C1E remains a mystery, however. It does not flush caches either, but latency still drops by ≈ 279 µs, which is significantly more than the expected ≈ 225 µs. C1E is now very close in latency to C1, where previously it had a quite significant additional latency cost. As the hardware is the same, differences in the speed of changing frequency and voltage seem unlikely. As such, there does not seem to be any obvious reason why it had such significant additional cost before. The new measurement seems more likely to be accurate though, as the ACPI specification states about C1 that "[t]he hardware latency of this state must be low enough that OSPM does not consider the latency aspect of the state when deciding whether to use it" [21, p. 476]. C1's wake-up latency should thus be negligible, which should also apply to C1E as a C1 sub-state.

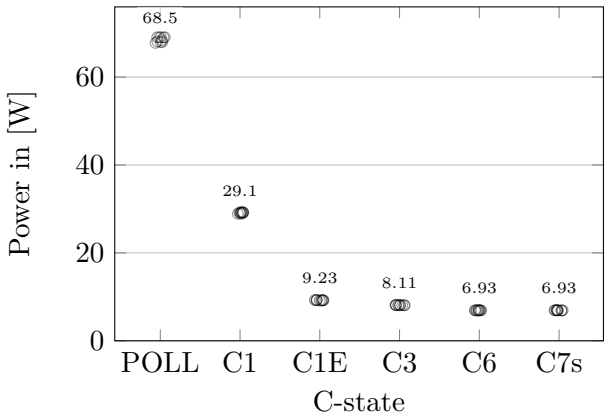
[4] I ruled out changes to the custom kernel code as the reason through a separate measurement. I used the version of the framework from "Sleep Well" [20] as well as the associated custom kernel with the new kernel config, producing results very similar to Figure 5.6.



(a) Wake-up latencies for wake-up by HPET interrupt.



(b) Wake-up latencies for wake-up by write to `monitor`ed memory.

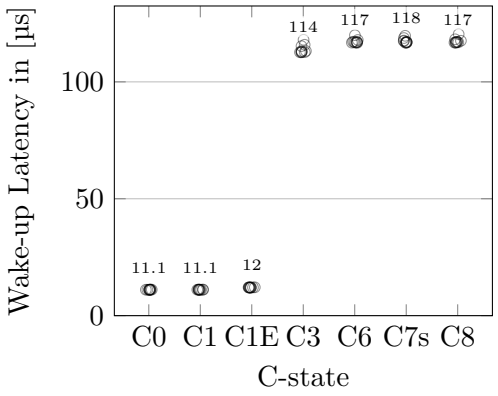


(c) Processor energy consumption. POLL workload shown for C0.

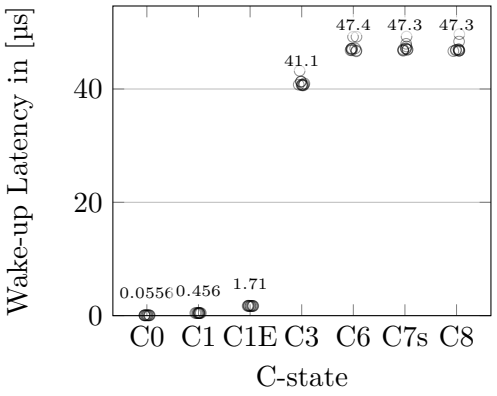
Figure 5.7: Core C-state characteristics of the Intel Core i7-4790. For this measurement, Package C-states were completely disabled.

5.2.2 Wake-up Latencies when writing to `monitor`ed memory

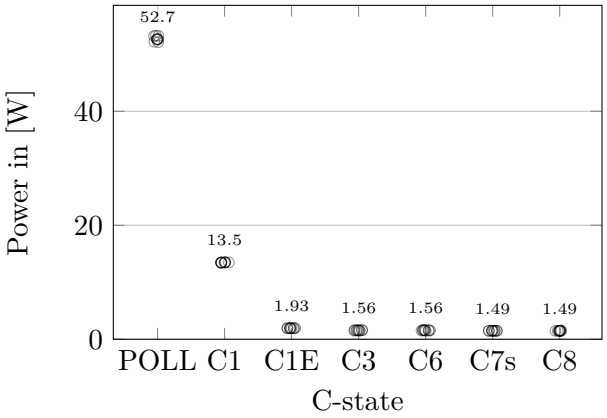
I also evaluated both Intel systems using the methodology introduced in Section 3.2. As for triggering the wake-up by writing to memory one core needs to already be active, Package C-states (PC-states) can not be entered on systems with only one package. Because of this, getting insight into the latency associated with PC-states would require a multi-socket machine. Both Intel systems only feature one socket, however. To still be able to meaningfully compare latencies between the wake-up methods of triggering `monitor` and receiving an interrupt by the HPET, I remeasured wake-up latencies for the HPET with PC-states completely disabled. I verified that no PC-states were entered during the measurements by looking at special residency MSRs provided by Intel. All results of these measurements where PC-states were disabled can be found in Figure 5.7 for the Intel Core i7-4790, and Figure 5.8 for the Intel Core i7-6700K.



(a) Wake-up latencies for wake-up by HPET interrupt.



(b) Wake-up latencies for wake-up by write to `monitor`ed memory.



(c) Processor energy consumption. POLL workload shown for C0.

Figure 5.8: Core C-state characteristics of the Intel Core i7-6700K. For this measurement, Package C-states were completely disabled.

The measured wake-up latencies when writing to `monitor`ed memory to wake other cores contain some surprising results. In general, my expectation was an overall reduction in latencies when compared to triggering the wake-up by sending an interrupt from the HPET. My reasoning behind this was the higher overhead of the interrupt, coming from a device in the chipset and having to pass through the I/O APIC and Local APIC before being able to actually trigger the wake-up; see Figure 2.2 for a rough overview of these components. Additionally, with the version of the framework I used for this measurement, the kernel executes some interrupt handling code before control flow reaches the handler of the framework. This delays the timestamp that ends the measured wake-up period. All this should contribute to increasing the measured latencies for the HPET. Compare this to waking by write, where the source of the wake-up is local to the processor and the timestamp is taken immediately once the `mwait` instruction terminates upon reaching C0. All in all, shorter wake-up latencies for waking a core through triggering `monitor` seem plausible.

I did not observe shorter wake-up latencies for triggering `monitor`, however, at least not across the board. Let us first look at the wake-up latencies for the Intel Core i7-4790. Remember that in C0 the framework executes the "POLL" workload described in Section 4.4, which repeatedly checks a variable. While before, latencies in C0 measured the time until reaching the wake-up interrupt handler, it now is the time between the leader core rewriting POLL's variable and this change becoming visible to the target core, causing it to leave POLL. For C0, results are in line with my expectations, going from a latency of $\approx 3.07\,\mu s$ for waking by interrupt to $\approx 0.551\,\mu s$, ignoring an outlier in the first measurement at $\approx 0.0570\,\mu s$. This significant drop in latency could be explained by the factors discussed before, namely the lack of interrupt routing and handling. For C1, latencies drop from $\approx 3.12\,\mu s$ to $\approx 0.911\,\mu s$, also exluding an outlier in the first measurement at $\approx 0.439\,\mu s$. This, too, is in line with expectations. For C1E, however, latencies actually rise from $\approx 4.99\,\mu s$ to $\approx 14.1\,\mu s$. Again, we have to exclude some noticeable outliers from the averages. For the first measurement, latency for wake-up by interrupt is unusually high at $\approx 19.3\,\mu s$, while it is exceptionally low for wake-up by write at $\approx 0.429\,\mu s$. Apart from another outlier for wake-up by write at $\approx 2.10\,\mu s$, measurements are very stable, however. For C3 ($\approx 14.8\,\mu s$ to $\approx 33.4\,\mu s$), C6 ($\approx 18.9\,\mu s$ to $\approx 23.1\,\mu s$) and C7s ($\approx 18.9\,\mu s$ to $\approx 23.0\,\mu s$), wake-up latencies rise as well when comparing wake-up by interrupt to wake-up by write. For C6 and C7s I again excluded significant outliers in the first measurements for wake-up by interrupt, at $\approx 29.8\,\mu s$ and $\approx 30.0\,\mu s$, respectively. Even including these outliers, results for C6 and C7s are barely distinguishable. In general, there seems to be some ramp-up effect, as for all C-states the first measurement shows the highest latency for wake-up by interrupt while showing the smallest latency for wake-up by write, often with a significant margin. After that the values quickly stabilize. The reason for this behavior remains unclear. Nevertheless, when disregarding the outliers, the results are still very surprising. Not only do they show an increase in latency for the deeper C-states when waking by write instead of interrupt. They also do not fit with the general logic of C-states, were deeper C-states "buy" energy savings by increasing their wake-up latencies. This symmetry is broken by C6 and C7s using less power (both $\approx 6.93\,W$) in these measurements while having a lower wake-up latency ($\approx 23.1\,\mu s$ and $\approx 23.0\,\mu s$) than C3 ($\approx 8.11\,W$; $\approx 33.4\,\mu s$).

For the Intel Core i7-6700K I could not make the same observations. When comparing wake-up by interrupt to wake-up by write, latencies decrease for all C-states as predicted. For C0 the wake-up latency goes from ≈ 11.1 µs to ≈ 0.0556 µs, while for C1 it decreases from ≈ 11.1 µs to ≈ 0.456 µs and for C1E from ≈ 12.0 µs to ≈ 1.71 µs. For C3 it goes from ≈ 114 µs to ≈ 41.1 µs. The results for the deepest three C-states are close enough to be practically indistinguishable, with their latencies decreasing from ≈ 117/118 µs to ≈ 47.3/47.4 µs. These results do not indicate the same symmetry-breaking behavior as with the Intel Core i7-4790. Since I used the exact same code to measure both systems, the plausible results on the Intel Core i7-6700K give a little more credibility to the strange values observed on the older Intel Core i7-4790. Also, unlike with the previous measurement, there is no significant ramp-up effect. In general, waking a core of the Intel Core i7-6700K through a write to a `monitor`ed memory address is significantly faster than waking it through an interrupt generated by the HPET.

5.2.3 General Discussion

When surveying all the data I presented so far, an oddity catches the eye regarding the results for the Intel Core i7-6700K with Package C-states (PC-states) dis- and enabled. Compare Figures 5.6 and 5.8a, both dealing with wake-up latencies for wake-up by HPET interrupt, the second one having PC-states disabled, however. If the residency MSRs provided by Intel are to be believed, no deeper PC-state than PC2 is ever entered, even with all PC-states being enabled. As PC2 is only a transitional state and documentation doesn't list any specific energy saving measures for this state, the impact of disabling it should be minor if it is even present. The latency measurements do support this expectation for the most part. For C0, C1 and C1E disabling PC-states does not change latency results at all, as those states could not enter a PC-state even when they were enabled; see Section 2.1. For C3 (≈ 117 µs to ≈ 114 µs), C6 (≈ 121 µs to ≈ 117 µs) and C7s (≈ 127 µs to ≈ 118 µs) there is a noticeable, but small difference. This, too, is expected, as those states did enter PC2 before, but can't in the new measurement. C8, however, is curious in this regard. It's wake-up latency drops from ≈ 276 µs to ≈ 117 µs, making it barely distinguishable in latency from C6 and C7s where it had more than double their latency before. The associated power measurements show a similar picture. Compare Figures 5.4 and 5.8c. For C0, C1 and C1E, energy consumption barely changes, while for C3 (≈ 1.44 W to ≈ 1.56 W), C6 (≈ 1.44 W to ≈ 1.56 W) and C7s (≈ 1.44 W to ≈ 1.49 W) energy consumption increases slightly. Where before C8 was able to save a significant further percentage of this remaining energy consumption and lower it to ≈ 1.02 W, its new energy consumption is indistinguishable from C7s at ≈ 1.49 W. When consulting the documentation presented in Section 2.1, the results without Package C-states make sense, as Core C-states CC6 to CC8 should have very similar characteristics since there are no major, documented differences in their applied energy saving measures. As Intel does not go into detail on these C-states, especially not sub-states like C7s, small differences can be overlooked. What does not make sense, however, is that the characteristics of C8 change so drastically when PC-states are disabled. If truly no Package C-states deeper than PC2 had been entered as the residency MSRs suggest, the difference would not be close to this big. At this point I can only speculate, but these

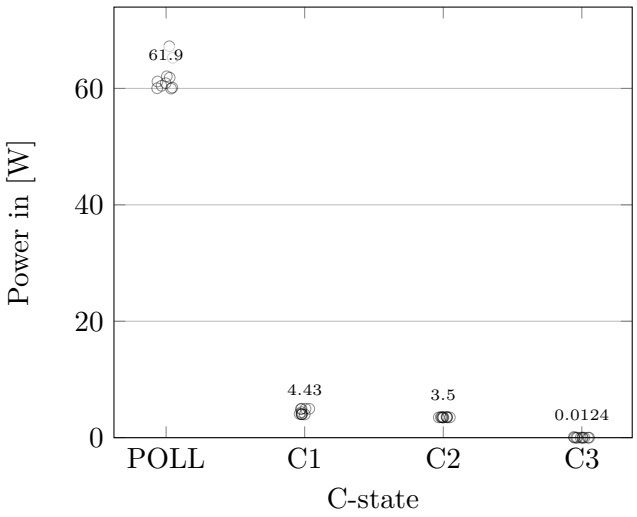


Figure 5.9: Processor energy consumption of different C-states supported by the Ryzen 5 5600G. POLL workload shown for C0.

results suggest that the Intel Core i7-6700K does enter deeper Package C-states than PC2, with the residency MSRs misreporting this, however.

5.3 AMD

Apart from the Intel processors, I applied the framework to a CPU by AMD as well. Specifically, I measured a system using a Ryzen 5 5600G. This section will go over the various results, starting with the observed energy consumption before presenting the wake-up latencies.

Figure 5.9 shows the energy consumption measured for the different C-states supported by the Ryzen 5 5600G. Notice how the C-states supported by this AMD processor are quite different compared to those supported by the Intel processors. In general, AMD seems to adhere more closely to the naming scheme outlined in the *Advanced Configuration and Power Interface (ACPI) Specification* [21] which defines C-states C2 and C3 as optional C-states [21, pp. 473–478], while Intel chooses to apply its own naming conventions. When executing the POLL workload, the energy consumption as reported by RAPL is $\approx 61.9\,\mathrm{W}$, with the biggest outliers being the first two measurements at $\approx 67.2\,\mathrm{W}$ and $\approx 65.2\,\mathrm{W}$, and all other measurements being within $2\,\mathrm{W}$ of the mean. For C1, the observed energy consumption already drops to a way lower $\approx 4.43\,\mathrm{W}$, and for C2 it decreases further to $\approx 3.50\,\mathrm{W}$. Finally, for C3, nine out of 10 measurements report a energy consumption of exactly $0\,\mathrm{W}$, with only one reporting $\approx 0.124\,\mathrm{W}$. While values for C2 are very stable with a standard deviation of $\approx 0.0334\,\mathrm{W}$ (the biggest outlier being $\approx 3.58\,\mathrm{W}$), measurements for C1 have more variance with a standard deviation of $\approx 0.459\,\mathrm{W}$. The reason for this stark decrease in variance and then the drop to exactly $0\,\mathrm{W}$ when going from C1 to C3 is not known, but can probably be found within RAPL. As mentioned in Section 2.3, AMD, unlike Intel, uses a model-based

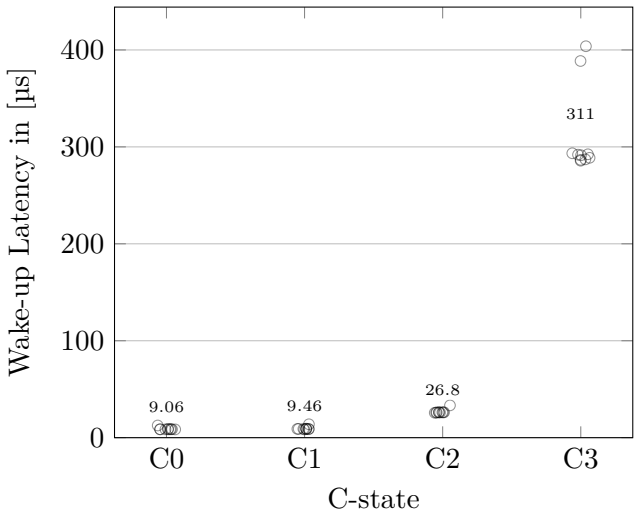


Figure 5.10: Wake-up latencies of different C-states supported by the Ryzen 5 5600G.

approach for calculating the energy values reported by RAPL. Presumably this model uses the executed instructions as an input, making it slowly break down when entering C-states where no instructions are executed. The one measurement for C3 where energy consumption was not 0 W may be explained by the update period of the RAPL registers of ≈ 1 ms. For this one measurement, an update may have occurred during the wake-up process right before the registers were read at the end of the measurement, making them report some of the energy consumption of the wake-up process. For the other measurements, no such update may have happened in that timeframe.

I measured wake-up latencies as well; see Figure 5.10 for the results. For C0 nearly all measured latencies are very close at around ≈ 8.68 µs, with the only significant outlier being the first measurement at ≈ 12.6 µs, leading to an overall average of ≈ 9.06 µs. Results for C1 look very similar, being quite stable around ≈ 8.97 µs for nine measurements, the first measurement also being an outlier at ≈ 13.8 µs, however. Overall, the mean is at ≈ 9.46 µs. The first noticeable rise in latencies occurs for C2, where most observed values are around ≈ 26.0 µs, again having the only outlier in the first measurement at ≈ 33.3 µs, which pushes the overall average to ≈ 26.8 µs. Lastly, for C3 the latencies increase starkly to ≈ 311 µs. This time there are two significant outliers at ≈ 404 µs and ≈ 389 µs, and unlike for the other C-states they don't come from the first measurements, but from the last two. All other values are very stable around ≈ 290 µs. All in all, there seems to be some ramp-up effect, clearly affecting the series of measurements for C0 to C2, being absent for C3, however. Nonetheless, the results seem plausible overall.

I also measured wake-up latencies when waking a core by writing to its `monitor`ed memory region. However, as I already mentioned in Section 3.3, the `mwait` instruction on the Ryzen 5 5600G only works with C1, not C2 or C3. The alternative C-state entry mechanism of reading designated I/O ports does not allow monitoring memory. As such, cores in C2 or C3 can not be woken by write. Instead, the framework continues using Inter-Processor Interrupts (IPIs) for waking cores from these C-states and measures the

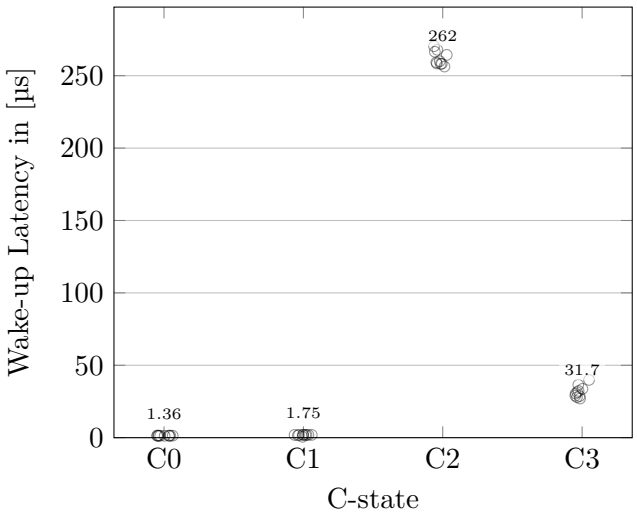


Figure 5.11: Wake-up latencies of different C-states supported by the Ryzen 5 5600G when woken by write to `monitor`ed memory (C1) and IPI (C2, C3).

time between the leader sending the IPI and the other core waking up. These results are not as comparable to C1, though. The observed wake-up latencies are visualized in Figure 5.11. For C0, where no actual reactivation of the core needs to occur, the measured value describes the time between the leader core rewriting the condition variable of the POLL workload and the other cores leaving POLL due to reading this changed state. Observed values are very stable at $\approx 1.36\,\mu s$, which is much lower than the latency of $\approx 9.06\,\mu s$ that I observed in the same measurements for the leader core being woken by the HPET interrupt. This seems plausible, since no interrupt routing or handling takes place. I would have expected it to be even faster if anything, as these latencies basically describe the time for a write to a memory address on one core becoming visible to other cores, which in my mind should happen near instantaneously. However, cache coherency protocols are known to be non-trivial, so it might make sense that this latency comes from them. Also, I could observe this latency for all 11 non-leader cores, adding to its credibility. For C1 most values are very stable at around $\approx 1.92\,\mu s$, with two outliers at $\approx 0.611\,\mu s$ and $\approx 1.58\,\mu s$, giving us an average of $\approx 1.75\,\mu s$. This, too, is significantly lower than when being woken by the HPET. As there is no interrupt handling, and the protocol waking a core on write is presumably faster than an interrupt from the chipset, this also seems plausible. For C2 and C3, however, something interesting happens. While wake-up latencies for C2 increase drastically to $\approx 262\,\mu s$, they drop to $\approx 31.7\,\mu s$ for C3. This is practically a reversal of when they were woken via the HPET previously, where wake-up latencies were $\approx 26.8\,\mu s$ and $\approx 311\,\mu s$, respectively. The drop for C3 is somewhat explainable. While AMD does not give nearly as many details about the implementation of its C-states as Intel, they might also support some Package C-states (PC-states). Since at the point of this measurement the leader core is already active, the latencies observed in this measurement do not include any PC-state wake-up latency. In contrast, for the previous measurement where the leader was woken by the HPET, any

additional PC-state wake-up latency would be included. Additionally, interrupt routing should be faster as well, as the route for a HPET interrupt originating in the chipset to the leader core is simply longer than the route for the IPI from the leader to the other cores. The reason for the dramatic increase in latencies for C2 is a mystery, however. It seems unlikely to simply be a bug, as an error in the code should affect C3 as well. Repeated measurements also showed similar results. While I do not have a satisfying explanation for this, it is very reminiscent of C3 on the Intel Core i7-4790, where similar "symmetry-breaking" behavior occurred. This suggests that there may be an underlying reason that even crosses manufacturer boundaries.

For the measurements done so far, the leader core sent its wake-up IPI for C2 and C3 as a Non-Maskable-Interrupt. However, as these are I/O-based C-states, a normal interrupt would suffice, too; see Section 4.3. Latency measurements would not include the overhead of interrupt handling in this case, as the framework disables it during measurements. To test this, I made a small modification to the main version of the framework[5], and repeated the measurements. I won't discuss the results in as much detail, as they remain mostly the same. The main difference is — as expected — in the wake-up latencies for C2 and C3. For these two C-states measured wake-up latencies are lower by $\approx 10.5\,\mu s$ and $\approx 12.0\,\mu s$, respectively. Thus, to measure wake-up latencies as accurately as possible, it seems advisable to minimize this interrupt handling overhead, as it does have a noticeable impact. Another small difference to point out is that, while the power values gathered from this measurement run behave very similarly overall, they decreased by roughly 9 % for C1 and C2 compared to the earlier measurement. There is no apparent reason in the code that fully explains this difference. This suggests that despite all current measures to control the environment, there are still factors which can moderately affect results when measurements are longer apart temporarily.

5.4 ARM

For this section, I measured an ARM device to proof the functionality of the framework on this architecture. The experiments ran on a Raspberry Pi 5, which contains a quad-core Cortex-A76 processor. At the time of the measurement it ran a kernel of Linux kernel version 6.6.20. As an external source of energy measurements I used an "ODROID Smart Power"[6]. This device advertises an error tolerance of 2 % and a sample rate of 10 samples per second. While I did not check the error tolerance for this thesis, I could not confirm the sample rate to actually be that high, with the actual sample rate mostly fluctuating between 3 and 5 samples per second. For these specific measurements this is fine and can be easily corrected for by making individual measurements longer, increasing the chance of one or multiple samples in the associated time range. If at any point the goal becomes greater than a simple proof of concept however, and more reliable data is needed, a better external power measuring device should be looked into.

[5] See `https://github.com/obibabobi/MWAITmeasurements/commit/af69540181ef64b1f1f9fb65ab38b1142cf452fc`

[6] For details see `https://www.hardkernel.com/shop/smart-power/`

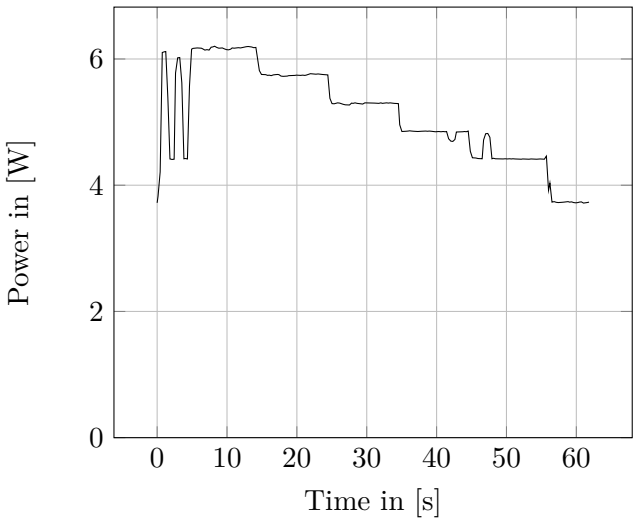


Figure 5.12: Power values over time during a measurement run on the Raspberry Pi 5.

As on x86, the framework pinned the processor to the highest available frequency using the "performance" frequency governor on all cores. Unlike previous measurements however, measuring externally does not allow singling out the processor energy consumption like RAPL does. Instead, the measured energy consumption will be that of the entire system. Because of this, further work needs to be done to control the power usage of other system components. For this, I pinned the case fan to the maximum speed as well. To deal with the unreliable sample rate of the external measurement device, I lengthened individual measurements from 100 milliseconds to 1 second. As the Cortex-A76 only allows requesting one sleep state, just measuring all cores being in the same state, i.e. executing POLL or `wfi`, would have resulted in only two data points. Instead, to get more data to discuss and substantiate the functionality of the framework, I took measurements gradually sending more and more cores into sleep, yielding five data points for the quad-core processor.

Figure 5.12 shows the power log produced by the measurement run. The framework measured each of the 5 states 10 times, with each individual measurement taking 1 second. Combined with the necessary time for the signal pattern in the beginning, measurements overall took nearly 1 minute in this case. Right at the start of the log, the measurement script on the Raspberry Pi 5 set the frequency governor to "performance". This is why the power log starts at a lower value than is reached at any point during the measurement, even when all cores execute `wfi`. The effect of pinning the case fan of the Raspberry Pi 5 to the highest speed is not visible on the other hand, as I set it manually at an earlier point. Clearly distinguishable in the beginning of the log are two "spikes" which the framework explicitly generated by alternatingly having all cores execute POLL and `wfi`. These spikes comprise the signal pattern, which the framework uses to associate power values with individual measurements in post-processing. After that, 5 "steps" can be distinguished, each lasting about 10 seconds and representing the measurements of one specific state, first executing POLL on all cores and then one by one sending more cores

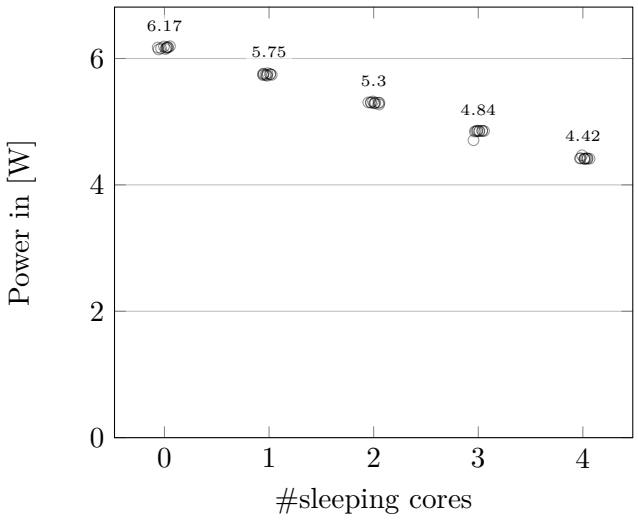


Figure 5.13: System energy consumption associated with different fractions of available cores sleeping on a Raspberry Pi 5.

to sleep. During the measurements of each of these states the power seems very stable, apart from 2 exceptions, a small dip towards the end of the fourth step and a spike at the beginning of the fifth. For the first, the reason is not known. As we are measuring a complete system, possible causes are manifold. For the second exception, due to the height of the spike it seems likely that one of the cores simply did not sleep as expected. The data collected by the framework supports this, stating that the third measurement of this state had to be repeated once, which is done automatically if at least one core's wake-up count surpasses a certain threshold. This automatic repetition is also the reason why this spike does not show up as an outlier in Figure 5.13. At the end of the fifth step, the measurement framework relinquishes its control over the Raspberry Pi 5 and also resets the frequency governor to the one used before the measurements started. This immediately leads to a drop in energy consumption as presumably the frequency is lowered.

The extracted results of the power measurements on the Raspberry Pi 5 can be seen in Figure 5.13. The values are again very stable. When all cores are executing POLL, the energy consumption of the entire system is $\approx 6.17\,\mathrm{W}$. When letting one core sleep, energy consumption decreases to $\approx 5.75\,\mathrm{W}$ and for half the cores sleeping it is further reduced to $\approx 5.30\,\mathrm{W}$. Once three cores execute `wfi` it is $\approx 4.85\,\mathrm{W}$, excluding the outlier at $\approx 4.71\,\mathrm{W}$ which was already visible in the power log. Finally, for all cores sleeping, energy consumption reaches $\approx 4.42\,\mathrm{W}$. Overall, energy consumption decreases quite linearly with each increase in sleeping cores, each step lowering energy consumption by a relatively constant $\approx 0.44\,\mathrm{W}$. From this data alone it is unfortunately unclear if any core entered the "Core dynamic retention mode" of the Cortex-A76, as described in Section 2.6.1. Still, overall results seem plausible and support the claim that the support for ARM and external power measurements in the framework works quite well.

Apart from the power values, the framework also collected wake-up latencies during these measurements. Plotting them makes little sense however, as for POLL the latency is exactly 0 ns, while for `wfi` it is always 37 ns. While these minuscule, perfectly stable values might seem too good to be true at first, there are plausible explanations for this. For one, the setup is somewhat different, which is why comparing these values to x86 should only be done cautiously; see Section 4.1. In a nutshell, wake-up latencies on x86 contain some overhead from interrupt routing and handling. Compare this to ARM, where the wake-up interrupt is generated by the core-adjacent Generic Timer, minimizing the time needed for interrupt routing. Also, no interrupt handler is started. Instead, the interrupt simply causes `wfi` to exit, after which the framework immediately takes the timestamp ending the measured period, eliminating any interrupt handling overhead from the results. All this means that measured wake-up latencies on ARM can be much shorter and more stable. Combine this with a resolution of the System Counter (SC) of ≈ 18.5 ns, and it becomes more understandable that the measured latency is always the same at a multiple of this value. As for the latency of 0 ns for the active state, the reason can be found in the POLL workload. As mentioned in Section 4.4, on ARM, the POLL workload repeatedly reads the SC and leaves its loop once the SC equals or exceeds the deadline where the measurement ends. The wake-up latency is then calculated as the difference between the deadline and the SC. This means that, as long as POLL's condition is checked at least once every ≈ 18.5 ns, the result would always be 0 ns, as the SC could never exceed the deadline. Wake-up latencies obtained by the POLL workload are not as useful on ARM, as they are not directly comparable to the other latencies. To conclude the evaluation of these latency measurements, the main takeaway should be that waking up from `wfi` seems to be near instantaneous. Direct comparisons to the previously measured values for x86 are difficult, however.

5.5 Unmodified Kernel on x86

The last measurements I did were to test out the modified version of the framework[7] that I created to overcome the need for a custom kernel on x86. The changes made to the framework were described in Section 3.7. For this section, I remeasured the two Intel systems, which contain an Intel Core i7-4790 and an Intel Core i7-6700K, respectively. I could not remeasure the Ryzen 5 5600G, however, as the modified version of the framework does not work on AMD; see Section 4.2. To keep measurements as comparable as possible, I used Linux kernel version 6.2.9 for these measurements as well, only with the custom kernel code removed. I also reused the same kernel configuration.

When it comes to the results, the power values are very similar to the ones observed in the previous measurements in Section 5.2. Interestingly, even the measurements for the POLL workload are nearly the same, even though it changed; see Section 4.4. The only result that did change in a noticeable way was for C1E on the Intel Core i7-6700K, where results dropped from the previously recorded ≈ 1.91 W to ≈ 1.62 W, which interestingly

[7] The associated code can be found under `https://github.com/obibabobi/MWAITmeasurements/tree/MasterarbeitNoCustomKernel`

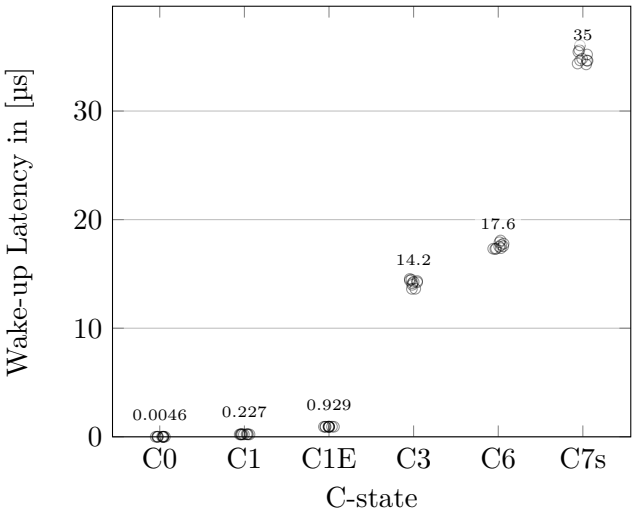


Figure 5.14: Wake-up latencies of different C-states supported by the Intel Core i7-4790 measured after abolishing the custom kernel.

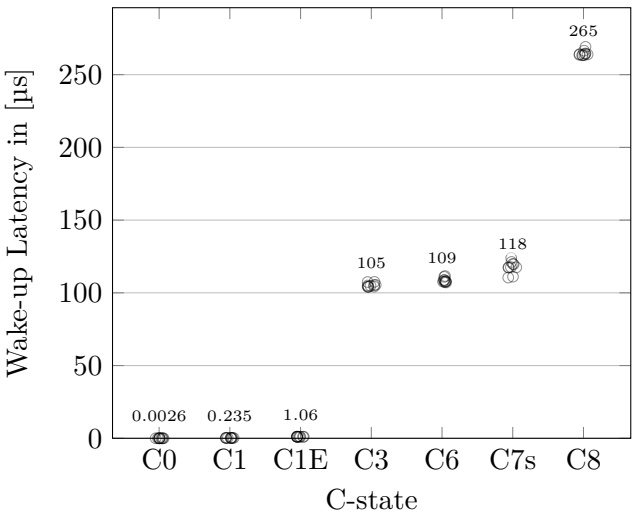


Figure 5.15: Wake-up latencies of different C-states supported by the Intel Core i7-6700K measured after abolishing the custom kernel.

brings it more in line again with the energy consumption observed in “Sleep Well” [20]. The reason for this is unclear.

The more interesting results can be found when comparing the wake-up latencies obtained through this new technique to those observed previously. Instead of using an NMI generated by the HPET like before, the interrupt sent by the Local APIC (LAPIC) Timer is a normal interrupt, for which handling is disabled during the measurement. Because of this, the results do not contain the overhead anymore that comes from the handling of the NMI in the Linux kernel, making them more accurate. Additionally, the route for the interrupt from the Local APIC Timer is shorter than for an interrupt generated by the HPET, which should also reduce the overhead incurred from interrupt routing. I already discussed this difference in overhead in Section 4.1. The latencies observed through this new method can be found in Figures 5.14 and 5.15 for the Intel Core i7-4790 and Intel Core i7-6700K processors, respectively. When comparing to the previous results regarding the Intel Core i7-4790 in Figure 5.5, the first striking difference is that the outliers that occurred with the previous measuring method are gone, with the new results being very stable. Apart from this, latencies are also lower as expected. The specific results as well as what was measured previously for the same C-states can be found in Table 5.2. For the Intel Core i7-6700K, measured wake-up latencies are again very stable, while also being lower overall. The measured latencies as well as their previously measured counterparts can be found in Table 5.3.

C-state	**HPET NMI**	**Local APIC Timer interrupt**	**+/−**
C0	≈ 3.00 µs	≈ 0.0046 µs	−3.00 µs
C1	≈ 3.16 µs	≈ 0.227 µs	−2.93 µs
C1E	≈ 6.43 µs	≈ 0.929 µs	−5.50 µs
C3	≈ 24.9 µs	≈ 14.2 µs	−10.7 µs
C6	≈ 30.0 µs	≈ 17.6 µs	−12.4 µs
C7s	≈ 61.3 µs	≈ 35.0 µs	−26.3 µs

Table 5.2: Wake-up latencies on the Intel Core i7-4790 when measured by different versions of the framework (using HPET NMI or Local APIC Timer interrupt to wake up leader core).

There are some interesting takeaways from these results. Firstly, the wake-up latencies observed for C1 are very low now, which suggests that there is little unnecessary overhead still contained in the results. Latencies for C0 are also very low, but similarly to ARM they are not directly comparable to the other latencies, as they are obtained through the POLL workload of the version without modified kernel on x86; see Section 4.4. Secondly, for the Intel Core i7-6700K, the overhead incurred by interrupt routing and handling is more or less constant across C-states at ≈ 11 µs. This is good to know, as it was just assumed previously in “Sleep Well” [20], but could now be shown experimentally for this CPU. For the Intel Core i7-4790, this assumption does not hold, however. Here, the overhead seems to rise from ≈ 3 µs to ≈ 26 µs over the course of the measurements.

All in all, these measurements show that it makes sense to move from an NMI generated by the HPET to an interrupt generated by the Local APIC Timer as the wake-up source.

C-state	HPET NMI	Local APIC Timer interrupt	+/−
C0	≈ 11.1 µs	≈ 0.0026 µs	−11.1 µs
C1	≈ 11.1 µs	≈ 0.235 µs	−10.9 µs
C1E	≈ 12.0 µs	≈ 1.06 µs	−11.0 µs
C3	≈ 117 µs	≈ 105 µs	−12.0 µs
C6	≈ 121 µs	≈ 109 µs	−12.3 µs
C7s	≈ 127 µs	≈ 118 µs	−9.00 µs
C8	≈ 276 µs	≈ 265 µs	−11.6 µs

Table 5.3: Wake-up latencies on the Intel Core i7-6700K when measured by different versions of the framework (using HPET NMI or Local APIC Timer interrupt to wake up leader core).

Not only did this change enable us to get rid of the custom kernel, but it also made measurements more accurate as a side effect. Enabling normal interrupts to trigger wake-ups on the leader core also did not noticeably increase the number of unwanted wake-ups as had been feared. The only downside is that the current implementation uses Intel-specific features, so AMD had to be excluded for the moment.

5.6 Comparison to Default Values

In Linux, the "cpuidle governor" decides which sleep state to enter when no actual work is to be done, basing its decision on some characteristics of these sleep states. The "cpuidle driver" provides these characteristics, by using hard-coded magic numbers in case of the "intel_idle" driver, or by extracting them from specific ACPI tables with the "acpi_idle" driver. Both of these cpuidle drivers, which are in use across the measured x86 systems, set a wake-up latency value to be used by the governor. They do not, however, set a power value as a second input.

So, while there are no meaningful default power values to compare our results to, it is still interesting to compare the default latency values to their measured counterparts. For this comparison I will use the most accurate results achieved in this thesis. Regarding the Intel processors, these are the values from Section 5.5, where the overhead from interrupt routing and handling was minimal. As that section does not provide new values for AMD, I will use the results from Section 5.3, instead. As for the Raspberry Pi 5 there was no working cpuidle driver, I will omit ARM from this section. Another important detail found in the documentation about the "intel_idle" driver is that, when entering a Core C-state that has a corresponding Package C-state (PC-state), the "intel_idle" driver always assumes that the PC-state might be entered. Because of this, the latency values refer to the PC-state in that case[8]. As ACPI documentation specifically talks about the "[...] worst-case latency [...]" [21, p. 482], it seems plausible that this holds true for the "acpi_idle" driver as well. This means that only comparisons to some measurements made here actually make sense, namely the ones where Package C-states

[8] This statement can be found at `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/admin-guide/pm/intel_idle.rst?h=v6.2.9#n251`

could be entered. This excludes any measurement were a core was woken by writing to its `monitor`ed memory region or by an Inter-Processor Interrupt, as these are actions that require an already active core, effectively disabling PC-states on the single-socket machines used here. Also, while ACPI documentation clearly states that the unit of its latency values is µs [21, p. 482], the "intel_idle" driver never seems to specify this. As µs seem like the most sensible unit, I will assume this here, too. With that out of the way, the comparisons for the two Intel systems can be found in Tables 5.4 and 5.5 for the Intel Core i7-4790 and Intel Core i7-6700K, respectively. For the Ryzen 5 5600G, the comparison can be found in Table 5.6, but keep in mind that the measured values in this table still contain significantly more overhead from interrupt handling and routing than those for the Intel systems.

C-state	**Default Latency**	**Measured Latency**
C0	0 µs	≈ 0.0046 µs
C1	2 µs	≈ 0.227 µs
C1E	10 µs	≈ 0.929 µs
C3	33 µs	≈ 14.2 µs
C6	133 µs	≈ 17.6 µs
C7s	166 µs	≈ 35.0 µs

Table 5.4: Default wake-up latencies on the Intel Core i7-4790 compared to latencies measured by the framework.

C-state	**Default Latency**	**Measured Latency**
C0	0 µs	≈ 0.0026 µs
C1	2 µs	≈ 0.235 µs
C1E	10 µs	≈ 1.06 µs
C3	70 µs	≈ 105 µs
C6	85 µs	≈ 109 µs
C7s	124 µs	≈ 118 µs
C8	200 µs	≈ 265 µs

Table 5.5: Default wake-up latencies on the Intel Core i7-6700K compared to latencies measured by the framework.

C-state	**Default Latency**	**Measured Latency**
C0	0 µs	≈ 9.06 µs
C1	1 µs	≈ 9.46 µs
C2	18 µs	≈ 26.8 µs
C3	350 µs	≈ 311 µs

Table 5.6: Default wake-up latencies on the Ryzen 5 5600G compared to latencies measured by the framework.

The main takeaway from these comparisons is that the default wake-up latency values are probably mostly good enough for the cpuidle governor to base its decisions on. Especially for the Intel Core i7-6700K and the Ryzen 5 5600G, while there is a lot of error when compared to the measured values, the default values still more or less accurately portray the relation between C-states and the rough ballpark of the latencies, which should be enough to make a reasonable selection. Only for the Intel Core i7-4790 the default latencies are more off the mark, especially for C6 and C7s. The cause for this could be that no Package C-states were entered as the Intel MSRs report, even though I already questioned the plausibility of these statistics in Section 5.2.3. If that is the case, and the Intel Core i7-4790 can not enter PC-states in general, the measurements could replace the worst-case, PC-state latency with the Core C-state latency, drastically improving its accuracy. In any case, replacing the default values with the measured ones would definitely be an improvement for all these processors, but especially for the Intel Core i7-4790, where the actual, measured latency for the deepest C-states is significantly lower than the default one. Using the measured values would allow the governor to select those deep C-states more often, saving additional energy.

6 Conclusion And Outlook

In this thesis, I extended an existing framework for measuring the energy consumption of sleep states. Most importantly, I enabled it to measure wake-up latencies, too, and added support for more processors. Where before only Intel CPUs had been measured, the framework is now capable of working with another manufacturer with AMD and even an entirely different architecture with ARM. To overcome the lack of internal energy measuring features on ARM, I also added support for using external measuring devices instead.

Considering the various measurements I successfully conducted, I argue that the extended framework works well. I was able to measure energy consumption and wake-up latencies for a wide variety of systems that used x86 CPUs from Intel and AMD as well as an ARM processor. External measurements on ARM did prove to work well, too. Nearly all results match my expectations, or can be reasonably explained. The overwhelming majority of measured values is also very stable, and consistent with one another.

Future Work

Nevertheless, there are some caveats. These include measured values changing when repeating measurements after some time. I also could not avoid all outliers, and some measurements showed a clear ramp-up effect. All this suggests that there are still some variables in the environment that the framework does not perfectly control. Nonetheless, these issues were moderate at most. They were never severe enough to hinder me from usefully analyzing the results. Still, future work on this framework may find ways to control the measured system even more fully, ruling out these undesirable effects.

There are other, more fundamental aspects of the framework that also might be worth revisiting in the future. One of these is the POLL workload, which the framework employed whenever measurements included active cores. Section 4.4 defined it in detail. I designed the POLL workload mostly for simplicity and to create a useful and generic baseline for measurements. However, as it makes most sense to use the framework to supply data to a cpuidle governor, measuring idle states actually used on Linux would be optimal. These real-world idle states include one that leaves the core active and is meant for very short idle periods. Effectively, it is a while loop that occasionally checks for new, more useful tasks[1]. It also uses architecture-specific features to make its idling more efficient, like the `pause` instruction on x86. Implementing this idle state in the measurement framework and measuring its energy consumption would be preferable,

[1] See `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/cpuidle/poll_state.c?h=v6.2.9#n13`

being more relevant than the custom POLL workload. Unfortunately, this is not trivial, as it involves interactions with other parts of the Linux kernel — especially the scheduler — which are difficult to emulate inside the framework. This is why I ultimately opted for defining a simpler workload for this thesis. In the paper "Energy Efficiency Aspects of the AMD Zen 2 Architecture" [18], Schöne et al. did also not measure this exact idle state [18, p. 568], presumably for similar reasons. For future research, it might be interesting to tackle this challenge and implement a workload closer to the actual idle state.

The task description outlines further possible research in the secondary goals that I did not, or only partially, fulfill. Firstly, while I added the capability to measure wake-up latencies when writing to `monitor`ed memory, I never did the same for Inter-Processor Interrupts. Additionally, all measurements for triggering `monitor` were conducted on single-socket machines. This meant that Package C-state could not be entered, making the gathered results less comprehensive. In the future, these measurements should be repeated on more suitable hardware. When it comes to running on an unmodified Linux kernel, the framework works well for the Intel and ARM systems I tested. However, AMD did prove to be more difficult to support in the same way, which is why the AMD processor used for testing — a Ryzen 5 5600G — is still not capable of running the framework on an unmodified kernel. This, too, might need revisiting. As for the last two secondary goals — supporting "Short-Time RAPL" and measuring a heterogeneous CPU — I did not tackle them in this thesis. The reason was simply that time constraints did not allow putting significant effort into all secondary goals, so I focused on those which seemed most interesting to me. Nonetheless, these last two goals are still appealing and should be explored at some point.

Apart from further extending and improving the framework, there is also the question of how to use the gathered values. As the introduction to this thesis pointed out, the natural application of these accurate sleep state characteristics would be within a cpuidle governor. Further research could plug the measured values from the framework into such a governor, and evaluate the impact on energy consumption and performance. A roadblock, however, would be that common governors do not use the power information directly. Instead, they base their decision on a value called "target residency". As entering and exiting a sleep state costs energy too, this value describes the time that needs to be spent in the sleep state to break even on this energy cost[2]. Currently, the framework is not geared towards measuring this value, focussing instead on the energy consumption while already in a sleep state. A custom governor might thus be necessary to use it directly in conjunction with the framework. Alternatively, future work could use measured values by the framework to calculate the target residency, too.

[2] See `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/cpuidle/governors/menu.c?h=v6.2.9#n41`

Bibliography

[1] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual, Volume 2: System Programming*. Rev. 3.42. Mar. 2024. URL: https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf (visited on Apr. 30, 2024).

[2] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual, Volume 3: General-Purpose and System Instructions*. Rev. 3.36. Mar. 2024. URL: https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24594.pdf (visited on May 19, 2024).

[3] Advanced Micro Devices, Inc. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 16h Models 30h-3Fh Processors*. Rev 3.06. Mar. 18, 2016. URL: https://www.amd.com/content/dam/amd/en/documents/archived-tech-docs/programmer-references/52740_16h_Models_30h-3Fh_BKDG.pdf (visited on Feb. 1, 2024).

[4] Arm Limited. *Arm Architecture Reference Manual for A-profile architecture*. Version K.a. Mar. 20, 2024. URL: https://developer.arm.com/documentation/ddi0487/latest/ (visited on Apr. 24, 2024).

[5] Arm Limited. *Arm® Cortex®-A76 Core Technical Reference Manual*. Version 0401-01. June 30, 2023. URL: https://developer.arm.com/documentation/100798/latest/ (visited on Feb. 14, 2024).

[6] Arm Limited. *Learn the architecture - Generic Timer*. 0102-01. Aug. 24, 2023. URL: https://developer.arm.com/documentation/102379/0102 (visited on Feb. 29, 2024).

[7] Shuhaizar Daud, R. Badlishah Ahmad, Ong Bi Lynn, Zahereel Ishwar Abd Kareem, Latifah Munirah Kamarudin, Phaklen Ehkan, Mohd. Nazri Mohd. Warip, and Rozmie Razif Othman. "The effects of CPU load & idle state on embedded processor energy usage." In: *2014 2nd International Conference on Electronic Design (ICED)*. Aug. 2014, pp. 30–35. DOI: 10.1109/ICED.2014.7015766. URL: https://ieeexplore.ieee.org/document/7015766 (visited on Mar. 2, 2024).

[8] Daniel Hackenberg, Roland Oldenburg, Daniel Molka, and Robert Schöne. "Introducing FIRESTARTER: A processor stress test utility." In: *2013 International Green Computing Conference Proceedings*. June 2013, pp. 1–9. DOI: 10.1109/IGCC.2013.6604507. URL: https://ieeexplore.ieee.org/document/6604507 (visited on May 7, 2024).

[9] Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. "An Energy Efficiency Feature Survey of the Intel Haswell Processor." In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. May 2015, pp. 896–904. DOI: 10.1109/IPDPSW.2015.70. URL: https://ieeexplore.ieee.org/document/7284406 (visited on Feb. 8, 2024).

[10] Thomas Ilsche, Robert Schöne, Philipp Joram, Mario Bielert, and Andreas Gocht. "System Monitoring with lo2s: Power and Runtime Impact of C-State Transitions." In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2018, pp. 712–715. DOI: 10.1109/IPDPSW.2018.00114. URL: https://ieeexplore.ieee.org/document/8425481 (visited on Mar. 1, 2024).

[11] Intel Corporation. *6th Generation Intel® Processor Families for S-Platforms*. Rev. 010. Vol. 1. Feb. 2022. URL: https://www.intel.com/content/www/us/en/content-details/332687/6th-generation-intel-core-processor-family-datasheet-volume-1.html (visited on Feb. 7, 2024).

[12] Intel Corporation. *IA-PC HPET (High Precision Event Timers) Specification*. 1.0a. Oct. 2004. URL: https://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/software-developers-hpet-spec-1-0a.pdf (visited on Feb. 29, 2024).

[13] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B, 2C, & 2D): Instruction Set Reference, A-Z*. Dec. 2023. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html (visited on Feb. 5, 2024).

[14] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B, 3C, & 3D): System Programming Guide*. Dec. 2023. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html (visited on Feb. 8, 2024).

[15] Intel Corporation. *Making of a Chip*. Jan. 2012. URL: https://download.intel.com/newsroom/kits/chipmaking/pdfs/Sand-to-Silicon_22nm-Version.pdf (visited on May 8, 2024).

[16] Abdelhafid Mazouz, Alexandre Laurent, Benoît Pradelle, and William Jalby. "Evaluation of CPU frequency transition latency." en. In: *Computer Science - Research and Development* 29.3 (Aug. 2014), pp. 187–195. ISSN: 1865-2042. DOI: 10.1007/s00450-013-0240-x. URL: https://doi.org/10.1007/s00450-013-0240-x (visited on Mar. 2, 2024).

[17] Robert Schöne, Thomas Ilsche, Mario Bielert, Andreas Gocht, and Daniel Hackenberg. "Energy Efficiency Features of the Intel Skylake-SP Processor and Their Impact on Performance." In: *2019 International Conference on High Performance Computing & Simulation (HPCS)*. July 2019, pp. 399–406. DOI: 10.1109/HPCS48598.2019.9188239. URL: https://ieeexplore.ieee.org/document/9188239 (visited on Mar. 1, 2024).

[18] Robert Schöne, Thomas Ilsche, Mario Bielert, Markus Velten, Markus Schmidl, and Daniel Hackenberg. "Energy Efficiency Aspects of the AMD Zen 2 Architecture." In: *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. ISSN: 2168-9253. Sept. 2021, pp. 562–571. DOI: 10.1109/Cluster48925.2021.00087. URL: https://ieeexplore.ieee.org/abstract/document/9556102 (visited on Feb. 9, 2024).

[19] Robert Schöne, Daniel Molka, and Michael Werner. "Wake-up latencies for processor idle states on current x86 processors." en. In: *Computer Science - Research and Development* 30.2 (May 2015), pp. 219–227. ISSN: 1865-2042. DOI: 10.1007/s00450-014-0270-z. URL: https://doi.org/10.1007/s00450-014-0270-z (visited on Mar. 1, 2024).

[20] Till Smejkal, Jan Bierbaum, Thomas Oberhauser, Horst Schirmeier, and Hermann Härtig. "Sleep Well: Pragmatic Analysis of the Idle States of Intel Processors." In: *Proceedings of the IEEE/ACM 8th International Conference on Big Data Computing, Applications and Technologies*. IEEE/ACM 8th International Conference on Big Data Computing, Applications and Technologies. BDCAT '23. Taormina (Messina), Italy: ACM, Dec. 2023. ISBN: 979-8-4007-0473-4. DOI: 10.1145/3632366.3632385.

[21] UEFI Forum, Inc. *Advanced Configuration and Power Interface (ACPI) Specification*. Release 6.5. Aug. 29, 2022. URL: https://uefi.org/sites/default/files/resources/ACPI_Spec_6_5_Aug29.pdf (visited on Feb. 6, 2024).