Quantifying the Power Consumption of Processes on Linux using Intel RAPL

Valerie Luna Dickson, Péter Tamás Sebők

Computer Science, cs-23-pt-10-02, 2023-06

Master's Project

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Selma Lagerløfs Vej 300
DK-9220 Aalborg Ø

Title:
Quantifying the Power Consumption of Processes on Linux using Intel RAPL

Theme:
Energy Aware Programming

Project Period:
February 2023 - June 2023

Project Group:
cs-23-pt-10-02

Participant(s):
Valerie Luna Dickson
Péter Tamás Sebők

Supervisor(s):
Bent Thomsen
Thomas Bøgholm

Copies: 1

Page Numbers: 67

Date of Completion:
June 9, 2023

Abstract:

The aim of this project is to produce a set of tools that can help software developers and researchers determine the effects of background processes on energy consumption. We created a system, comprised of three components, that can be used to estimate the power used by individual processes. The first component is a profiler that uses a set of benchmarks to calculate the energy consumption characteristic of a given CPU. The second is a process tracker that records the CPU usage of running programs. Finally, a number of scripts are used to analyze the data provided by the first two components. While results generated from the process tracker tests are inconclusive, results from the profiler have provided an interesting insight into how energy costs differ between programs.

Summary

As energy efficiency of software is becoming increasingly important, the number of experiments analyzing the power consumption of the different components of software (such as programming languages, frameworks, etc) is also on the rise. Some of these experiments suffer from a lack of precision, reliability, and reproducibility due to the fact that current power monitoring solutions, such as Intel RAPL, provide only high-level information (e.g. package power consumption). This makes it impossible to determine the effect of background tasks and processes on the power consumption of the actual software under test. The current best practice is to try to minimize the effect of such processes by disabling unnecessary programs, automatic updates, etc. Unfortunately, as modern operating systems are very sophisticated, this solution is neither reliable nor reproducible.

Additionally, the complexity of modern CPUs also introduces difficulties when trying to measure the energy usage of individual programs. One of the factors that must be considered is hyperthreading, supported by virtually all modern CPUs. Hyperthreading enables a CPU to divide its physical cores into multiple logical ones. From the point of the operating system and the applications, each logical core behaves as if they were physical cores.

The goal of this project is to develop tools that can help researchers and software developers ascertain the effect of background processes on power consumption. To do so, we must take into account the CPU usage of running processes, as well as hyperthreading. When programs are run over only physical cores, the power consumption increases in a reasonably linear fashion with each active physical core. However, when hyperthreading is enabled, the increase in power consumption will not be linear, due to the fact that hyperthreads running on the same physical core share hardware components. The amount of power used by two hyperthreads (running on the same physical core) is thus the amount of power used by a single hyperthread times some multiplier. We assume that this multiplier is unique to each CPU model, or even to each individual CPU, due to variations in manufacturing.

To help us compute the power usage of individual processes, we have created a system composed of three components: a *profiler* that can run a pre-defined set of benchmarks to collect the information used to calculate the CPU's power consumption characteristics, a *process tracker*, that can collect data on the running processes, and a set of *scripts*, that are used to analyze the results obtained by the other tools.

The *profiler* is a C++ application, able to run micro and macro benchmarks. It accepts configuration options, such as the actual benchmarks to run, number of cores to use, and whether or not hyperthreading should be used. It then measures the amount of time taken, and power

consumed by each benchmark. This data is used to determine the power consumption characteristic of the given CPU. This characteristic is comprised of the static power (the theoretical power consumed when no cores are active), and the hyperthreading ratio mentioned above. The *process tracker* is a Rust program, that tracks the CPU usage of currently running processes using Linux's /proc pseudo file system.

Finally, the *scripts* are responsible for analyzing the results produced by the *profiler* and *process tracker*. They can calculate the CPU's static power consumption, and the hyperthread energy usage ratio, as well as the CPU utilization, process CPU usage, and process power consumption using the results from the *process tracker*. Three methods are used to calculate these results. The first naively calculates the power as a proportion of CPU time. The second takes into account hyperthreading over individual cores, while the third includes an algorithm derived from the HaPPy paper[42].

There are also plans for additional improvements that we could not implement in time. Such improvements are an UEFI mode for the *profiler*, and a modification of the Linux scheduler. Running the *profiler* in UEFI mode would allow us to measure the power consumption of each benchmark without the effects of any background processes. Modifying the Linux scheduler to record the amount of time a given process was scheduled for, and on which core it was running should, in theory allow for more accurate energy usage estimations. We performed several tests using our tools. Results using the *process tracker* were inconclusive, demonstrating large differences between what should be similar environments. However, results from the *profiler* provided insight into different energy usages between programs, shining light on how complex the problem of measuring the energy usage of a single program is. Further research is required to investigate these results.

Acknowledgements

We would like to thank our supervisors, Bent Thomsen and Thomas Bøgholm for the continuous input, feedback, and advice they have provided throughout our last two semesters.

Contents

Glossary

jiffies A unit of measurement for the Linux kernel, one 'jiffy' is the time Linux takes for a single clock tick. For backwards compatibility reasons, in userspace one jiffy is always 1/100th of a second..

RAPL Software-based power consumption measurement tool on newer Intel CPUs.

Acronyms

CLBG Computer Language Benchmark Game.

ICT Information and Communication Technology.

MSR Model Specific Register.

TDP Thermal Design Power.

Chapter 1

Introduction

The power usage and energy efficiency of Information and Communication Technology (ICT) systems has become a major point of consideration in recent years. One of the primary reasons for this is cost: reducing the power consumption of ICT devices may lead do considerable savings, especially in large data centers. However, perhaps an even more important factor is the environmental impact of such devices. Some estimates state that, in the worst-case, 20% of the global electricity demand may be used by ICT infrastructure by 2030 [18].
Improvements to energy efficiency have been primarily driven by advances in chip (and other hardware) design. For example, an *Intel Pentium II Xeon 400 MHz* ([14]) server processor, released in 1998 has a Thermal Design Power (TDP) of 30.8 watts. It has a single core, running at 400 MHz. Contrasting it with a *Intel Xeon W-1390T* ([17]), released in 2021, we can see the increase in efficiency: the W-1390T has 8 physical (16 logical) cores, running at 1.5 GHz, with a TDP of 35 watts.
However, there is another aspect to consider when we attempt to lower to power consumption of any ICT system: the energy efficiency of software. As power consumption becomes an increasingly important question, developers have begun to consider the effects of software. To help with this, research has been conducted to try do determine the energy efficiency of different programming languages ([2, 25]), frameworks ([4]), coding practices ([22]), and so on.
An important consideration when conducting such an experiment is the choice of power measuring instrument. There are two categories: external power meters, (devices such as the Watts Up Pro [39]) that measure the power drawn from a socket by the entire system, and internal power meters that rely on sensors and performance counters. The most prominent example of the second category is Intel RAPL.
RAPL has several advantages over external power meters: it requires no extra investment, as it is available on all relatively recent Intel CPUs, it is easy to read, and it provides accurate and frequent measurements. However, it only provides socket-level power readings, making it impossible to determine the power consumption of individual CPU cores, threads, or even processes. This limitation presents an issue: the goal of many of the power consumption experiments is to evaluate the efficiency of some language, framework, etc, in a realistic setting. This means using a device such as a workstation, or server, and running the tests under an

operating system. Due to the complexity of modern operating systems, it is impossible to predict and control the appearance of background processes. Such processes may influence the results of the experiments, as they themselves use some CPU time, and thus, power.
In this project, our goal is to develop tools that can help researchers quantify the power usage of such processes, using Intel RAPL.
The structure of this report is as follows: Chapter 1 presents the introduction, problem statement, research questions, and the expected contributions. Chapter 2 lists a number of related works. Chapter 3 gives a short overview of Intel RAPL, as well as the methods we used to retrieve in different parts of our system. Chapter 4 introduces our the different components of our tool. Chapter 5 shows the results we achieved by using our tools. Chapter 6 details some of the challenges encountered during development, including unimplemented functionality. Finally, Chapter 7 presents some possible future works, as well as the overall conclusion.

1.1 Problem Statement

The field of computer energy use is gaining more and more relevance in an increasingly green-focused world. Much research has been done to try and identify the impact of code on energy use, but experiments often plagued by the issue of background tasks influencing results. Most modern systems are not single-purpose systems, and run many tasks simultaneously, which may use computer resources at arbitrary times. Today's methods of measuring power use cannot differentiate between different processes, so results may be skewed by code that is not the code under test. Not much research has been done into identifying or isolating background tasks in energy measurement. The current best practice is to try and minimize background tasks where possible, but the actual impact of these processes, or the mitigation, is unknown. The aim of this research is to develop different measurement techniques and tools to shed some light on the impact of background tasks. Ideally, it will provide tools that can accurately measure individual process power consumption. We aim to accomplish this through both practical and statistical methods with preexisting power consumption measurement tools.

1.2 Research Questions

Our goal in this project is to create a set of tools that can help developers and researchers determine the power consumption of background processes under Linux, and to evaluate the accuracy of said tool. This leads to our research question:

RQ1 Is it possible to measure power consumption of individual processes on modern hardware and systems?

RQ1A If so, what level of accuracy can we accomplish?

1.3 Expected Contributions

Our goal is to produce a number of tools that can help researchers and software developers quantify the power consumption of a specific process in an environment with background processes.

1. A profiling tool, which runs a set of benchmarks to create an energy profile of the system. This profile is used to calculate two sources of energy consumption:
   (a) **Static energy consumption**, which is the baseline level of power the CPU uses just by being online.
   (b) **Dynamic energy consumption**, which is the power used by the CPU in an active state, running user code.
2. A process monitoring tool, that monitors running processes and collects statistics such as CPU cycle counts and memory usage.
3. A set of scripts to aid in the calculation of the total power consumption of individual processes, using the data generated by the profiling tool and the process monitoring tool.

Additionally, we would like to evaluate the efficiency and accuracy of these tools, to determine their usefulness compared to existing solutions.

Chapter 2

Related Works

2.1 RAPL

Intel RAPL (Rolling Average Power Limit) is a hardware feature present in recent Intel CPUs, that allows us to programmatically monitor the power consumption of the CPU, and in some cases the DRAM and the integrated GPU. Khan et al. in “RAPL in Action” [19] investigate the strengths and weaknesses of the RAPL interface, using methods such as microbenchmarks, application-level benchmarks, and additional datasets from *Taito*, a cluster of the Finnish Center of Scientific Computing. They find that “RAPL readings are highly correlated with plug power, promisingly accurate enough, and have negligible performance overhead.”. These findings indicate that RAPL can be used in power consumption experiments. They do note some problems, the most significant ones being the unpredictable timing of the register updates, and possible register overflows.

2.2 Scaphandre

Scaphandre [10] is a tool designed to measure power consumption of individual processes on a system. It’s designed for use on a bare metal host, but has the ability to measure the power consumption of KVM virtual machines. It is intended to be used in industry with conventional metric monitoring solutions, but of more interest to us is their method of devising individual process power metrics.

Scaphandre uses Intel RAPL (3) on both Windows and Linux to gain a system-wide power consumption metric. To separate processes, Scaphandre assumes the power consumption of a process is directly proportional to the CPU time e.g a process that uses half of a CPU’s capacity used half the measured energy usage.

2.3 Intel Power Gadget

Intel Power Gadget [15] is a tool provided by Intel to monitor and log processor information such as frequency, temperature, and power usage. Officially, it is supported on Windows and macOS operating systems, however, there is an unofficial Linux version as well [12]. All versions use the RAPL Model Specific Registers (MSRs) to gather power usage information.

All versions are capable of outputting the measured values in a comma-separated file for further analysis. The Windows and macOS versions also provide a user interface that displays the power, temperature, etc. as a graph.

The advantage of Intel Power Gadget is that it is available for Windows, while most other energy measurement tools focus on Linux. Additionally, it offers the same granularity as other RAPL-based tools: package, core, uncore, and dram domains.

2.4 HaPPy

HaPPy: Hyperthread-aware Power Profiling Dynamically [42] is a paper that posits a new model for attributing power consumption to jobs, taking into account hyperthreading. They posit an initial, hyperthreading-unware model, where additional core usage linearly translates to additional power usage, with a static power usage added on top. They show how this model can map a variety of workloads without hyperthreading, but fails when hyperthreading is enabled, due to the sharing of various execution components in a single core. They then present an extended, hyperthreading-aware power model, which takes into account which core each thread is running on, modelling power consumption with a greater degree of accuracy than the hyperthreading-unaware model.

2.5 Energy Consumption Benchmarks

Benchmarking is the act of measuring some property of software. In this project, we are interested in energy consumption. While execution time can sometimes used as a proxy for power usage, as shown in by Pereira et al. in [26], that is not always the case.
Microbenchmarks are small, well defined programs that usually only do one thing (e.g. some heavy computation). The Computer Language Benchmark Game (CLBG) [40] is a collection of such benchmarks. They are used extensively in energy efficiency research, for example, in papers [26] and [21], that investigate the energy consumption of different programming languages and programs.
While microbenchmarks are useful, as they can be implemented easily, they are not representative of real-life workloads. For such tests, macrobenchmarks can be used instead.
OpenBenchmarking.org [3] provides a repository of such benchmarks. Alternatively, it is possible to use complete programs (such as ones found in the Software-artifact Infrastructure Repository [33]) as benchmarks. They are used (among other sources) in [31], a paper that attempts to quantify the influence of software features on energy consumption. Finally, SPEC

([34]) provides standardized benchmarks for performance evaluation, including power characteristics ([20]).

Chapter 3

Intel RAPL

RAPL is a set of MSRs, [13] suitable for tracking energy consumption, introduced in the Sandy Bridge line of processors. The RAPL interfaces can be used to retrieve the energy consumed by the following domains:

- **Package (PKG)**: energy consumption of the socket.
- **Power Plane 0 (PP0)**: energy consumption of all processor cores on the socket.
- **Power Plane 1 (PP1)**: energy consumption of the integrated GPU.
- **DRAM**: energy consumption of the RAM attached to the CPU.

Not all domains are available on all devices. **PP1** is only accessible on desktop CPUs, while **DRAM** is only accessible on server devices. **PKG** and **PP0** is available on both client and server CPUs.

The RAPL MSRs are 32-bit registers, and are updated roughly once every 1ms. This means that RAPL can be used to collect real-time (or near real-time) power consumption information. However, as found by [19], the RAPL registers can also easily overflow, which must be taken into account if they are accessed and read directly.

Different processor architectures might store the consumed power values in different units and increments. For example, the default value for Sandy Bridge processors is 15.3μJ. The values of the power and time units can be determined by reading the MSR_RAPL_POWER_UNIT register [13].

Additionally, the RAPL interface also provides functionality to limit the power usage of the different CPU domains. However, in this project we only use the reporting functionality of RAPL.

3.1 Reading RAPL

There are several ways to access the power readings provided by RAPL. This section lists the methods used in the project, and the advantages and disadvantages of the different approaches.

3.1.1 Reading MSRs directly

The values of the RAPL registers can be read by using the *RDMSR* instruction [13]. This is the most direct way of accessing the RAPL MSRs, however there are a number of disadvantages.
The main drawback is that the *RDMSR* instruction must be executed at privilege level 0 (kernel privilege), otherwise a general protection exception will be generated by the CPU. This means that user space programs cannot use the *RDMSR* instruction. Additionally, as discussed in Intel RAPL, the program using *RDMSR* must be aware that different architectures use different power and time units, and must be able to calculate these based on the values in the corresponding registers. Finally, the program must be ready to handle possible overflows in the RAPL register that stores the actual power consumption values.
Despite these disadvantages, direct MSR access is required if we attempt to retrieve RAPL readings in bare metal configuration.

Listing 3.1: Reading register values directly.

```
uint64_t RawMSRInterpret::getRegister(uint64_t offset)
{
    uint64_t ret = __LONG_LONG_MAX__;
    uint32_t *lo, *hi;
    lo = (uint32_t*)&ret;
    hi = lo + 1;
    asm volatile("rdmsr" : "=a"(*lo), "=d"(*hi) : "c"(offset));
    return ret;
}
```

3.1.2 Using the Powercap interface

It is possible for programs running in user space to access the RAPL readings by using the Powercap interface provided by the Linux kernel [29]. It is located under the *sysfs* pseudo file system: */sys/devices/virtual/powercap*. The interface exposes several objects that can be used to access settings related to power capping.
The exact contents of the pseudo file system might vary depending on the capabilities of the system, but it typically contains objects (folders) that represent the different power zones (or domains) of the system.

Figure 3.1 shows the partial result of running the *tree* command in the */sys/devices/virtual/powercap* directory. Here *intel-rapl:0* is the power zone for **package-0** (indicated by *intel-rapl:0/name*). *intel-rapl:0/intel-rapl:0:0* is a subzone, in this case the **core** domain. *energy_uj* is the energy counter in microjoules, and *max_energy_range_uj* is the range of the counter, also in microjoules.

```
powercap
├── intel-rapl
├── enabled
├── intel-rapl:0
│   ├── device -> ../../intel-rapl
│   ├── enabled
│   ├── energy_uj
│   ├── intel-rapl:0:0
│   │   ├── device -> ../../intel-rapl:0
│   │   ├── enabled
│   │   ├── energy_uj
│   │   ├── max_energy_range_uj
│   │   ├── name
│   │   ├── power
│   │   │   └── autosuspend_delay_ms
│   │   ├── subsystem -> ../../../../../../class/powercap
│   │   └── uevent
│   └── name
│
```

Figure 3.1: Powercap objects.

Listing 3.2: Reading Powercap.

```
uint64_t PowercapInterface::getCurrentJoules()
{
    ifstream file(this->path);
    string line;
    if (file.is_open() && getline(file, line))
    {
        file.close();
        uint64_t power = stoll(line);
        return power / 1000;
    }
    else
    {
        exit(-1);
    }
}
```

Listing 3.2 shows how we can read energy consumption values using the Powercap interface. Here, the `path` field of the `PowercapInterface` class contains the path to the file that corresponds to the power zone we would like to access (e.g. if we are interested the total power consumption of the system, its value will be set to "/sys/class/powercap/intel-rapl:0/energy_uj")

3.1.3 Using the MSR interface

Another possibility is to make use of the interface located under */dev/cpu/CPUNUM/msr* [24], which provides access to the MSRs of an x86 CPU. As it is an interface similar to Powercap, it can be accessed from user space. It provides raw values, same as reading the registers directly. This can be used to test code that is responsible for converting these raw values into some desired format without having to boot into bare metal mode.

Listing 3.3: Reading an arbitary MSR value.

```
uint64_t LinuxMSRInterpret::getRegister(uint64_t offset)
{
    // cannot use ifstream because of caching
    int file;
    uint64_t value;
    file = open("/dev/cpu/0/msr", O_RDONLY);
    if (!file) goto fail;
    if (pread(file, (void*)&value, 8, offset) != 8) goto fail;
    close(file);
    return value;
    fail: exit(-2);
}
```

Listing 3.3 shows how we read the value of an arbitrary MSR (defined by the `offset` parameter) using C++.

Chapter 4

Components

Our system is a design in three parts. First, a profile can be created by the **profiler**, which runs a series of benchmarks to calculate the static and runtime power consumption of the CPU. Second, the **process tracker** is run to gather data on the process usage, and a set of **scripts** to calculate the energy usage of individual processes from the profile and gathered process data.

4.1 Profiler

The profiler is conceptually simple, consisting of a set of benchmarks to stress the system in different ways, measurement functionality for these benchmarks, and post-processing to create the profile, a set of data used by the process tracker to estimate power consumption of a process, even with other processes running on the same system.

4.1.1 Testing Process

The user has the ability to specify options on launch through command line parameters. This includes which tests to run, how many iterations of tests to run, the filename of the output file, and which 'test specifications' to run. For our purposes, a test specification is a number of threads plus whether it is hyperthreading-enabled or hyperthreading-disabled. A hyperthreading enabled specification will run on the smallest number of physical cores possible, while a hyperthreading-disabled specification will not run more than one thread on a single physical core.

Additionally, it's possible to specify a confidence interval and a margin of error. These values are used to automatically determine the number of iterations. This is done by using the following equation (taken from [7]):

$$Z^* \cdot \sqrt{\frac{\hat{p} \cdot (1-\hat{p})}{n}} \leq marginOfError$$

In this equation Z^* is the critical value, based on the provided confidence interval. $\hat{p}$ is set to 0.5, as that is the worst-case scenario (0.5 maximizes the value of $\hat{p} \cdot (1 - \hat{p})$). Finally, n is the number of desired iterations. After rearranging for n, we get:

$$n >= \frac{Z^{*2}}{4 \cdot marginOfError^2}$$

At this point, the right-hand side of the equation can be calculated. The result is the rounded up to the nearest integer and will be set as the number of iterations for each benchmark.

It is also possible to manually set the number of iterations when the benchmark harness is created.

By default, the profiler runs with a set of default tests over every possible core specification.

The profiler will then run each test in series over each core specification. The order of tests and the order of core specifications are randomized, but a single test-spec will run all iterations at once. The tests will record the power consumption in memory. Once all the tests have completed, the results will be printed, and - if an output file is specified - written to a file in the json format.

The profiler uses Intel RAPL to measure power consumption, through either the Powercap interface, or direct MSR access, as specified at compile time.

4.1.2 Benchmarks

We have implemented a number of benchmarks that stress the CPU and memory usage of the target machine. We've included both microbenchmarks and macrobenchmarks, after finding that microbenchmarks tended to exhibit strange behavior when under load.

Sleep Test

The first 'benchmark' we implemented was the sleep test, a simple test that sleeps for a given number of seconds. This leads to zero processor usage, as the CPU is placed into a halt state. Our intention behind this test was twofold — first, it is a simple test to implement and verify correctness. Second, we originally thought a sleep test could be used to get the static power consumption of the CPU, given it would be online, but not executing any code. This was quickly dismissed as not viable. While every core in a CPU is asleep, much of the CPU hardware that takes up the static power consumption can be disabled. This leads to appreciable power savings, but leaves this unable to calculate the static power consumption. However, tests similar to this one were used to calculate the static idle power consumption, as detailed in 4.3.1.

Spin Loop

A spin loop was a second simple benchmark to implement. Given a length of time, it will spin endlessly on a time-checking function call, until the given length of time has passed. While this achieves the same goal as the sleep test, it does so in a far more inefficient manner, ensuring power is consumed. It was a second simple test to build while we were early on in development.

To ensure the empty loop is not optimized away by the compiler, we disable compiler optimizations. In addition, we have inspected the resulting machine code to verify that the loop is being performed.

Fannkuch-Redux

The Fannkuch-Redux test is an implementation of the Fannkuch-Redux algorithm [9], taken from the CLBG [8]. This test is intended to simulate CPU-heavy workloads that do not use a large amount of memory.

Binary Trees

Similarly to the Fannkuch-Redux benchmark, the Binary Trees test is an implementation from the CLBG. This test simply creates a binary tree of the specified depth. For each node in the tree, space for an array of doubles with a specified number of elements is also allocated on the heap. The purpose of this benchmark is to test the effects of memory-intensive applications (applications that perform frequent allocations) on power consumption.

Ray Tracing

The ray-tracing test was designed to more closely resemble a "real-world" application. It is an (incomplete) implementation of the simple ray tracing application described in [32]. This was our first attempt at a more real-world benchmark, under the hypothesis that a more complex benchmark would better stress the CPU and bring more accurate results than a microbenchmark.

While not currently implemented (see 4.1.3 for the current multithreading implementation), this workload should be easy to parallelize and scale indefinitely.

Stress

The stress test uses the Linux program, stress [35] to put the system under load for a period of time. Like the original sleep and spin loop tests, this was not intended to be an effective benchmark, but a simple test for our framework.

Phoronix Test Suite

At some point in our testing the need for more in-depth tests was established. We turned our attention to the Phoronix Test Suite [27], a set of open-source community benchmarks built on an extensible community platform. While our implementation does not use the Phoronix Test Suite harness itself, we have adapted a few of their tests to our process.

1. 7zip [1]

2. Linux Kernel Build [36]

3. x265 [41]

All tests were chosen specifically because they exhibited good scaling over multiple threads.

4.1.3 Multithreading

Our approach to running tests on multiple threads evolved as development progressed. Originally, the approach was simple - tests would be built single-threaded, and then run multiple times simultaneously, over multiple threads. This approach had limitations. Measurement started from the start of the first test, to the end of the last test. We assumed each test would run independently and finish at approximately the same time. This was not the case.

In our experiences, when running basic tests on multiple threads, they would end at different points. While we were not able to confirm any assumptions, we assume that contention was happening either on the shared cache level, the memory reading level, or through shared hardware in the hyperthreading cores.

As further development progressed, and we developed the capability to run external tests, rather than only our own implemented code, a different approach to multithreading was developed. These tests (Stress and the Phoronix Test Suite) run with secondary processes, rather than in the main process as all previous tests have. To limit which cores they can run on, we attach these processes to cgroups, using the cpuset subsystem to limit which CPUs they run on. Then, on a per-test basis, we instruct them to use as many cores as possible, if necessary.

4.1.4 Calculating energy consumption

If Powercap (3.1.2) is used to determine the power consumption, getting both current power usage and the total amount of energy consumed is simple. Powercap exposes the current power in microwatts, and the total amount of energy used in microjoules. We can work with these values directly, or simply divide them to get the current power in watts, and the total consumed energy in joules.
Calculating the energy usage by reading the MSRs (3.1.3, 3.1.1) (whether directly in a bare metal, or using the msr interface in Linux), is slightly more involved. The `MSR_PKG_ENERGY_STATUS` register stores how much energy has been used, however, as mentioned in Intel RAPL the time and power units can vary between platforms. This information (along with the power unit and time unit) is stored in the `MSR_RAPL_POWER_UNIT` register, and must be retrieved before the energy usage can be calculated.

Listing 4.1: Calculating the power usage from MSR values.

```
#define MSR_RAPL_POWER_UNIT 1542
#define MSR_PKG_ENERGY_STATUS 1553

uint64_t MSRInterpretPowerInterface::getCurrentJoules()
{
    if (!this->registered)
    {
        this->registered = true;
        this->shift = this->getShiftMultiplier();
    }
    uint64_t power = getRegister(MSR_PKG_ENERGY_STATUS);
    return (power * 1000) >> shift;
}

#define ENERGY_MASK 0b1111100000000
uint64_t MSRInterpretPowerInterface::getShiftMultiplier()
{
    uint64_t reg = getRegister(MSR_RAPL_POWER_UNIT);
    reg = (reg & ENERGY_MASK) >> 8;
    return reg;
}
```

Listing 4.1 shows how we calculate the current value of the total amount of consumed joules using the raw values from the MSRs. This calculation is the same on both Linux (in which case the `getRegister()` call will use the function in Listing 3.3), and on bare metal (in which case the function in Listing 3.1 will be used).
The values for `MSR_RAPL_POWER_UNIT` and `MSR_PKG_ENERGY_STATUS` are the addresses of the corresponding registers, as defined in the Intel Developer Manual [13]. The value for `ENERGY_MASK` is a bit mask that can be used to retrieve the Energy Status Units stored in bits 12:8 of the `MSR_RAPL_POWER_UNIT` register.

Once the amount total energy used can be retrieved, we can calculate the energy used by each test by measuring directly before and after a test is run, and calculating the difference between the two values. The power usage of each test can be similarly calculated by measuring how long a test takes, and then dividing the energy used with the duration.

Listing 4.2: Measuring test power usage and duration

```
TestDataPoint Harness::runTest(function<Test*()> factory, uint32_t*
    ↪ threadids, uint32_t numthreads)
{
    Test** teststorage = new Test*[numthreads];
    for (uint32_t i = 0; i < numthreads; i++)
        teststorage[i] = factory();
    uint64_t start_time = timeMs();
    uint64_t start = this->interface->getCurrentJoules();
    for (uint32_t i = 0; i < numthreads; i++)
        this->threads[threadids[i]]->runTest(teststorage[i]);
    for (uint32_t i = 0; i < numthreads; i++)
        this->threads[threadids[i]]->waitForFinish();
    uint64_t end = this->interface->getCurrentJoules();
    uint64_t end_time = timeMs();
    uint64_t value = end - start;
    TestDataPoint tdp = {
        .powerUsed = value,
        .timeTakenMs = end_time - start_time
    };
    for (uint32_t i = 0; i < numthreads; i++)
        delete teststorage[i];
    delete[] teststorage;
    return tdp;
}

```

Listing 4.2 shows our core logic for running tests. The current power is acquired before the tests begin, and after the tests finish. This captures both the time taken to finish the tests, and the total joules consumed by the system in that time. The power in watts can be easily calculated from this.

4.1.5 Profiler Output

The output of the profiler is a .json file that records the results of all performed benchmarks. Specifically, it contains the timer's resolution, and a list of entries for each benchmark. Each of these entries contain the following data:

- **Name:** The name of the benchmark (e.g. 7zip, raytracing, etc.)
- **Thread count:** The number of threads that were used when running this specific instance of the benchmark.
- **Hyperthread enabled:** Whether this instance of the benchmark was run in hyperthreaded or non-hyperthreaded mode.

- **Benchmark results:** A list of entries for each run that was performed by this instance of the benchmark. These contain the following information:
  - **Time taken:** The time it took for the run to finish, in milliseconds.
  - **Power used:** The power used by the given run, in millijoules.

4.2 Process Tracker

The process tracker is a relatively simple Rust utility, with a single purpose: to collect the CPU usage information of currently running processes.

4.2.1 Configuration Options

There are several configuration options, which can be set based on the current requirements:

Duration: This parameter controls how long the tracker will run for.

1. **Set time**: run the process tracker for a pre-determined length of time.
2. **While a process is running**: run the process tracker alongside another process, and stop when that process terminates.
3. **Indefinitely**: run the process tracker indefinitely.

Target: This setting will control which process or processes will be tracked.

1. **All**: collect information of all running processes.
2. **Single process**: collect information of a single running process, determined by the process' PID.

These options can be combined, for example, if we are only interested in the CPU usage of a single process with a known PID, we can configure the process tracker to only record that processes' information, and to shut down when the process terminates.
Since the tracker can also be run indefinitely, it is necessary to periodically save the collected information. This prevents data loss in case if a crash, or system power-down, and will reduce the memory usage of the tracker (as it will not be necessary to keep all measurements in memory). It is possible to configure how often the measurements are saved.
Finally, the user can set the frequency with which the data is collected. This involves a trade-off that must be considered on a case-by case basis: higher frequency will mean better accuracy, however, it will also cause higher CPU (and thus power) usage. Additionally, capturing more measurements will mean higher memory usage. This can be somewhat offset by configuring the tracker to write the results more often, but that solution also introduces some overhead (e.g. CPU time spent serializing the measurements, and disk I/O).

4.2.2 Process Data Collection

The process tracker will collect the necessary information using the *proc* pseudo-file system [30]. It provides a file-system like interface to the Linux kernel, allowing us to retrieve information as if we were simply reading files, similarly to using the MSR (3.1.1) or Powercap (3.1.2) interfaces.
Within \proc there is a numerical "subdirectory" for each running process, with the following format: \proc\*[pid]*\\. To track each processes' CPU usage, we collect data from \proc\*[pid]\stat* and \proc\*[pid]\task*\\.
From the stat file we retrieve the following information:

- The process' PID (which is the same as the subdirectory name).
- The command that was used to start the process.
- The state of the process (e.g. Sleeping, Running, etc.).
- The time the process has been scheduled in kernel mode (stime).
- The time the process has been scheduled in user mode (utime).
- The logical processor on which the process was last executed.

The \proc\*[pid]\task*\\ directory contains one subdirectory (\proc\*[pid]\task\[tid]*) for each thread started by the process with the given PID. Within these subdirectories, there are files with the same name and contents as in \proc\*[pid]*. From here, we collect the same data as we do for the top-level process.

4.2.3 Process Tracker Output

When the tracker is first started, it first collects some details about the system it is running on. Currently, this includes two pieces of information.
The first is the hyperthread layout of the system (that is, a list of hyperthreads, their siblings, and a map that shows which physical core the hyperthread belongs to). We collect this, so during data analysis we can map each running process to a hyperthread and to a physical core.
The second is the CLC_TCK parameter (collected by running the *getconf CLK_TCK* command). This value can be used to calculate the CPU time used by a process.
After this, the tracker takes measurements at the pre-configured interval. Each of these measurements contain the following information:

- UNIX timestamp (milliseconds since epoch).
- Total millijoules consumed.
- A list, containing details of all currently running processes. For each of these processes we collect the data described in 4.2.2.

The tracker writes these measurements to a file at the given interval. At the moment, a new file is created every time the measurements are written. This means that there is no chance of corrupting the previous measurements (e.g. if the tracker crashes while writing the file), but will result in large number of output files, especially if run for a long time and/or it is configured to write the results often.
Additionally, the tracker currently produces .json files as output. This has some advantages, such as easy parsing (both by humans and machines), but it also means that the resulting files are quite verbose, and potentially large. This may cause problems if the tracker is run on a device with limited storage (e.g. embedded systems).

4.3 Analysis Scripts

Both the profiler (4.1) and the process tracker (4.2) produce data only, without actually analyzing the results. To perform this analysis we have created a number of scripts using Python and C#. We have selected Python because of the ease of scripting, and C# because of the data transformation capabilities of LINQ, as well as our familiarity with both languages. However, as the profiler and the process tracker output raw data, it is possible to implement this analysis step using different languages, tools, or frameworks depending on the specific needs of the user.

4.3.1 Using the Process Tracker Results

As described in Process Tracker Output, the process tracker produces a series of .json files. Each file contains a list of measurements containing data about the system's uptime, power consumption, and the state of the running processes at the time of the measurement. From these, we can calculate some additional information:

- **Delta time:** The difference between the timestamp values of two consecutive measurements. Similarly to the **delta uptime** value, it should be constant, but can vary slightly between measurements. This provides better resolution than the **delta uptime** value (milliseconds instead of seconds).

- **Delta process time:** For each process that was running, we measure its current **utime** and **stime**, corresponding to the time spent in userspace and in the kernel relating to this program. These are measured in jiffies. With the hz rate of jiffies from the system config, we can calculate the delta process time, which is the time in milliseconds the process has been active, in the last time step.

- **Delta millijoules:** The difference between the millijoules consumed between two consecutive measurements. In other words, this shows how much power was used since the last measurement.

- **Watts:** The number of watts consumed at the time the measurement was taken. It is calculated by dividing the **delta millijoules** value with the **delta time**.

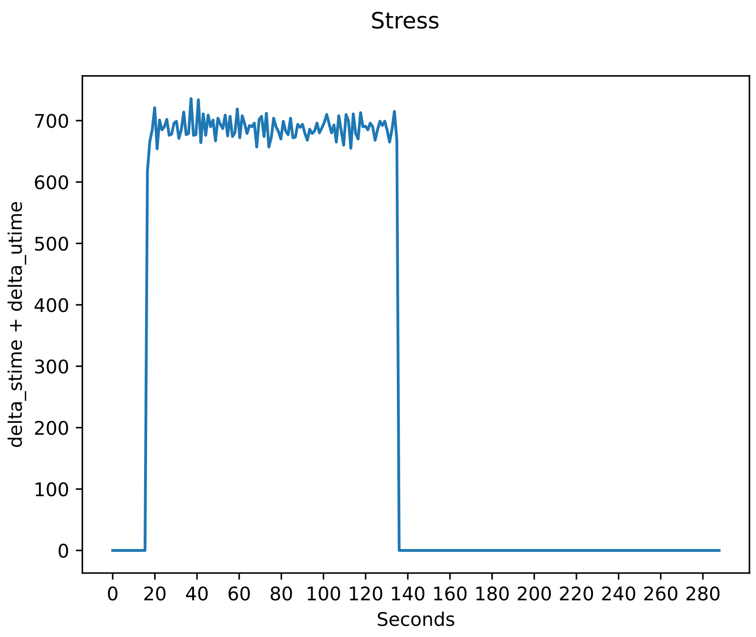

Figure 4.1: The delta process time values when running the “stress” command.

Figure 4.1 plots the **delta process time** values for the process started by the “stress” command. It shows that the process was running for 120 seconds. While it was running, it used approximately 650-700 jiffies worth of CPU time between each measurement.

Calculating CPU And Power Usage

Using the data provided by the process tracker, as well as the additional information we calculated, we can determine the following:

- CPU utilization percentage.
- Total millijoules consumed.
- Watts consumed.
- Process CPU utilization.
- Watts consumed by a process.

Note, these values can only be calculated if the process tracker recorded information on all running processes (“Target” was set to “All”), as we need the total CPU time used by all processes.

CPU Utilization

To calculate the overall CPU utilization of the system, we must first compute the CPU time used by all processes for each measurement (n is the number of processes):

$$total\_cpu\_time\_clktck = \sum_{p=1}^{n} process_p.delta\_stime + process_p.delta\_utime$$

As *utime* and *stime* values are measured in jiffies, the resulting value must be divided with the CLK_TCK parameter to convert it to seconds.

$$total\_cpu\_time = \frac{total\_cpu\_time\_clktck}{CLK\_TCK}$$

Now, *total_cpu_time* is the CPU time used since the last measurement. To convert it to a percentage, we simply divide it with the amount of time elapsed since the last measurement:

$$cpu\_util\_percentage = \frac{total\_cpu\_time}{delta\_time}$$

We can perform this calculation for each measurement, and plot the results:

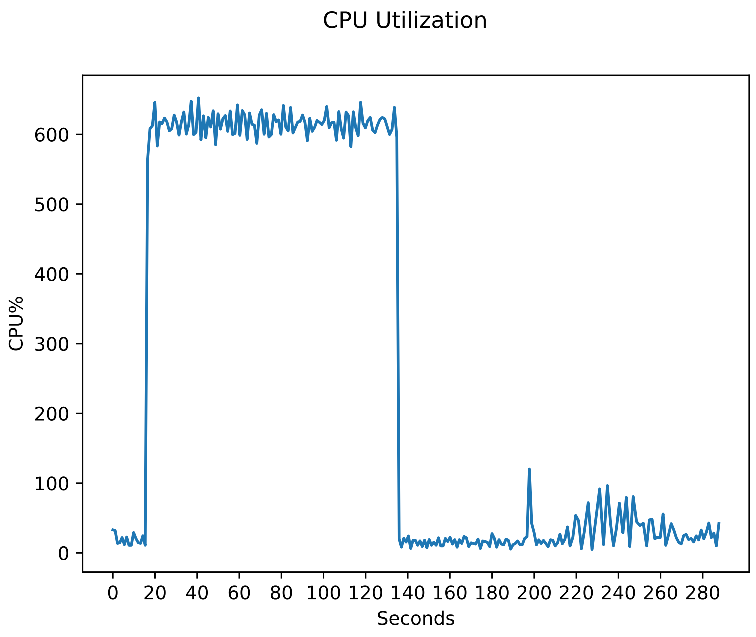

Figure 4.2: CPU Usage Percentage.

Figure 4.2 shows the CPU utilization over approximately 290 seconds. The same dataset as in Figure 4.1 was used.
This method of calculating the CPU usage does not take into account multiple cores. For example, if a process uses two threads for 1 seconds, the total CPU time consumed will be 2

seconds. As only one second of real time passed, the process' CPU usage will be 200%. Figure 4.2 shows that some process (in this case "stress") was using 600% of the CPU, as it was configured to run on six threads.

Overall Power Consumption

The total power used can simply be calculated by subtracting the total amount of millijoules recorded in the first measurement from the total amount recorded in the last measurement. Additionally, the watts used at each measurement is also calculated, as described above.

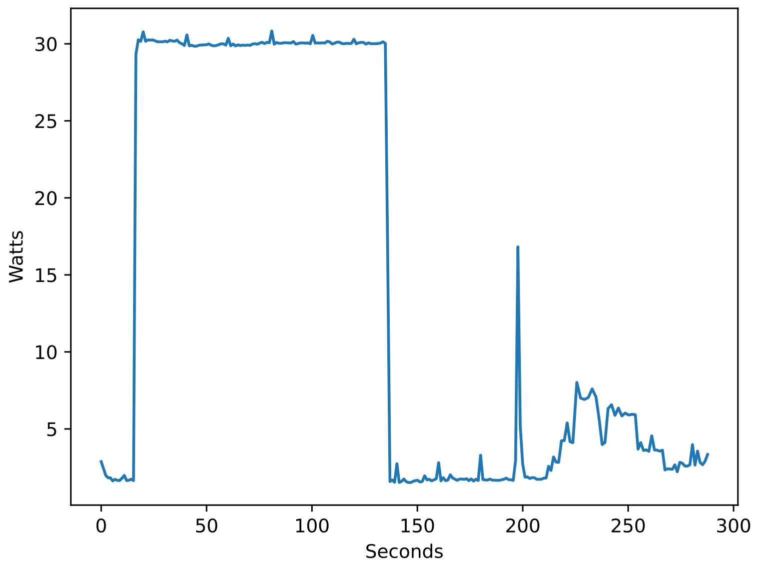

Figure 4.3: Total Amount of Watts Used

Figure 4.3 shows both the amount of millijoules and the amount of watts used during the measurement period. There is visible correlation between CPU (Figure 4.2) and power usage.

Process CPU Usage

We have two ways to calculate a specific process' CPU usage. The first is the same procedure we used to determine the overall (system) CPU utilization: for each measurement we sum up the process' delta stime and delta utime values witch is then divided by the CLK_TCK value. This result is then divided again by the measurement's delta time value. Just as before, this can indicate that a specific process used more than 100% of the CPU, if it was running over multiple threads.

Figure 4.4 shows the CPU used by the "stress" (which is configured to use six threads) and the "firefox" processes.

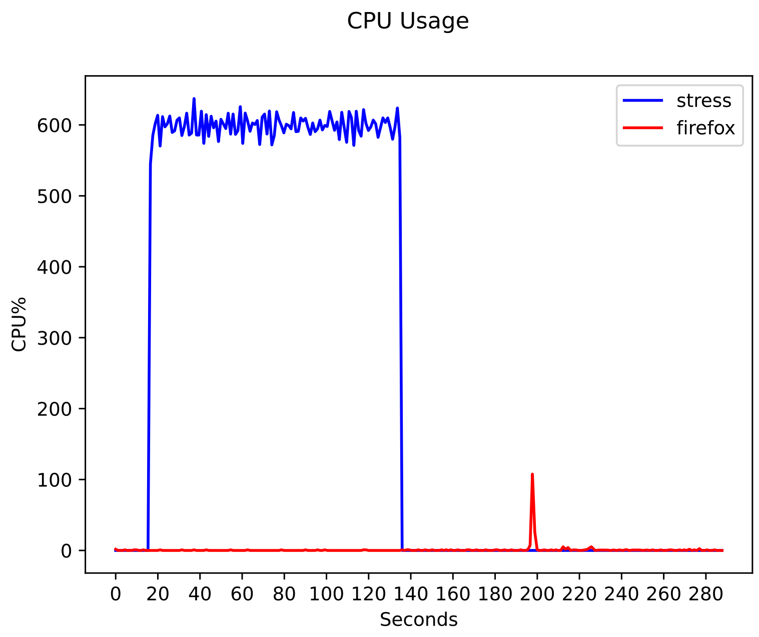

Figure 4.4: CPU Usage

An alternative way of calculating a process' CPU usage is to first compute the total amount of CPU time used by the target process p:

$process\_cpu\_time = (p.delta\_stime + p.delta\_utime)$

We then calculate the CPU time used by all processes:

$total\_cpu\_time = \sum_{p=1}^{n} process_p.delta\_stime + process_p.delta\_utime$

With these, we can get the percentage of CPU time spent by process p:

$cpu\_percentage = (\frac{process\_cpu\_time}{total\_cpu\_time}) \cdot 100$

This approach scales the resulting values between 0 and 100.

Figure 4.5 shows the results of plotting the same two processes as in Figure 4.4 using this alternative solution.

Process Power Consumption

From the values calculated above, we can estimate the power usage of each process. We call this a "naive" estimation, as it does not take into account hyperthreading, or data from the

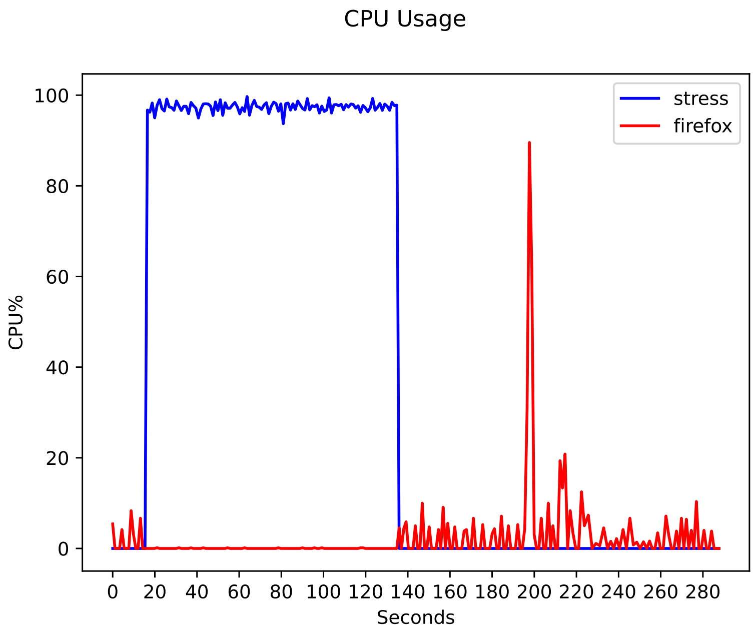

Figure 4.5: CPU Usage

profiler.

This estimation works by simply dividing up the amount of watts (or millijoules) used at a given measurement based on the processes' CPU usage. For example, if at a given moment, 30 watts were consumed, and a particular process used 50% of the CPU, then that process will be attributed 50% of the power consumption, that is, 15 watts. It is essentially the same approach as in [10].

$$process_p\_watts = watts \cdot (\frac{process_p.process\_cpu\_time}{total\_cpu\_time})$$

Figure 4.6 shows the estimated watts used by two processes, "stress" and "firefox". Again, there is a visible correlation between the processes' CPU (Figure 4.5), and power usage.

Additionally, we have a mechanism to calculate the "naive hyperthreaded" power consumption. This is similar to the naive estimation, where we first allocate power per physical core, then per process on that core.

$$core_n\_watts = watts \cdot (\frac{\Sigma core_n\_processes.process\_cpu\_time}{total\_cpu\_time})$$
$$process_p\_watts = core_n\_watts \cdot (\frac{process_p.process\_cpu\_time}{core_n\_cpu\_time})$$

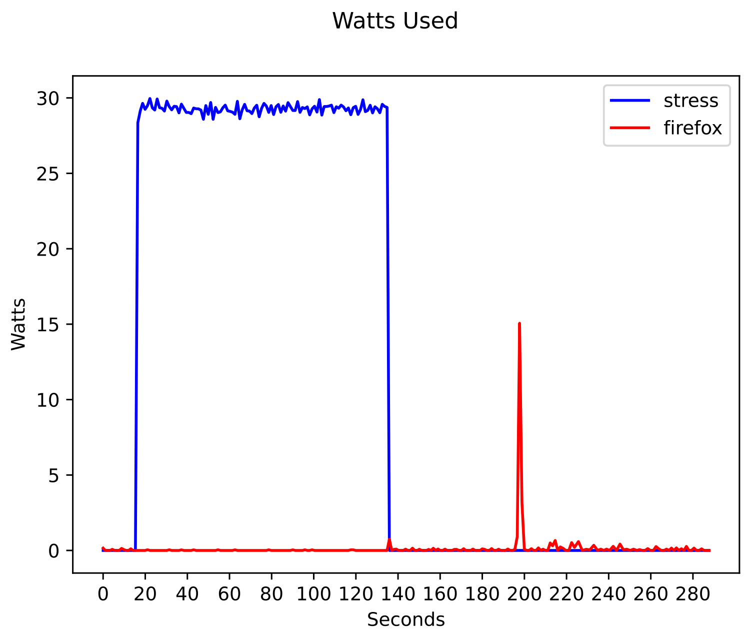

Figure 4.6: Estimated Watts Used

4.3.2 Using the Profiler Results

To use the data produced by the profiler, the Python script loads the .json output, which has the contents described in Profiler Output.
Using the information from the profiler, we can calculate the **static power** and **hyperthread ratio**, as described in [42].

Static Power

The static power is the power consumed by the CPU while no physical cores are in use. It is calculated by linearly extrapolating from the power consumption of multiple active physical cores. For example, if for a 4-core machine we measure 10, 15, 20, and 25 watts used when 1, 2, 3, and 4 cores are active respectively, we can extrapolate that the energy usage when 0 cores are in use is 5 watts.

Figure 4.7 shows the power usage per active core on a 6-core machine, calculated from all non-hyperthreaded tests. It also shows the extrapolated power usage for 0 active cores, which in this case is approximately 5.25 watts.

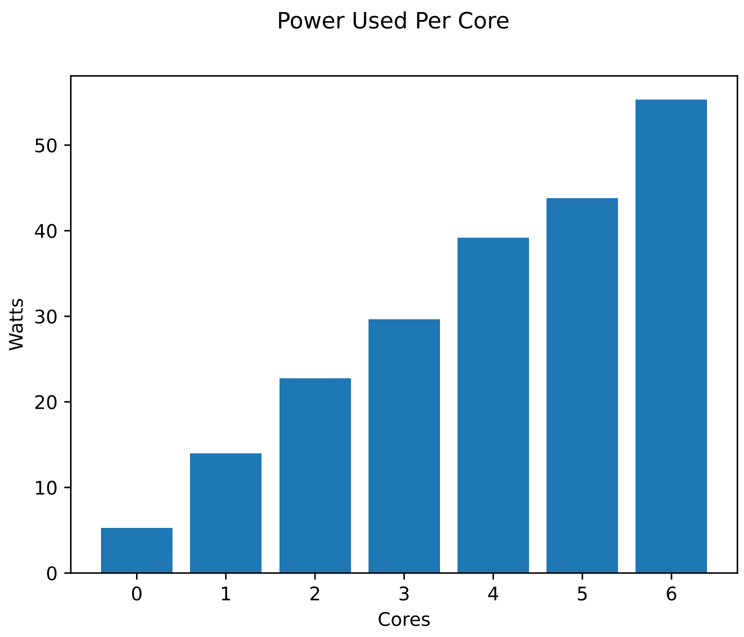

Figure 4.7: Power Usage Per Core

The static power under use, however, can vary. Static power in this context refers to the power used by the CPU while the cores are active, but that is not attributable to a specific core. When the CPU is not under any load, it can disable itself to a level where the static power drops dramatically. We term this the idle static power. We can estimate the actual static power consumption quite simply using the CPU active time.

$$cpu\_active\_time\_ratio = \max\left(\frac{core_n\_active\_time\_ms}{measurement\_duration\_ms}\right)$$

$$static\_power\_actual = static\_power\_idle + (static\_power - static\_power\_idle) \cdot cpu\_active\_time\_ratio$$

Hyperthread Ratio

The hyperthread ratio is the ratio between the energy consumption when only one thread is active versus when both hyperthreads are active per core. As per [42], using both hyperthreads of a physical core only increases the power consumption slightly, as most low-level hardware components are shared between hyperthreads.

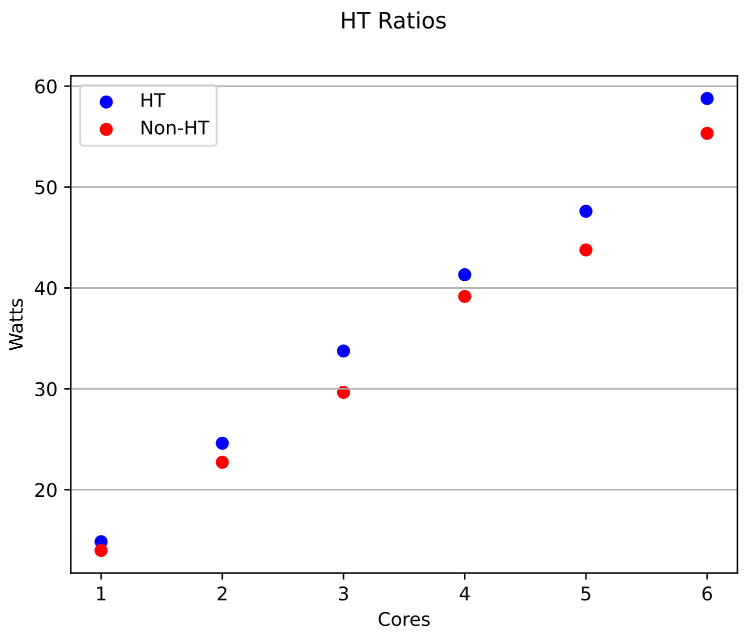

Figure 4.8: Hyperthread Ratio

This is illustrated by Figure 4.8. It shows the calculated energy consumption per physical core when only one thread is running (blue), versus when both hyperthreads are in use (red). Aggregating the results shows that the hyperthread ratio of this machine is approximately 1.11. In this case, the ratio is computed over all cores, however, it is also possible to retrieve it on a per-core basis:

Cores	HT Ratio
1	1.128
2	1.147
3	1.189
4	1.082
5	1.113
6	1.074

Table 4.1: HT Ratios Per Active Physical Cores

Hyperthreaded Power Consumption

Finally, with these values, we can calculate the hyperthreaded power consumption. First, we subtract the static power usage.

$$process\_power = total\_power - static\_power\_actual$$

Then, we can assign the power similarly to the naive power estimation. We calculate two values, for physical and logical cores.

$$logical\_core_n\_power = process\_power \cdot (\frac{\Sigma logical\_core_n\_processes.process\_cpu\_time}{total\_cpu\_time})$$

$$physical\_core_n\_power = process\_power \cdot (\frac{\Sigma physical\_core_n\_processes.process\_cpu\_time}{total\_cpu\_time})$$

We can calculate the hyperthreading power — the power a process would have used, were it running alone on a core — using the hyperthreading ratio.

$$physical\_core_n\_hyperthread\_power = physical\_coren\_power / ht\_ratio$$

Then, the calculation to calculate the process' power consumption depends on whether the thread was currently running hyperthreaded — that is, assigned to a physical core with another running thread — or not. We assume for each measurement taken that if any other threads ran on that core, every process that ran on that physical core was hyperthreaded. This can reduce accuracy — if a thread runs hyperthreaded for only a small fraction of its runtime, the calculation will be inaccurate, but we do not have the data to determine how much was hyperthreaded, and how much was not.

$$process_p\_power = \begin{cases} RunningHyperthreaded & physical\_core_n\_hyperthread\_power \cdot \frac{process_p\_time}{logical\_core_n\_active\_time} \\ RunningAlone & physical\_core_n\_power \cdot \frac{process_p\_time}{logical\_core_n\_active\_time} \end{cases}$$

This calculation estimates the virtual power consumption of a hyperthreaded program. In the case of a program running hyperthreaded, it can be more than the actual power consumption. The issues and pitfalls with this approach are explored more in section 6.3.

Chapter 5

Results

5.1 System Configuration

Our tests were run on a system with the following specifications:

- **CPU:** Intel(R) Xeon(R) W-1250P CPU @ 4.10GHz
- **RAM:** 16 GiB DDR4 3200MHz
- **Storage:** 512 GB NVMe SSD (Samsung MZVLB512HBJQ-000L7)
- **OS:** Arch Linux

For the exact components, as well as the list of installed packages, please see Appendix A

5.2 Profiler Results

On our test machine, the profiler produced the following results. The Arrayfire, 7zip and x265 tests were run with 25 iterations each. These numbers were chosen so we could get the maximum number of test results in a manageable amount of time. Obviously-incorrect outliers (any calculation that produced a value over 10,000 watts) were filtered. We used Powercap to get the energy readings.

Overall, four results were filtered — one from 7zip with 4 cores, 7 threads, one from x265 with 4 cores, 8 threads, one from x265 with 5 cores, 10 threads, and one from arrayfire with 4 cores, 7 threads. It is likely these results are from RAPL overflows in some manner. Through some investigation of the Linux Kernel source code, Linux's powercap interface does not handle RAPL overflows in any meaningful way.

With this data we calculated the hyperthreading coefficient to be approximately 1.13. To do this, we paired each test without hyperthreading with the appropiate test with hyperthreading, e.g 6 cores 6 threads was paired with 6 cores 12 threads. We then averaged the results over every test.

Table 5.1: Average Power Consumption, with hyperthreading enabled and Intel Turbo Boost disabled.

Average Power Consumption	**7Zip**	**x265**	**Arrayfire**
1 Core, 1 Thread	12.072 W	14.340 W	15.054 W
1 Core, 2 Threads	13.168 W	16.865 W	16.530 W
2 Cores, 3 Threads	20.488 W	26.292 W	23.527 W
2 Cores, 4 Threads	22.399 W	28.812 W	27.376 W
3 Cores, 5 Threads	28.257 W	37.315 W	34.073 W
3 Cores, 6 Threads	30.742 W	39.385 W	37.877 W
4 Cores, 7 Threads	36.586 W	47.017 W	**41.246 W**
4 Cores, 8 Threads	39.848 W	48.706 W	**41.281 W**
5 Cores, 9 Threads	44.560 W	54.802 W	**44.322 W**
5 Cores, 10 Threads	48.260 W	57.014 W	**44.160 W**
6 Cores, 11 Threads	54.554 W	62.258 W	**61.609 W**
6 Cores, 12 Threads	56.595 W	63.416 W	**61.705 W**
1 Core, 1 Thread	11.937 W	14.381 W	15.117 W
2 Cores, 2 Threads	18.564 W	23.903 W	24.809 W
3 Cores, 3 Threads	24.280 W	33.133 W	33.338 W
4 Cores, 4 Threads	32.762 W	41.729 W	43.429 W
5 Cores, 5 Threads	35.356 W	59.520 W	51.662 W
6 Cores, 6 Threads	47.646 W	58.471 W	61.632 W

We also calculated the static power consumption to be 9.416 watts. This was taken by averaging every calcuation over each hyperthreaded thread configuration, then linearly extrapolating to 0 cores, 0 threads. This is not the true static power consumpion, as compared to the computer under no load — a seperate test calculated that to be approximately 0.356 watts.

Some tests did not perform well in a hyperthreaded environment. Arrayfire, for instance, did not have any noticable difference in average power consumption when active on four, five, or six cores. The relevant values have been bolded in table 5.1. Further testing is required to determine how well Arrayfire utilizes hyperthreads. These values have not been excluded in the calculation of the hyperthreading coefficient.

We calculated two other profiles. One with hyperthreading disabled completely, and one with turbo boost enabled.

Table 5.2: Average Power Consumption, with hyperthreading disabled and Intel Turbo Boost disabled.

Average Power Consumption	**7zip**	**x265**	**Arrayfire**
1 Core	11.693 W	14.451 W	14.863 W
2 Cores	18.461 W	24.137 W	24.370 W
3 Cores	24.478 W	33.333 W	33.230 W
4 Cores	32.979 W	41.144 W	42.898 W
5 Cores	35.236 W	47.521 W	51.053 W
6 Cores	47.195 W	55.237 W	60.771 W

With hyperthreading disabled, the average power usage overall seems lower, even as compared to single-thread-per-core tests with hyperthreading enabled. While this difference is small enough it could be attributed to noise, it's also possible that disabling the hyperthreading hardware entirey can lower power consumption, even if by only a small amount. There was no significant difference in time taken between hyperthreading enabled and hyperthreading disabled, for comparable tests with one thread per core.

Table 5.3: Average Power Consumption, with hyperthreading enabled and Intel Turbo Boost enabled.

Average Power Consumption	**7zip**	**x265**	**Arrayfire**
1 Core, 1 Thread	23.529 W	28.998 W	29.608 W
1 Core, 2 Threads	26.929 W	33.653 W	32.815 W
2 Cores, 1 Thread	40.609 W	51.733 W	45.943 W
2 Cores, 2 Threads	44.205 W	56.413 W	52.933 W
3 Cores, 1 Thread	49.122 W	64.347 W	58.949 W
3 Cores, 2 Threads	53.369 W	68.021 W	65.125 W
4 Cores, 1 Thread	61.810 W	80.670 W	71.525 W
4 Cores, 2 Threads	67.579 W	83.221 W	71.391 W
5 Cores, 1 Thread	63.069 W	77.054 W	62.733 W
5 Cores, 2 Threads	67.460 W	80.080 W	62.697 W
6 Cores, 1 Thread	75.934 W	87.831 W	87.674 W
6 Cores, 2 Threads	78.714 W	89.335 W	87.898 W

The problems with turbo boost become immediately apparent when the power usage goes down from four cores to five cores, as the CPU is unwilling to continue using turbo boost frequencies. This breaks core assumptions made by our models.

5.3 Tracker Results

For our initial tests, we ran a number of Phoronix Benchmarking Suite tests with the process tracker active. These tests include:

- **Mocassin**, monte carlo simulations of ionised nebulae.
- **Stockfish**, an advanced open-source chess benchmark.
- **Z3**, a theorem prover and SMT solver.
- **x264**, a multi-threaded video encoder.

Each test was run thirty times, after which the results were graphed and inspected visually. Many tests had to be excluded for faults and obvious inaccuracies. We chose thirty iterations to complete tests in a reasonable amount of time.

The Mocassin test had the largest number of invalid tests, with seventeen of the tests having to be excluded. Fourteen of these tests exhibited similar issues, with the hyperthreading

Table 5.4: Results (in Joules) of each profiled test.

	Average	**Standard Deviation**	**Minimum**	**Maximum**	**Valid Tests**
Mocassin	51490 J	833.32 J	50773 J	53965 J	13
Stockfish	98468 J	7129.5 J	86395 J	112251 J	25
Z3	326.76 J	5.09 J	317 J	342 J	30
x264	5703.4 J	45.21 J	5589 J	5768 J	27

measurement being far more unstable than a valid test. This is likely due to some artifact introduced in our measurements or calculations.

Other invalid results included:

- An instance where Mocassin did not seem to use any meaningful power. We suspect the program crashed or failed to run for some reason.
- An instance where the estimated power consumption was 20% of the others.
- An instance where the estimated power consumption was greater than the total power consumed.

The Stockfish test exhibited similar issues, though its invalid graphs had a different appearance to Mocassin. Another thing of note is the large variation in power consumption in the Stockfish test. This may be due to the test itself, as other tests are more consistent in this regard. Stockfish does not appear to have a consistent execution pattern, which is visible in individual test graphs.

The Z3 and x264 tests, by comparision, were far more well behaved than the Stockfish or Mocassin tests. We suspect this is due to their more deterministic nature, as well as being shorter tests overall. Only three tests had to be excluded from x264, appearing to exhibit similar-looking flaws as to the Stockfish tests.

Graphs for a variety of these tests can be found in Appendix B.

5.4 Non-Hyperthreaded Tracker Results

We re-ran our tests with hyperthreading disabled, and used the non-hyperthreading profile to generate the graphs and data. Other than this, the test parameters were exactly the same as the hyperthreading results.

Table 5.5: Results (in Joules) of each profiled test, with hyperthreading disabled.

	Average	**Standard Deviation**	**Minimum**	**Maximum**	**Valid Tests**
Mocassin	*46870 J*	*3345 J*	*29998 J*	*49128 J*	*29*
Stockfish	34891 J	2780 J	29825 J	40777 J	28
Z3	300.72 J	24.57 J	213 J	325 J	29
x264	2913.8 J	26.88 J	2833 J	2953 J	25

As compared to the hyperthreading-enabled results, the power consumption seems decreased across the board. Of particular note was the Mocassin test. Every valid test in the Mocassin test exhibited the strange behaviour that we classified as invalid in the multithreading test. Further improvement and refinement of our algorithm seems necessary.

5.5 Test Under Load

Our final test was to test the algorithm under load. We put the system under stress with the stress command, saturating every core with load. Then, we ran the z3 test. The assumption is that, z3 is doing the same amount of computation and should use the same amount of power.

However, our algorithm failed to produce a useful result under these. Instead, we attempted to compare the naive algorithm both under load and not under load, as well as our algorithm not under load.

Table 5.6: Z3 tests, with and without load, calculated with different algorithms.

	Average	**Standard Deviation**	**Minimum**	**Maximum**
Z3 With Load	194.1 J	3.95 J	187 J	202 J
Z3	326.9 J	4.99 J	318 J	342 J
Z3 Our Algorithm[1]	359.2 J	5.09 J	349 J	374 J

It is clear either our assumption that the z3 test would use the same amount of power is incorrect, or the naive algorithm cannot accurately model the power consumption. In addition, our algorithm produced a different result for the power consumption than the naive algorithm. Further research is required.

[1]The values here are different from the values in 5.3 because those values did not add the static power consumption. The naive values were not calculated with the static power consumption removed, so we added the static power consumption back in for these values.

Chapter 6

Development and Discussion

6.1 Profiler

Initial development of the profiler began with a simple base, focusing first on getting some tests up and running, as well as results reporting. At this time the results weren't compiled into a profile, but instead just reported on the screen.

The profiler was built to be extensible and portable, and at this point can run both directly under Linux, as a userspace program, and as a UEFI bare-metal application. Under Linux it supports both reading the Powercap interface as well as the RAPL MSRs directly (using the /dev/cpu/*/msr interface). For the purposes of these tests, the Linux implementation reads the registers directly. This was done mostly to ensure our calculations between the raw register value and the joules were accurate.

Initial tests were performed with two very simple, basic tests. One was a spin-loop, using library-provided functions to check the time repeatedly until a set amount of time had passed. The other was a sleep test, using a library-provided function to do no work for a set amount of time. In both of these cases, the tests were configured to run for ten seconds each, and to run fifteen times each in total. Under Linux, background processes were kept to a minimum of what was required to run the machine. Under UEFI, the initial assumption was that the only code running will be the code we wrote, and code from libraries we called. While this assumption turned out not to be entirely accurate, as there is an interrupt handler that handles periodic timer interrupts, we still expected the UEFI version to still produce more accurate results than the Linux version.

The results were immediately surprising, with two notable results. One was that the spin-loop power usage was not comparable across the Linux and UEFI implementation, being nearly twice as much on average under Linux than UEFI. The other was that the sleep test under UEFI was using almost as much power as the spinloop.

The theory for the sleep test was simple. Investigating the source code of POSIX UEFI[28] revealed that the sleep function we were using was implemented in terms of the UEFI Stall function[38]. The UEFI stall function is defined as a fine-grained stall which does not yield execution of the processor for the duration. While the actual implementation is firmware-

	Linux		**UEFI**	
	Spin	**Sleep**	**Spin**	**Sleep**
1	268.335 J	3.598 J	115.869 J	122.262 J
2	274.870 J	3.429 J	126.696 J	118.663 J
3	277.260 J	3.428 J	127.443 J	121.469 J
4	282.228 J	3.453 J	127.445 J	122.294 J
5	282.282 J	3.468 J	126.782 J	117.263 J
6	281.114 J	3.434 J	125.638 J	121.154 J
7	283.572 J	4.009 J	127.513 J	122.651 J
8	284.103 J	3.551 J	128.256 J	124.066 J
9	284.093 J	3.424 J	127.169 J	123.163 J
10	284.770 J	3.448 J	127.334 J	122.981 J
11	284.983 J	3.468 J	127.360 J	122.272 J
12	285.355 J	3.766 J	128.223 J	122.699 J
13	285.573 J	3.787 J	128.080 J	123.234 J
14	286.178 J	3.432 J	128.040 J	123.717 J
15	285.068 J	3.537 J	127.889 J	123.276 J

Table 6.1: Energy used by the spin and sleep tests under Linux and UEFI

dependent, it's reasonable to believe it could be implemented as a spin-loop, for fine grained control.

As for the different results for the spin-loop, multiple hypothesises were presented. One is that the UEFI environment does not have as fine grained control of the CPU's frequency and speed as Linux does. Another is that the implementation of the library functions differ signifigantly. A third, is that the overhead from Linux or other processes is affecting the results, but this seems unlikely. Other than the second hypothesis, we can assume from the results of the sleep test that Linux' influence is not on the same magnitude as the difference seen, as the total power usage of a system not doing anything is far lower.

To address the second hypothesis, a third test was devised. This test was based on an implementation of the Computer Language Benchmark Game's Fannkuch Redux test[6]. For this test, only five iterations of the tests were performed. Our assumption was that the code and running time for this should be identical, because it makes no library calls and was compiled with the same compiler flags, excluding what was necessary to compile to the different environments.

The results seem to confirm the hypothesis that there's some environmental effect influencing the power consumption. Unfortunately, due to time constraints, we were not able to continue to determine the true cause of it.

We were able to lower the energy consumption of the sleep test, to approximately 40J, by implementing a proper sleep function, using UEFI's events to set a timer, and halting the CPU using the HLT instruction [13]. The difference between this and the Linux sleep measurements is assumed to be due to the UEFI interrupt loop, which periodically awakens the CPU on a timer to service events.

To solve this, and take advantage of the full power of the CPU, we would have to bring

	Linux	UEFI
	fannkuch-redux	**fannkuch-redux**
1	1232.451 J	993.775 J
2	1238.991 J	991.534 J
3	1248.773 J	990.912 J
4	1245.948 J	989.649 J
5	1246.257 J	985.584 J

Table 6.2: Energy used by the fannkuch-redux test under Linux and UEFI

the CPU out of the UEFI Boot Services and take complete control of the system, running completely on the bare metal. To this end, several tasks would have to be completed.

1. Establishment of our own memory allocator.
2. Basic terminal capabilities, to write to the screen.
3. Activation of the High Precision Event Timer, or other timer hardware, to deliver an interrupt after a period of time.
4. Setting up interrupts on the CPU.
5. Multi-core capability.

In the time we had allocated to this, only some of these tasks were even partially completed. While we maintain that it's possible, and likely still useful, to continue this work and port the profiler to a truly bare-metal environment, we did not have the time, resources, or expertise required to do it. Even if we had, we would only have a program that worked on systems of this particular configuration. For example, while Intel CPUs implement the RAPL interface, AMD CPUs have a different RAPL implementation.

6.2 Bare-Metal Profiler

One of our initial goals with the profiler was to develop a version that could run and execute in a UEFI bare-metal environment. The assumption behind this was to take elimination of background processes to the extreme; running our code and our code alone in a bare-metal environment would allow us to acquire 'pure' samples.

This did not work as planned for a number of reasons. Development was an immediate issue, as programming in a bare metal environment was more difficult than programming under Linux. There are fewer libraries and API calls that could be used, which meant many things taken for granted under a standard operating system had to be programmed by hand, or gone without.

As development continued, even the UEFI boot services weren't enough for our needs, so we briefly branched out into pure bare-metal development. Under the UEFI boot services, we had access to a limited number of APIs [37] and limited library support. Not only that, but we suffered the constraints of the UEFI platform as well. By default, only one CPU core was active, and the CPU frequency was locked, with a consistent timer tick to service UEFI events. Shortly before we abandoned the UEFI bare-metal support, we considered going beyond the UEFI platform and going completely bare metal. But this required contending with various features of the CPU directly, such as CPU frequency setting, hardware P-states, and High Precision Event Timers [11].

Ultimately these efforts were struggling against a larger problem; the mismatch between the UEFI bare-metal environment, and a Linux environment. We theorized that this would allow us to get more pure measurements of the performance of the hardware, but we failed to take into account how complex the hardware actually was, and the environment that the operating system sets up to take advantage of these hardware features would be difficult to replicate exactly. The effort required to replicate a Linux-like CPU environment from scratch is far less than simply running Linux.

Furthermore, while relevant for our research, avoiding realistic environments for more 'pure' measurements may not achieve ideal results. As discussed in 6.3, there is no clear singular way to assign power usage to processes when hyperthreading is involved. Accurate measurements may only be attainable either with certain conditions or assumptions made, or as a complete system, with results unable to be generalized.

We still believe that further elimination of background processes is a desirable goal for future work. However, a pure bare-metal implementation may not be feasible, and other approaches should be investigated instead. One possible approach is using a Linux kernel, with userspace stripped down to the absolute minimum.

6.2.1 Design Decisions

Much of our initial design was affected by the goal for UEFI compatibility, which precluded the use of libraries. The Sleep Test, Spin Loop, Fannkuch-Redux, Binary Trees and Ray Tracing tests were all affected in a variety of ways.

The primary issue in design was one of extracting results. Printing the results to the screen and copying them manually is slow and error-prone, but UEFI provided functionality for opening and saving files on the EFI filesystem. We never implemented this functionality, but did not expect it to be difficult.

Sleep Test: The Sleep Test sleeps for a period of time, a conceptually simple test. However, the UEFI function we were using, Stall, did not put the CPU into an optimized sleep state, rendering the Sleep test's power consumption similar to the later Spin Loop test. We worked on an optimized sleep implementation to mitigate this. Our first attempt used UEFI events and a halting loop, which was able to achieve moderate power savings, but not enough to be comparable to Linux. We theorized a second, fully bare-metal attempt, using timer interrupts fully controlled by us, but abandoned the prospect before completing it.

Spin Loop: The spinloop test used an API call to get the current system time. Our interface, posix-UEFI, translated the equivilent UEFI call into a format similar to the localtime function. On UEFI, the time calls do not return time to a precision greater than one second, however, which limited this use. We contemplated using more precise timers, such as HPET, but decided against it.

Fannkuch-Redux: Fannkuch-Redux was chosen specifically because of it's simple, plain-C implementation. There were no major issues implementing or running this on UEFI.

Binary Trees: While there were no major issues implementing binary trees on UEFI, when we contemplated implementing it on the bare-metal, several issues arose in the terms of memory management. We would have to write our own memory management routines, as binary trees made strong use of memory allocation functions.

Raytracing: Raytracing as implemented ran into similar issues as binary trees in terms of memory allocation, but had a few additional functions not accessible in UEFI. The raytracing code has the ability to save the image to a file, which is possible under Linux but difficult under UEFI. While it could have been implemented, as UEFI does provide capability for manipulating the file system, we opted not to implement this as it gave us no useful benefit other than debugging.

6.3 Hyperthreading Power Measurement

The HaPPy paper [42] presents a hyperthreading-aware model of power measurement that is capable of assigning power consumption to both individual cores and individual hyperthreads. However, assigning power consumption to individual processes presents a greater challenge not addressed by this paper. The crux of the issue comes down to the 'hyperthreading multiplier', termed in HaPPY as R_{ht}. This defines the ratio of power usage by a fully-active hyperthreaded core, to a hyperthreaded core with only a single thread active. However, when both threads are active, the power per thread, when both threads are running, is evenly split in their model between the two threads.

This produces unintuitive results in a process-tracking model. We are assuming values of R_{ht} between 1 and 2, and that two processes scheduled on hyperthreads will not have adverse affects on the runtime of the other. When running a process on a core, the power used by the process will go down if a second process is scheduled on that core. But this is not accurate to the true state of things, as the total power usage has only slightly gone up. This would seem a great win in energy efficiency, but it can lead to a very temperamental power consumption, where the power consumption of a process now varies wildly on factors outside of the programmer's control. In addition, it breaks the intuition that stopping that program would reduce the power consumption of the system, by the amount of power that program used.

We have not determined a way to model power consumption with a single value such that both factors can be accounted for at once. However, accounting for either factor is simple independently.

Additionally, the HaPPY paper's model of verifying the calculations of the power used by a single process can be drawn into question. Their model works by assuming the power usage is relatively constant, such that if a program uses X amount of power, not running the program will lower the total power consumption by X. With the factors mentioned here, our results, as well as turbo boost in the following section, we do not believe this model generates accurate results for our use case.

6.4 Intel Turbo Boost

During initial testing we have noticed behavior that was inconsistent with both our expectations and with the behavior described in [42]. We assumed that power consumption will scale mostly linearly with the number of active physical cores. However, in our tests we have observed a phenomenon where more active cores would sometimes consume less power than fewer cores.

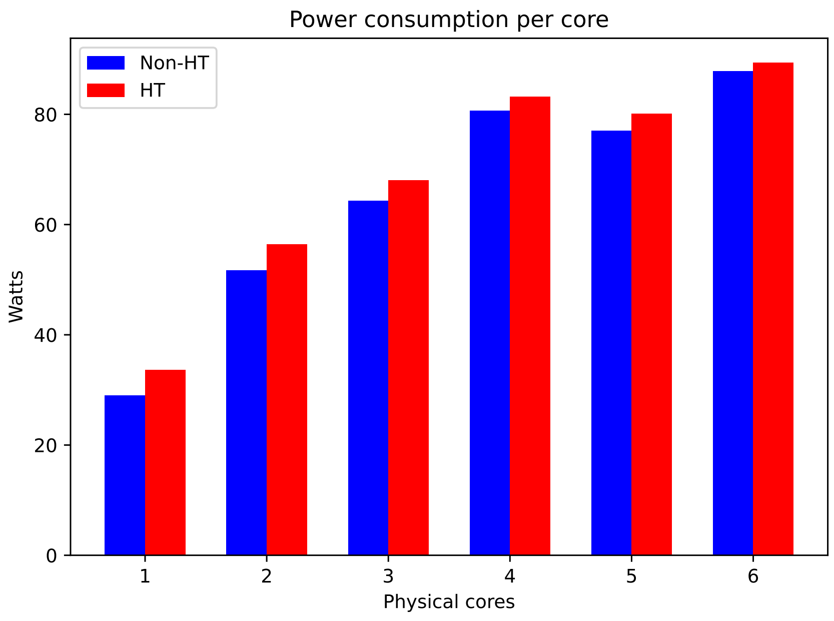

Figure 6.1: Inconsistent power consumption

Figure 6.1 illustrates this behavior. Each pair of columns in the graph represents a physical CPU core. The first bar in the pair is the power consumption when only one thread is running on that core (i.e. hyperthreading is not enabled). The second bar is the power consumed with hyperthreading enabled (i.e. there are two threads on one physical core).
In this example, running five physical cores simultaneously (with and without hyperthreading) used less power than using only four cores.
After investigating, we concluded that this behavior was caused by Intel Turbo Boost [16]. Turbo Boost increases the frequency of processor cores when certain criteria (such as low power consumption, low temperature, etc.) is met. This means that the effects of Turbo Boost are unpredictable, as we can not determine when, and how much it will increase the CPU frequency. Therefore, in order for our tests to produce consistent results, Turbo Boost must be disabled. Unfortunately, this means that our power usage estimations may be less accurate, if the system under test is using Turbo Boost when the process tracking measurements are taken.

Figure 6.2 shows the power consumption when Turbo Boost is disabled. This aligns with

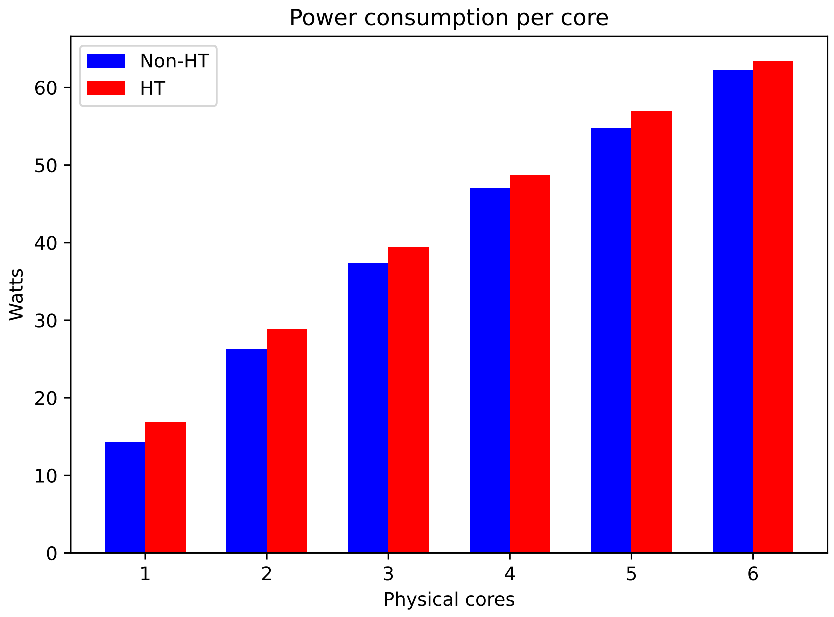

Figure 6.2: Power consumption with Turbo Boost disabled.

our expectations, as with each additional active physical core, the power consumption increases in a more-or-less linear fashion.

Chapter 7

Conclusion

In this project, we attempted to develop a set of tools that could be used to estimate the power consumption of individual processes under Linux.
We have created a configurable profiling tool that is able to measure the power usage of a CPU using several pre-defined benchmarks. This data can be used to determine the CPU's characteristics, such as the static power, and the hyperthreading ratio.
We have also created an utility program for tracking the CPU time used by all running processes and threads. This allows us to calculate the CPU usage of each process. It also tracks supplemental information, such as the logical processor on which the process or thread was last executed, and the state it was in.
Finally, we wrote a number of scripts that are used to process the data collected by the other two components. This processing results in three possible estimates for process-level power consumption:

1. A simple naive estimation, that only takes into account the CPU percentage used by the process,

2. A naive hyperthreading that takes into account the time a logical core was active, and

3. A hyperthread-aware estimation, that attempts to take into account hyperthreading, and its effects on power consumption.

While the results of the naive estimation work as expected (e.g. a process using 50% of the CPU will be attributed 50% of the energy used), the results from the hyperthread-aware estimations are more erratic. Sometimes they are able to produce estimates that align with our expectations (although we have no basis for comparison), but other times the estimates are obviously wrong (e.g. estimates a higher power usage that the total amount consumed).

We have also encountered a phenomenon whereby disabling hyperthreading slightly, but consistently lowered the power consumption, even compared to similar tests ran while hyperthreading is enabled.

To answer RQ1, we were not able to create a tool that is can take into account hyperthreading, and produce reliable estimates. This also answers RQ1A.

However, we have achieved several of our expected contributions. The profiling tool and the process tracker are able to collect usable data. While unfortunately our algorithm for hyperthread-aware estimations is not reliable, we believe that with further refinement it could become an accurate tool for researchers and developers.

7.1 Future Works

7.1.1 UEFI Profiler

While some work was done on a bare-metal UEFI version of the profiler, we decided to focus our efforts on the Linux version. However, we believe that it would still be beneficial to create it, as we expect it to provide a more accurate representation of the CPU's power consumption. Further work would need to be done to align the bare-metal performance of the CPU with Linux' performance of the CPU.

7.1.2 Linux Scheduler

Our original plans included a modification of the Linux scheduler. The aim of the modification would have been to make the scheduler keep track of how long, and on which processor, a given process was scheduled, and to expose this information in a manner that is easy to access (e.g. a solution similar to */proc*).
Depending on how it is implemented it could either work in tandem with the process tracker (Section 4.2) to provide more accurate and comprehensive information, or it could replace it entirely.
While plans for this component never got beyond the discussion phase, we think revisiting it in the future may be worthwhile, as it has the potential of greatly simplifying the way we collect CPU usage information.

7.1.3 Profiler Improvements

There are some additional improvements to the profiler that may be able to enhance its usability.
The first of such improvements is a better method of calculating the number of samples required for a given confidence interval and margin of error. The current implementation uses a formula that makes some assumptions, that may not always be correct. For example, it assumes that the samples are normally distributed. A more sophisticated solution, such as the one used in [5], would probably yield better results.
To use the same formula as in [5], we would have to estimate the standard deviation of the population. This could be done by first running a limited number of "throwaway" tests that are only used to calculate the standard deviation. The result can then be used to compute the number of samples needed for the specified confidence interval and margin of error.

Running these throwaway tests could also double as warmup. Running warmups is considered a good practice when benchmarking software [23]. It allows all initializations to run, so that they will not impact the measurements.

7.1.4 Linux Process Accounting

Our method to acquire the process tracking data is crude, but not uncommon. We poll the currently running processes at intervals of approximately 100 ms, then read updated values of their CPU time and the RAPL power measurement at that interval. While polling for RAPL power measurement is not something we can avoid, there may be a better alternative to process tracking.

Our primary issue is the imprecision of our data. We do not know when a process switches running cores, only that it does during a given interval. Additionally,we do not know when hyperthreads overlap, in the sense of both running on a single core at the same time. We make some assumptions in these cases that could lead to inaccuracies.

A kernel-based solution would be better, as the kernel has access to all the data we need. Linux Process Accounting is a kernel subsystem that prints process data as processes start and end. This subsystem could be used to push data, rather than requring us to pull it, and may need to be extended to have the data we need.

Bibliography

[1] *7-Zip Compression*. URL: https://openbenchmarking.org/test/pts/compress-7zip-1.10.0 (visited on 05/21/2023).

[2] Sarah Abdulsalam et al. "Program energy efficiency: The impact of language, compiler and implementation choices". In: *International Green Computing Conference*. 2014, pp. 1–6. DOI: 10.1109/IGCC.2014.7039169.

[3] *Benchmark Test Profiles - OpenBenchmarking.org*. URL: https://openbenchmarking.org/tests (visited on 06/06/2023).

[4] Coral Calero, Macario Polo, and Mª Ángeles Moraga. "Investigating the impact on execution time and energy consumption of developing with Spring". In: *Sustainable Computing: Informatics and Systems* 32 (2021), p. 100603. ISSN: 2210-5379. DOI: https://doi.org/10.1016/j.suscom.2021.100603. URL: https://www.sciencedirect.com/science/article/pii/S2210537921000913.

[5] Casper Susgaard Nielsen et al. *The Influence of Programming Paradigms on Energy Consumption*. English. Tech. rep. Aalborg University, Jan. 2021, p. 109. URL: https://projekter.aau.dk/projekter/en/studentthesis/the-influence-of-programming-paradigms-on-energy-consumption(145e96f3-b48b-443b-9b74-39c7e4aa9a85).html (visited on 06/04/2023).

[6] *Computer Language Benchmarks Game - fannkuch-redux C gcc #3 program*. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/program/fannkuchredux-gcc-3.html (visited on 03/30/2023).

[7] *Determining sample size based on confidence and margin of error (video)*. en. URL: https://www.khanacademy.org/math/ap-statistics/xfb5d8e68:inference-categorical-proportions/one-sample-z-interval-proportion/v/determining-sample-size-based-on-confidence-and-margin-of-error (visited on 05/24/2023).

[8] *fannkuch-redux C gcc #3 program (Benchmarks Game)*. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/program/fannkuchredux-gcc-3.html (visited on 04/19/2023).

[9] *fannkuch-redux description (Benchmarks Game)*. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/fannkuchredux.html (visited on 04/19/2023).

[10] Hubblo. *Scaphandre*. https://github.com/hubblo-org/scaphandre. 2023.

[11] *IA-PC HPET (High Precision Event Timers) Specification*. URL: https://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/software-developers-hpet-spec-1-0a.pdf (visited on 05/27/2023).

[12] *Intel(r) Power Gadget for Linux*. en. URL: https://github.com/vitillo/power_gadget (visited on 04/05/2023).

[13] *Intel® 64 and IA-32 Architectures Software Developer Manuals*. en. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html (visited on 03/08/2023).

[14] *Intel® Pentium® II Xeon® Processor 400 MHz, 512K Cache, 100 MHz FSB Product Specifications*. en. URL: https://www.intel.com/content/www/us/en/products/sku/49943/intel-pentium-ii-xeon-processor-400-mhz-512k-cache-100-mhz-fsb.html (visited on 05/30/2023).

[15] *Intel® Power Gadget*. en. URL: https://www.intel.com/content/www/us/en/developer/articles/tool/power-gadget.html (visited on 04/05/2023).

[16] *Intel® Turbo Boost 2.0: High Performance Intel Turbo Boost Technology...* en. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html (visited on 06/04/2023).

[17] *Intel® Xeon® W-1390T Processor (16M Cache, up to 4.90 GHz) Product Specifications*. en. URL: https://www.intel.com/content/www/us/en/products/sku/212269/intel-xeon-w1390t-processor-16m-cache-up-to-4-90-ghz.html (visited on 05/30/2023).

[18] Nicola Jones. "How to stop data centres from gobbling up the world's electricity". en. In: *Nature* 561.7722 (Sept. 2018), pp. 163–166. DOI: 10.1038/d41586-018-06610-y. URL: https://www.nature.com/articles/d41586-018-06610-y (visited on 05/30/2023).

[19] Kashif Nizam Khan et al. "RAPL in Action: Experiences in Using RAPL for Power Measurements". en. In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* 3.2 (June 2018), pp. 1–26. ISSN: 2376-3639, 2376-3647. DOI: 10.1145/3177754. URL: https://dl.acm.org/doi/10.1145/3177754 (visited on 03/01/2023).

[20] Jóakim von Kistowski et al. "Measuring and Benchmarking Power Consumption and Energy Efficiency". In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE '18. New York, NY, USA: Association for Computing Machinery, Apr. 2018, pp. 57–65. ISBN: 9781450356299. DOI: 10.1145/3185768.3185775. URL: https://doi.org/10.1145/3185768.3185775 (visited on 06/07/2023).

[21] Lukas Koedijk and Ana Oprescu. "Finding Significant Differences in the Energy Consumption when Comparing Programming Languages and Programs". In: *2022 International Conference on ICT for Sustainability (ICT4S)*. 2022, pp. 1–12. DOI: 10.1109/ICT4S55073.2022.00012.

[22] Mario Linares-Vásquez et al. “Mining Energy-Greedy API Usage Patterns in Android Apps: An Empirical Study”. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 2–11. ISBN: 9781450328630. DOI: 10.1145/2597073.2597085. URL: https://doi.org/10.1145/2597073.2597085.

[23] *MicroBenchmarks - MicroBenchmarks - OpenJDK Wiki*. URL: https://wiki.openjdk.org/display/HotSpot/MicroBenchmarks (visited on 06/04/2023).

[24] *msr(4) - Linux manual page*. URL: https://man7.org/linux/man-pages/man4/msr.4.html (visited on 03/22/2023).

[25] Rui Pereira et al. “Energy Efficiency across Programming Languages: How Do Energy, Time, and Memory Relate?” In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2017. Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 256–267. ISBN: 9781450355254. DOI: 10.1145/3136014.3136031. URL: https://doi.org/10.1145/3136014.3136031.

[26] Rui Pereira et al. “Energy Efficiency across Programming Languages: How Do Energy, Time, and Memory Relate?” In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2017. Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 256–267. ISBN: 9781450355254. DOI: 10.1145/3136014.3136031. URL: https://doi.org/10.1145/3136014.3136031.

[27] *Phoronix Test Suite*. URL: https://www.phoronix-test-suite.com/ (visited on 05/17/2023).

[28] *POSIX-UEFI*. URL: https://gitlab.com/bztsrc/posix-uefi (visited on 03/30/2023).

[29] *Power Capping Framework — The Linux Kernel documentation*. URL: https://www.kernel.org/doc/html/next/power/powercap/powercap.html (visited on 03/08/2023).

[30] *proc(5) - Linux manual page*. URL: https://man7.org/linux/man-pages/man5/proc.5.html (visited on 03/15/2023).

[31] Ajitha Rajan, Adel Noureddine, and Panagiotis Stratis. “A Study on the Influence of Software and Hardware Features on Program Energy”. en. In: *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. Ciudad Real Spain: ACM, Sept. 2016, pp. 1–10. ISBN: 9781450344272. DOI: 10.1145/2961111.2962593. URL: https://dl.acm.org/doi/10.1145/2961111.2962593 (visited on 06/07/2023).

[32] Peter Shirley. *Ray Tracing in One Weekend*. Dec. 2020. URL: https://raytracing.github.io/books/RayTracingInOneWeekend.html (visited on 04/19/2023).

[33] *Software-artifact Infrastructure Repository: Home*. URL: https://sir.csc.ncsu.edu/portal/index.php (visited on 06/07/2023).

[34] *SPEC Benchmarks and Tools*. URL: https://www.spec.org/benchmarks.html (visited on 06/07/2023).

[35] *stress*. URL: https://linux.die.net/man/1/stress (visited on 05/21/2023).

[36] *Timed Linux Kernel Compilation*. URL: https://openbenchmarking.org/test/pts/build-linux-kernel-1.15.0 (visited on 05/21/2023).

[37] *UEFI Specification 2.10*. URL: `https://uefi.org/specs/UEFI/2.10/` (visited on 03/30/2023).

[38] *UEFI Specification 2.10 - 7.5.2 EFI_BOOT_SERVICES.Stall()*. URL: `https://uefi.org/specs/UEFI/2.10/07_Services_Boot_Services.html#efi-boot-services-stall` (visited on 03/30/2023).

[39] *Watts Up Pro Portable Power Meter*. URL: `https://www.powermeterstore.com/p1206/watts_up_pro.php` (visited on 05/30/2023).

[40] *Which programming language is fastest? (Benchmarks Game)*. URL: `https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html` (visited on 06/06/2023).

[41] *x265*. URL: `https://openbenchmarking.org/test/pts/x265` (visited on 05/21/2023).

[42] Yan Zhai et al. "HaPPy: Hyperthread-aware Power Profiling Dynamically". In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, June 2014, pp. 211–217. ISBN: 978-1-931971-10-2. URL: `https://www.usenix.org/conference/atc14/technical-sessions/presentation/zhai`.

Appendix A

Exact Specifications

A.1 System Specification

```
testpc
    description: Desktop Computer
    product: 30DH00HKMT (LENOVO_MT_30DH_BU_Think_FM_ThinkStation P340)
    vendor: LENOVO
    version: ThinkStation P340
    serial: PC1QBC66
    width: 64 bits
    capabilities: smbios-3.2.0 dmi-3.2.0 smp vsyscall32
    configuration: administrator_password=disabled boot=normal chassis=desktop family=
        ↪ ThinkStation P340 keyboard_password=enabled power-on_password=disabled sku=
        ↪ LENOVO_MT_30DH_BU_Think_FM_ThinkStation P340 uuid=964A9692-1035-EB11-922E-
        ↪ ECE5418C3D00
  *-core
       description: Motherboard
       product: 1048
       vendor: LENOVO
       physical id: 0
       version: SDK0Q40104 WIN 3915000358763
       slot: Default string
     *-firmware
          description: BIOS
          vendor: LENOVO
          physical id: 0
          version: S08KT32A
          date: 10/14/2020
          size: 64KiB
          capacity: 16MiB
          capabilities: pci upgrade shadowing cdboot bootselect socketedrom edd
              ↪ int13floppy1200 int13floppy720 int13floppy2880 int5printscreen int9keyboard
              ↪ int14serial int17printer acpi usb biosbootspecification uefi
     *-memory
          description: System Memory
          physical id: 3c
          slot: System board or motherboard
          size: 16GiB
          capabilities: ecc
          configuration: errordetection=ecc
        *-bank:0
             description: [empty]
             physical id: 0
             slot: ChannelA-DIMM0
        *-bank:1
             description: DIMM DDR4 Synchronous 3200 MHz (0,3 ns)
             product: HMA82GU7DJR8N-XN
             vendor: SK Hynix
             physical id: 1
             serial: 73F9615A
             slot: ChannelA-DIMM1
             size: 16GiB
             width: 64 bits
             clock: 3200MHz (0.3ns)
        *-bank:2
             description: [empty]
             physical id: 2
             slot: ChannelB-DIMM0
        *-bank:3
             description: [empty]
             physical id: 3
             slot: ChannelB-DIMM1
     *-cache:0
          description: L1 cache
          physical id: 4f
          slot: L1 Cache
          size: 384KiB
          capacity: 384KiB
          capabilities: synchronous internal write-back unified
          configuration: level=1
     *-cache:1
          description: L2 cache
          physical id: 50
          slot: L2 Cache
          size: 1536KiB
          capacity: 1536KiB
          capabilities: synchronous internal write-back unified
          configuration: level=2
     *-cache:2
          description: L3 cache
          physical id: 51
          slot: L3 Cache
          size: 12MiB
          capacity: 12MiB
          capabilities: synchronous internal write-back unified
          configuration: level=3
     *-cpu
          description: CPU
          product: Intel(R) Xeon(R) W-1250P CPU @ 4.10GHz
          vendor: Intel Corp.
          physical id: 52
          bus info: cpu@0
          version: Intel(R) Xeon(R) W-1250P CPU @ 4.10GHz
          serial: To Be Filled By O.E.M.
          slot: U3E1
          size: 1100MHz
          capacity: 4100MHz
          width: 64 bits
          clock: 100MHz
          capabilities: lm fpu fpu_exception wp vme de pse tsc msr pae mce cx8 apic sep mtrr
               ↪ pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe
               ↪ syscall nx pdpe1gb rdtscp x86-64 constant_tsc art arch_perfmon pebs bts
               ↪ rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64
               ↪ monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1
               ↪ sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand
               ↪ lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single ssbd ibrs ibpb
               ↪ stibp ibrs_enhanced tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase
               ↪ tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap clflushopt
               ↪ intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm arat pln pts hwp hwp_notify
             ↪ hwp_act_window hwp_epp pku ospke md_clear flush_l1d arch_capabilities
             ↪ cpufreq
          configuration: cores=6 enabledcores=6 threads=12
     *-pci
          description: Host bridge
          product: Comet Lake-S 6c Host Bridge/DRAM Controller
          vendor: Intel Corporation
          physical id: 100
          bus info: pci@0000:00:00.0
          version: 05
          width: 32 bits
          clock: 33MHz
          configuration: driver=skl_uncore
          resources: irq:0
        *-display
             description: VGA compatible controller
             product: Comet Lake-S GT2 [UHD Graphics P630]
             vendor: Intel Corporation
             physical id: 2
             bus info: pci@0000:00:02.0
             version: 05
             width: 64 bits
             clock: 33MHz
             capabilities: pciexpress msi pm vga_controller bus_master cap_list rom
             configuration: driver=i915 latency=0
             resources: irq:138 memory:b0000000-b0ffffff memory:a0000000-afffffff ioport
                 ↪ :3000(size=64) memory:c0000-dffff
        *-generic:0 UNCLAIMED
             description: System peripheral
             product: Xeon E3-1200 v5/v6 / E3-1500 v5 / 6th/7th/8th Gen Core Processor
                 ↪ Gaussian Mixture Model
             vendor: Intel Corporation
             physical id: 8
             bus info: pci@0000:00:08.0
             version: 00
             width: 64 bits
             clock: 33MHz
             capabilities: msi pm cap_list
             configuration: latency=0
             resources: memory:b123f000-b123ffff
        *-generic:1
             description: Signal processing controller
             product: Comet Lake PCH Thermal Controller
             vendor: Intel Corporation
             physical id: 12
             bus info: pci@0000:00:12.0
             version: 00
             width: 64 bits
             clock: 33MHz
             capabilities: pm msi cap_list
             configuration: driver=intel_pch_thermal latency=0
             resources: irq:16 memory:b123e000-b123efff
        *-usb
             description: USB controller
             product: Comet Lake USB 3.1 xHCI Host Controller
             vendor: Intel Corporation
             physical id: 14
             bus info: pci@0000:00:14.0
             version: 00
             width: 64 bits
             clock: 33MHz
             capabilities: pm msi xhci bus_master cap_list
             configuration: driver=xhci_hcd latency=0
             resources: irq:124 memory:b1220000-b122ffff
           *-usbhost:0
                product: xHCI Host Controller
             vendor: Linux 6.3.4-arch1-1 xhci-hcd
             physical id: 0
             bus info: usb@1
             logical name: usb1
             version: 6.03
             capabilities: usb-2.00
             configuration: driver=hub slots=16 speed=480Mbit/s
           *-usb:0
                description: Generic USB device
                product: 802.11n WLAN Adapter
                vendor: Realtek
                physical id: 1
                bus info: usb@1:1
                version: 2.00
                serial: 00e04c000001
                capabilities: usb-2.00
                configuration: driver=rtl8192cu maxpower=500mA speed=480Mbit/s
           *-usb:1
                description: Keyboard
                product: DELL USB Keyboard
                vendor: DELL
                physical id: 6
                bus info: usb@1:6
                version: 1.05
                capabilities: usb-1.10
                configuration: driver=usbhid maxpower=100mA speed=1Mbit/s
           *-usb:2
                description: Mass storage device
                product: Cruzer Fit
                vendor: SanDisk
                physical id: 8
                bus info: usb@1:8
                version: 1.00
                serial: 4C530001140701111152
                capabilities: usb-2.10 scsi
                configuration: driver=usb-storage maxpower=224mA speed=480Mbit/s
           *-usb:3
                description: MMC Host
                product: USB2.0-CRW
                vendor: Generic
                physical id: 9
                bus info: usb@1:9
                logical name: mmc0
                version: 39.60
                serial: 20100201396000000
                capabilities: usb-2.00
                configuration: driver=rtsx_usb maxpower=500mA speed=480Mbit/s
        *-usbhost:1
             product: xHCI Host Controller
             vendor: Linux 6.3.4-arch1-1 xhci-hcd
             physical id: 1
             bus info: usb@2
             logical name: usb2
             version: 6.03
             capabilities: usb-3.10
             configuration: driver=hub slots=10 speed=10000Mbit/s
     *-memory UNCLAIMED
          description: RAM memory
          product: Comet Lake PCH Shared SRAM
          vendor: Intel Corporation
          physical id: 14.2
          bus info: pci@0000:00:14.2
          version: 00
          width: 64 bits
          clock: 33MHz (30.3ns)
          capabilities: pm cap_list
```

```
          configuration: latency=0
          resources: memory:b1236000-b1237fff memory:b123d000-b123dfff
     *-communication:0
          description: Communication controller
          product: Comet Lake HECI Controller
          vendor: Intel Corporation
          physical id: 16
          bus info: pci@0000:00:16.0
          version: 00
          width: 64 bits
          clock: 33MHz
          capabilities: pm msi bus_master cap_list
          configuration: driver=mei_me latency=0
          resources: irq:140 memory:b123c000-b123cfff
     *-communication:1
          description: Serial controller
          product: Comet Lake Keyboard and Text (KT) Redirection
          vendor: Intel Corporation
          physical id: 16.3
          bus info: pci@0000:00:16.3
          version: 00
          width: 32 bits
          clock: 66MHz
          capabilities: msi pm 16550 cap_list
          configuration: driver=serial latency=0
          resources: irq:19 ioport:30a0(size=8) memory:b123b000-b123bfff
     *-sata
          description: SATA controller
          product: Comet Lake SATA AHCI Controller
          vendor: Intel Corporation
          physical id: 17
          bus info: pci@0000:00:17.0
          version: 00
          width: 32 bits
          clock: 66MHz
          capabilities: sata msi pm ahci_1.0 bus_master cap_list
          configuration: driver=ahci latency=0
          resources: irq:123 memory:b1234000-b1235fff memory:b123a000-b123a0ff ioport
              ↪ :3090(size=8) ioport:3080(size=4) ioport:3060(size=32) memory:b1239000-
              ↪ b12397ff
     *-pci
          description: PCI bridge
          product: Comet Lake PCI Express Root Port #21
          vendor: Intel Corporation
          physical id: 1b
          bus info: pci@0000:00:1b.0
          version: f0
          width: 32 bits
          clock: 33MHz
          capabilities: pci pciexpress msi pm normal_decode bus_master cap_list
          configuration: driver=pcieport
          resources: irq:122 memory:b1100000-b11fffff
        *-nvme
             description: NVMe device
             product: SAMSUNG MZVLB512HBJQ-000L7
             vendor: Samsung Electronics Co Ltd
             physical id: 0
             bus info: pci@0000:01:00.0
             logical name: /dev/nvme0
             version: 4M2QEXF7
             serial: S4ENNX1NB02597
             width: 64 bits
             clock: 33MHz
             capabilities: nvme pm msi pciexpress msix nvm_express bus_master cap_list
             configuration: driver=nvme latency=0 nqn=nqn.2014.08.org.nvmexpress:144
                 ↪ d144dS4ENNX1NB02597      SAMSUNG MZVLB512HBJQ-000L7 state=live
        resources: irq:16 memory:b1100000-b1103fff
      *-namespace:0
           description: NVMe disk
           physical id: 0
           logical name: hwmon1
      *-namespace:1
           description: NVMe disk
           physical id: 2
           logical name: /dev/ng0n1
      *-namespace:2
           description: NVMe disk
           physical id: 1
           bus info: nvme@0:1
           logical name: /dev/nvme0n1
           size: 476GiB (512GB)
           capabilities: gpt-1.00 partitioned partitioned:gpt
           configuration: guid=ce5650e2-a211-4359-8999-5e2cf8c302dc logicalsectorsize
               ↪ =512 sectorsize=512 wwid=eui.0025388b01b7d135
         *-volume:0
              description: Windows FAT volume
              vendor: mkfs.fat
              physical id: 1
              bus info: nvme@0:1,1
              logical name: /dev/nvme0n1p1
              logical name: /boot
              version: FAT32
              serial: 81f3-8a30
              size: 1022MiB
              capacity: 1023MiB
              capabilities: boot fat initialized
              configuration: FATs=2 filesystem=fat mount.fstype=vfat mount.options=rw
                  ↪ ,relatime,fmask=0022,dmask=0022,codepage=437,iocharset=ascii,
                  ↪ shortname=mixed,utf8,errors=remount-ro name=EFI system partition
                  ↪  state=mounted
         *-volume:1
              description: EXT4 volume
              vendor: Linux
              physical id: 2
              bus info: nvme@0:1,2
              logical name: /dev/nvme0n1p2
              logical name: /
              version: 1.0
              serial: 1abeb985-7b3b-4b51-b935-2648174719e3
              size: 459GiB
              capacity: 459GiB
              capabilities: journaled extended_attributes large_files huge_files
                  ↪ dir_nlink recover 64bit extents ext4 ext2 initialized
              configuration: created=2023-02-14 13:01:42 filesystem=ext4
                  ↪ lastmountpoint=/ modified=2023-05-30 10:28:44 mount.fstype=ext4
                  ↪ mount.options=rw,relatime mounted=2023-05-30 10:28:44 name=Linux
                  ↪  filesystem state=mounted
         *-volume:2
              description: Linux swap volume
              vendor: Linux
              physical id: 3
              bus info: nvme@0:1,3
              logical name: /dev/nvme0n1p3
              version: 1
              serial: a447901a-69d4-4378-8ef8-690dce8badf4
              size: 15GiB
              capacity: 15GiB
              capabilities: nofs swap initialized
              configuration: filesystem=swap name=Linux swap pagesize=4096
*-isa
     description: ISA bridge
     product: Intel Corporation
```

```
        vendor: Intel Corporation
        physical id: 1f
        bus info: pci@0000:00:1f.0
        version: 00
        width: 32 bits
        clock: 33MHz
        capabilities: isa bus_master
        configuration: latency=0
   *-multimedia
        description: Audio device
        product: Comet Lake PCH cAVS
        vendor: Intel Corporation
        physical id: 1f.3
        bus info: pci@0000:00:1f.3
        version: 00
        width: 64 bits
        clock: 33MHz
        capabilities: pm msi bus_master cap_list
        configuration: driver=snd_hda_intel latency=32
        resources: irq:141 memory:b1230000-b1233fff memory:b1000000-b10fffff
   *-serial:0
        description: SMBus
        product: Comet Lake PCH SMBus Controller
        vendor: Intel Corporation
        physical id: 1f.4
        bus info: pci@0000:00:1f.4
        version: 00
        width: 64 bits
        clock: 33MHz
        configuration: driver=i801_smbus latency=0
        resources: irq:16 memory:b1238000-b12380ff ioport:efa0(size=32)
   *-serial:1
        description: Serial bus controller
        product: Comet Lake PCH SPI Controller
        vendor: Intel Corporation
        physical id: 1f.5
        bus info: pci@0000:00:1f.5
        version: 00
        width: 32 bits
        clock: 33MHz
        configuration: driver=intel-spi latency=0
        resources: irq:0 memory:fe010000-fe010fff
   *-network
        description: Ethernet interface
        product: Ethernet Connection (11) I219-LM
        vendor: Intel Corporation
        physical id: 1f.6
        bus info: pci@0000:00:1f.6
        logical name: eno1
        version: 00
        serial: 2c:f0:5d:4e:dd:e9
        size: 1Gbit/s
        capacity: 1Gbit/s
        width: 32 bits
        clock: 33MHz
        capabilities: pm msi bus_master cap_list ethernet physical tp 10bt 10bt-fd 100bt
            ↪  100bt-fd 1000bt-fd autonegotiation
        configuration: autonegotiation=on broadcast=yes driver=e1000e driverversion
            ↪ =6.3.4-arch1-1 duplex=full firmware=0.4-4 ip=172.28.210.106 latency=0
            ↪ link=yes multicast=yes port=twisted pair speed=1Gbit/s
        resources: irq:139 memory:b1200000-b121ffff
*-pnp00:00
     product: Motherboard registers
     physical id: 1
     capabilities: pnp
     configuration: driver=system
   *-pnp00:01
        product: Motherboard registers
        physical id: 2
        capabilities: pnp
        configuration: driver=system
   *-pnp00:02
        product: 16550A-compatible COM port
        physical id: 3
        capabilities: pnp
        configuration: driver=serial
   *-pnp00:03
        product: Motherboard registers
        physical id: 4
        capabilities: pnp
        configuration: driver=system
   *-pnp00:04
        product: PnP device INT3f0d
        vendor: Interphase Corporation
        physical id: 5
        capabilities: pnp
        configuration: driver=system
   *-pnp00:05
        product: Motherboard registers
        physical id: 6
        capabilities: pnp
        configuration: driver=system
   *-pnp00:06
        product: Motherboard registers
        physical id: 7
        capabilities: pnp
        configuration: driver=system
   *-pnp00:07
        product: Motherboard registers
        physical id: 8
        capabilities: pnp
        configuration: driver=system
   *-pnp00:08
        product: Motherboard registers
        physical id: 9
        capabilities: pnp
        configuration: driver=system
*-power UNCLAIMED
     description: To Be Filled By O.E.M.
     product: To Be Filled By O.E.M.
     vendor: To Be Filled By O.E.M.
     physical id: 1
     version: To Be Filled By O.E.M.
     serial: To Be Filled By O.E.M.
     capacity: 32768mWh
*-scsi
     physical id: 2
     bus info: scsi@4
     logical name: scsi4
     capabilities: scsi-host
     configuration: driver=usb-storage
*-network DISABLED
     description: Wireless interface
     physical id: 3
     bus info: usb@1:1
     logical name: wlp0s20f0u1
     serial: 08:be:ac:0a:7c:da
     capabilities: ethernet physical wireless
     configuration: broadcast=yes driver=rtl8192cu driverversion=6.3.4-arch1-1 firmware=N/A
         ↪  link=no multicast=yes wireless=IEEE 802.11
```

A.2 Installed Packages

```
acl 2.3.1-3
adobe-source-code-pro-fonts 2.042u+1.062i+1.026vf-1
adwaita-cursors 44.0-1
adwaita-icon-theme 44.0-1
alsa-lib 1.2.9-1
alsa-topology-conf 1.2.5.1-3
alsa-ucm-conf 1.2.9-1
aom 3.6.1-1
archlinux-keyring 20230504-1
argon2 20190702-5
at-spi2-core 2.48.3-1
attr 2.5.1-3
audit 3.1.1-1
autoconf 2.71-4
automake 1.16.5-2
avahi 0.8+22+gfd482a7-4
base 3-1
base-devel 1-1
bash 5.1.016-4
bc 1.07.1-4
binutils 2.40-6
bison 3.8.2-5
boost-libs 1.81.0-6
brotli 1.0.9-12
bzip2 1.0.8-5
ca-certificates 20220905-1
ca-certificates-mozilla 3.89.1-1
ca-certificates-utils 20220905-1
cairo 1.17.8-2
cantarell-fonts 1:0.303.1-1
clang 15.0.7-9
cmake 3.26.4-1
compiler-rt 15.0.7-2
confuse 3.3-3
coreutils 9.3-1
cpupower 6.3-2
cryptsetup 2.6.1-3
curl 8.1.1-2
dav1d 1.2.0-1
db5.3 5.3.28-2
dbus 1.14.6-2
dbus-glib 0.112-2
dbus-python 1.2.18-5
dconf 0.40.0-2
debugedit 5.0-5
debuginfod 0.189-1
default-cursors 2-1
desktop-file-utils 0.26-2
device-mapper 2.03.21-1
dhcpcd 10.0.1-1
diffutils 3.10-1
dmenu 5.2-1
dnssec-anchors 20190629-3
duktape 2.7.0-5
e2fsprogs 1.47.0-1
edk2-shell 202302-1
elfutils 0.189-1
expat 2.5.0-1
fakeroot 1.31-2
ffmpeg 2:6.0-8
file 5.44-3
filesystem 2023.01.31-1
```

```
findutils 4.9.0-3
firefox 113.0.2-1
flac 1.4.2-1
flex 2.6.4-5
fontconfig 2:2.14.2-1
freetype2 2.13.0-1
fribidi 1.0.13-1
fuse-common 3.14.1-1
fuse2 2.9.9-4
gawk 5.2.2-1
gc 8.2.2-1
gcc 13.1.1-1
gcc-libs 13.1.1-1
gdb 13.1-3
gdb-common 13.1-3
gdbm 1.23-2
gdk-pixbuf2 2.42.10-2
gettext 0.21.1-5
giflib 5.2.1-2
git 2.40.1-1
glib-networking 1:2.76.0-1
glib2 2.76.3-1
glibc 2.37-3
gmp 6.2.1-2
gnu-free-fonts 20120503-8
gnupg 2.2.41-1
gnutls 3.8.0-1
go 2:1.20.4-2
gobject-introspection-runtime 1.76.1-3
gperftools 2.10-1
gpgme 1.20.0-3
graphite 1:1.3.14-3
grep 3.11-1
grml-zsh-config 0.19.5-1
groff 1.22.4-10
gsettings-desktop-schemas 44.0-1
gsm 1.0.22-1
gtk-update-icon-cache 1:4.10.3-3
gtk3 1:3.24.38-1
guile 3.0.9-1
gzip 1.12-2
harfbuzz 7.3.0-1
hicolor-icon-theme 0.17-3
hidapi 0.14.0-1
highway 1.0.4-1
http-parser 2.9.4-1
hwdata 0.370-1
i3-wm 4.22-4
i3blocks 1.5-3
i3lock 2.14.1-1
i3status 2.14-1
iana-etc 20230405-1
icu 72.1-2
imath 3.1.8-1
intel-ucode 20230516.a-1
iproute2 6.3.0-2
iptables 1:1.8.9-1
iputils 20221126-2
iso-codes 4.15.0-1
jack2 1.9.22-1
jansson 2.14-2
json-c 0.16-1
json-glib 1.6.6-2
jsoncpp 1.9.5-2
kbd 2.5.1-2
keyutils 1.6.3-2
```

```
kmod 30-3
krb5 1.20.1-1
l-smash 2.14.5-3
lame 3.100-4
lcms2 2.15-1
ldns 1.8.3-2
less 1:633-1
libarchive 3.6.2-2
libass 0.17.1-1
libassuan 2.5.5-2
libasyncns 1:0.8+r3+g68cd5af-2
libavc1394 0.5.4-5
libbluray 1.3.4-1
libbpf 1.2.0-1
libbs2b 3.1.0-8
libcap 2.69-1
libcap-ng 0.8.3-2
libcloudproviders 0.3.1+r8+g3a229ee-1
libcolord 1.4.6-1
libcups 1:2.4.2-7
libdaemon 0.14-5
libdatrie 0.2.13-2
libdecor 0.1.1-2
libdrm 2.4.115-1
libedit 20221030_3.1-1
libelf 0.189-1
libepoxy 1.5.10-2
libev 4.33-2
libevdev 1.13.1-1
libevent 2.1.12-4
libffi 3.4.4-1
libfontenc 1.1.7-1
libgcrypt 1.10.2-1
libgirepository 1.76.1-3
libgit2 1:1.6.4-1
libglvnd 1.6.0-1
libgpg-error 1.47-1
libgudev 237-2
libice 1.1.1-2
libidn2 2.3.4-3
libiec61883 1.2.0-7
libinput 1.23.0-1
libisl 0.26-1
libjpeg-turbo 2.1.5.1-1
libjxl 0.8.1-2
libkeybinder3 0.3.2-4
libksba 1.6.3-1
libldap 2.6.4-2
libmfx 23.2.2-1
libmicrohttpd 0.9.76-1
libmnl 1.0.5-1
libmodplug 0.8.9.0-5
libmpc 1.3.1-1
libnetfilter_conntrack 1.0.9-1
libnfnetlink 1.0.2-1
libnftnl 1.2.5-1
libnghttp2 1.53.0-1
libnl 3.7.0-3
libnotify 0.8.2-1
libnsl 2.0.0-3
libogg 1.3.5-1
libomxil-bellagio 0.9.3-4
libopenmpt 0.7.1-1
libp11-kit 0.24.1-1
libpcap 1.10.4-1
libpciaccess 0.17-1
```

```
libpng 1.6.39-1
libproxy 0.4.18-3
libpsl 0.21.2-1
libpulse 16.1-6
libraw1394 2.1.2-3
librsvg 2:2.56.0-1
libsamplerate 0.2.2-2
libsasl 2.1.28-4
libseccomp 2.5.4-2
libsecret 0.20.5-2
libsm 1.2.4-1
libsndfile 1.2.0-1
libsoup3 3.4.2-1
libsoxr 0.1.3-3
libssh 0.10.5-1
libssh2 1.10.0-3
libstemmer 2.2.0-2
libsysprof-capture 3.48.0-2
libtasn1 4.19.0-1
libthai 0.1.29-2
libtheora 1.1.1-5
libtiff 4.5.0-4
libtirpc 1.3.3-2
libtool 2.4.7+4+g1ec8fa28-3
libtraceevent 1:1.7.2-1
libunistring 1.1-2
libunwind 1.6.2-2
libusb 1.0.26-2
libutempter 1.2.1-3
libuv 1.44.2-1
libva 2.18.0-1
libvdpau 1.5-1
libverto 0.3.2-4
libvorbis 1.3.7-3
libvpx 1.13.0-1
libwacom 2.7.0-1
libwebp 1.3.0-3
libx11 1.8.4-1
libxau 1.0.11-2
libxaw 1.0.15-1
libxcb 1.15-2
libxcomposite 0.4.6-1
libxcrypt 4.4.33-1
libxcursor 1.2.1-3
libxcvt 0.1.2-1
libxdamage 1.1.6-1
libxdmcp 1.1.4-2
libxext 1.3.5-1
libxfixes 6.0.1-1
libxfont2 2.0.6-2
libxft 2.3.8-1
libxi 1.8.1-1
libxinerama 1.1.5-1
libxkbcommon 1.5.0-1
libxkbcommon-x11 1.5.0-1
libxkbfile 1.1.2-1
libxml2 2.10.4-4
libxmu 1.1.4-1
libxpm 3.5.16-1
libxrandr 1.5.3-1
libxrender 0.9.11-1
libxshmfence 1.3.2-1
libxt 1.3.0-1
libxtst 1.2.4-1
libxv 1.0.12-1
libxxf86vm 1.1.5-1
```

```
libzip 1.9.2-1
licenses 20220125-2
linux 6.3.4.arch1-1
linux-api-headers 6.3-1
linux-firmware 20230404.2e92a49f-1
linux-firmware-whence 20230404.2e92a49f-1
lld 15.0.7-2
llvm-libs 15.0.7-3
lm_sensors 1:3.6.0.r41.g31d1f125-2
lshw B.02.19.2-5
luit 20230201-1
lz4 1:1.9.4-1
lzo 2.10-5
m4 1.4.19-3
mailcap 2.1.53-1
make 4.4.1-2
mesa 23.1.1-1
mesa-utils 9.0.0-2
mkinitcpio 35.2-1
mkinitcpio-busybox 1.35.0-1
mpfr 4.2.0.p9-1
mpg123 1.31.3-1
mtdev 1.1.6-2
nano 7.2-1
nasm 2.16.01-1
ncurses 6.4-1
netctl 1.28-2
nettle 3.9-1
npth 1.6-4
nspr 4.35-1
nss 3.89.1-1
ntfs-3g 2022.10.3-1
numactl 2.0.16-1
ocl-icd 2.3.1-1
oniguruma 6.9.8-1
opencore-amr 0.1.6-1
openexr 3.1.7-2
openjpeg2 2.5.0-2
openresolv 3.13.1-1
openssh 9.3p1-1
openssl 3.0.8-1
opus 1.4-1
p11-kit 0.24.1-1
p7zip 1:17.05-1
pacman 6.0.2-7
pacman-mirrorlist 20230410-1
pam 1.5.3-3
pambase 20221020-1
pango 1:1.50.14-1
patch 2.7.6-10
pciutils 3.10.0-1
pcre 8.45-3
pcre2 10.42-2
perf 6.3-2
perl 5.36.1-1
perl-error 0.17029-4
perl-mailtools 2.21-6
perl-timedate 2.33-4
phoronix-test-suite 10.8.4-1
php 8.2.6-1
pinentry 1.2.1-1
pixman 0.42.2-1
pkgconf 1.8.1-1
popt 1.19-1
portaudio 1:19.7.0-2
procps-ng 3.3.17-1
```

```
psmisc 23.6-1
python 3.11.3-1
python-autocommand 2.2.2-4
python-cairo 1.23.0-6
python-configobj 5.0.8-4
python-fastjsonschema 2.17.1-1
python-gobject 3.44.1-4
python-inflect 6.0.4-2
python-jaraco.context 4.3.0-3
python-jaraco.functools 3.6.0-3
python-jaraco.text 3.11.1-3
python-more-itertools 9.1.1-4
python-ordered-set 4.1.0-4
python-packaging 23.1-1
python-platformdirs 3.5.1-1
python-psutil 5.9.5-1
python-pydantic 1.10.8-1
python-setuptools 1:67.8.0-1
python-six 1.16.0-8
python-tomli 2.0.1-3
python-trove-classifiers 2023.5.24-1
python-typing_extensions 4.6.2-1
python-validate-pyproject 0.13-1
rav1e 0.6.6-1
readline 8.2.001-2
rhash 1.4.3-1
rustup 1.26.0-3
sdl2 2.26.5-2
sed 4.9-3
shadow 4.13-2
shared-mime-info 2.2+13+ga2ffb28-1
slang 2.3.3-2
source-highlight 3.1.9-10
speex 1.2.1-1
speexdsp 1.2.1-1
sqlite 3.42.0-1
srt 1.5.1-3
startup-notification 0.12-7
stress 1.0.6-1
sudo 1.9.13.p3-1
svt-av1 1.5.0-1
sysfsutils 2.1.1-1
systemd 253.4-1
systemd-libs 253.4-1
systemd-sysvcompat 253.4-1
tar 1.34-2
terminator 2.1.3-3
texinfo 7.0.3-1
tmux 3.3_a-3
tpm2-tss 4.0.1-1
tracker3 3.5.2-1
tree 2.1.0-1
tzdata 2023c-2
unzip 6.0-19
usbutils 015-2
util-linux 2.39-4
util-linux-libs 2.39-4
v4l-utils 1.24.1-2
valgrind 3.21.0-1
vid.stab 1.1.1-1
vmaf 2.3.1-1
vte-common 0.72.1-1
vte3 0.72.1-1
vulkan-icd-loader 1.3.245-1
wayland 1.22.0-1
which 2.21-6
```

```
x264 3:0.164.r3095.baee400-4
x265 3.5-3
xbitmaps 1.1.3-1
xcb-proto 1.15.2-3
xcb-util 0.4.1-1
xcb-util-cursor 0.1.4-1
xcb-util-image 0.4.1-2
xcb-util-keysyms 0.4.1-4
xcb-util-renderutil 0.3.10-1
xcb-util-wm 0.4.2-1
xcb-util-xrm 1.3-2
xdg-utils 1.1.3+25+g8ae0263-1
xf86-input-libinput 1.3.0-1
xkeyboard-config 2.38-1
xorg-bdftopcf 1.1.1-1
xorg-fonts-encodings 1.0.7-1
xorg-iceauth 1.0.9-1
xorg-mkfontscale 1.2.2-1
xorg-server 21.1.8-1
xorg-server-common 21.1.8-1
xorg-sessreg 1.1.3-1
xorg-setxkbmap 1.3.4-1
xorg-smproxy 1.0.7-1
xorg-x11perf 1.6.2-1
xorg-xauth 1.1.2-1
xorg-xbacklight 1.2.3-3
xorg-xcmsdb 1.0.6-1
xorg-xcursorgen 1.0.8-1
xorg-xdpyinfo 1.3.4-1
xorg-xdriinfo 1.0.7-1
xorg-xev 1.2.5-1
xorg-xgamma 1.0.7-1
xorg-xhost 1.0.9-1
xorg-xinit 1.4.2-1
xorg-xinput 1.6.4-1
xorg-xkbcomp 1.4.6-1
xorg-xkbevd 1.1.5-1
xorg-xkbprint 1.0.6-1
xorg-xkbutils 1.0.5-1
xorg-xkill 1.0.6-1
xorg-xlsatoms 1.1.4-1
xorg-xlsclients 1.1.5-1
xorg-xmodmap 1.0.11-1
xorg-xpr 1.1.0-1
xorg-xprop 1.2.6-1
xorg-xrandr 1.5.2-1
xorg-xrdb 1.2.1-1
xorg-xrefresh 1.0.7-1
xorg-xset 1.2.5-1
xorg-xsetroot 1.1.3-1
xorg-xvinfo 1.1.5-1
xorg-xwd 1.0.8-1
xorg-xwininfo 1.1.6-1
xorg-xwud 1.0.6-1
xorgproto 2022.2-1
xterm 380-1
xvidcore 1.3.7-2
xz 5.4.3-1
yajl 2.1.0-5
yasm 1.3.0-6
yay 12.0.4-1
zimg 3.0.4-1
zip 3.0-10
zlib 1:1.2.13-2
zsh 5.9-3
zstd 1.5.5-1
```

Appendix B

Process Tracker Result Graphs

Several graphs are presented in this section. For these graphs:

- The **solid black line** represents the total system wattages.
- The **dotted black line** represents the calculated static power usage of the system.
- The **dotted red line** represents the calculated power usage for the program under test.
- The **solid red line** represents the calculated power usage of the program plus the static usage of the system.

The vertical axis represents watts, and the horizontal axis represents a timestamp in miliseconds.

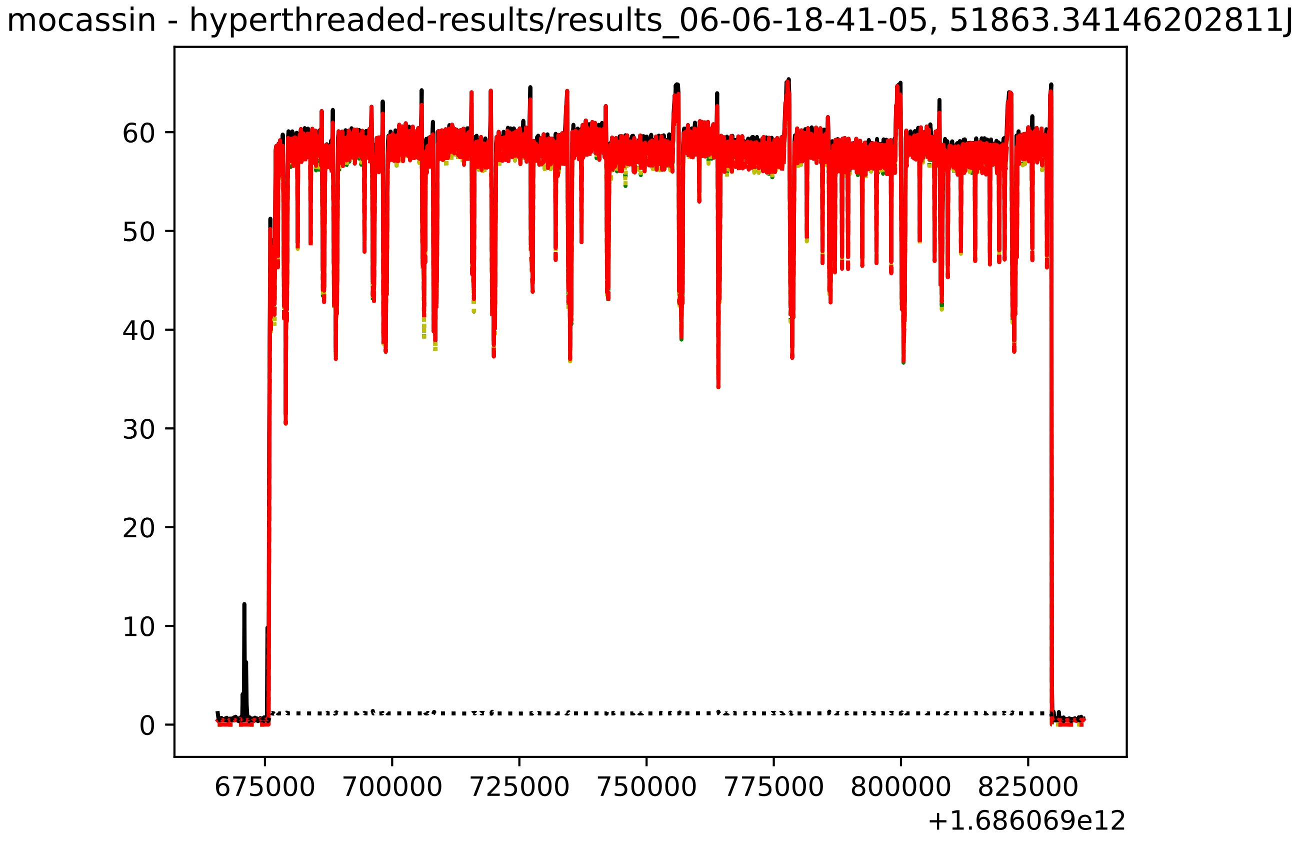

Figure B.1: An example of a good Mocassin test.

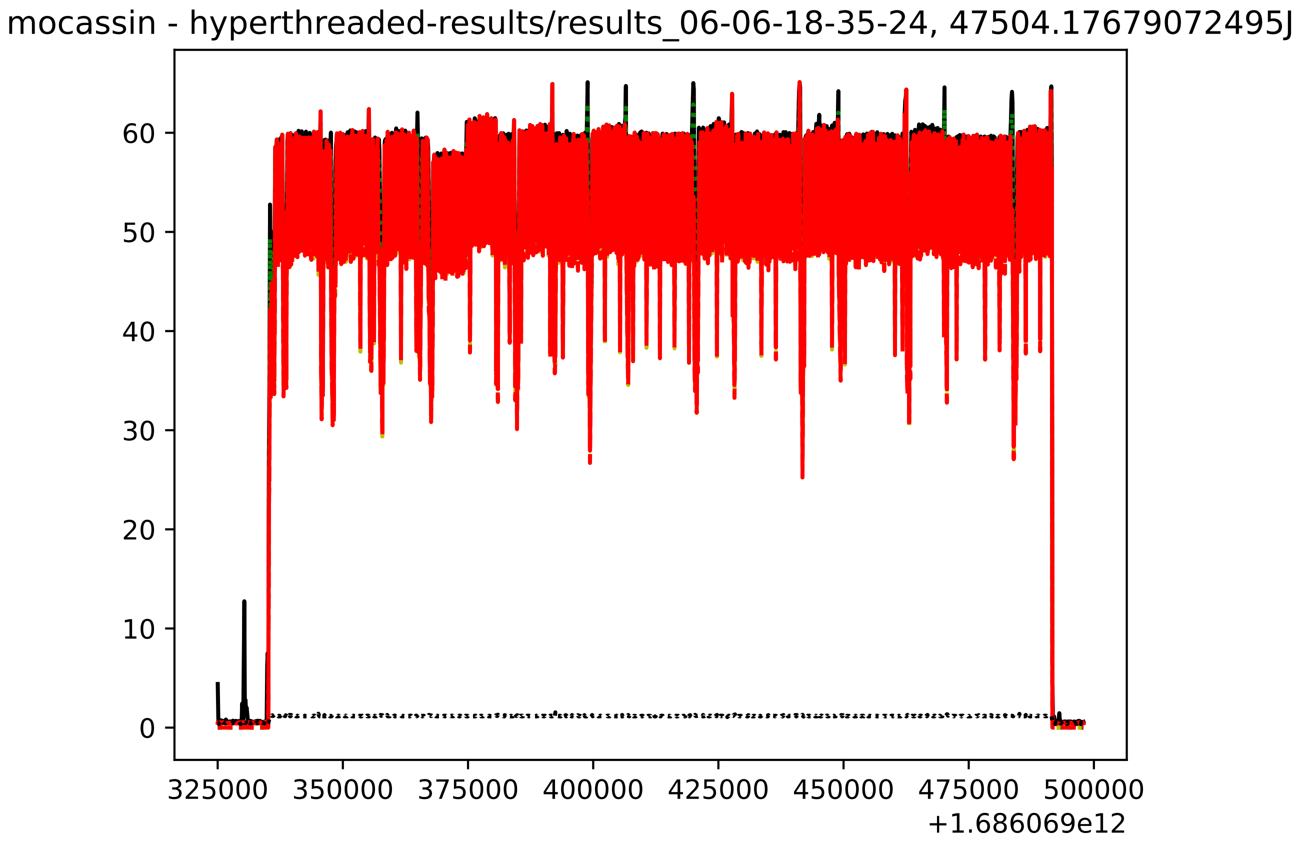

Figure B.2: An example of a bad Mocassin test.

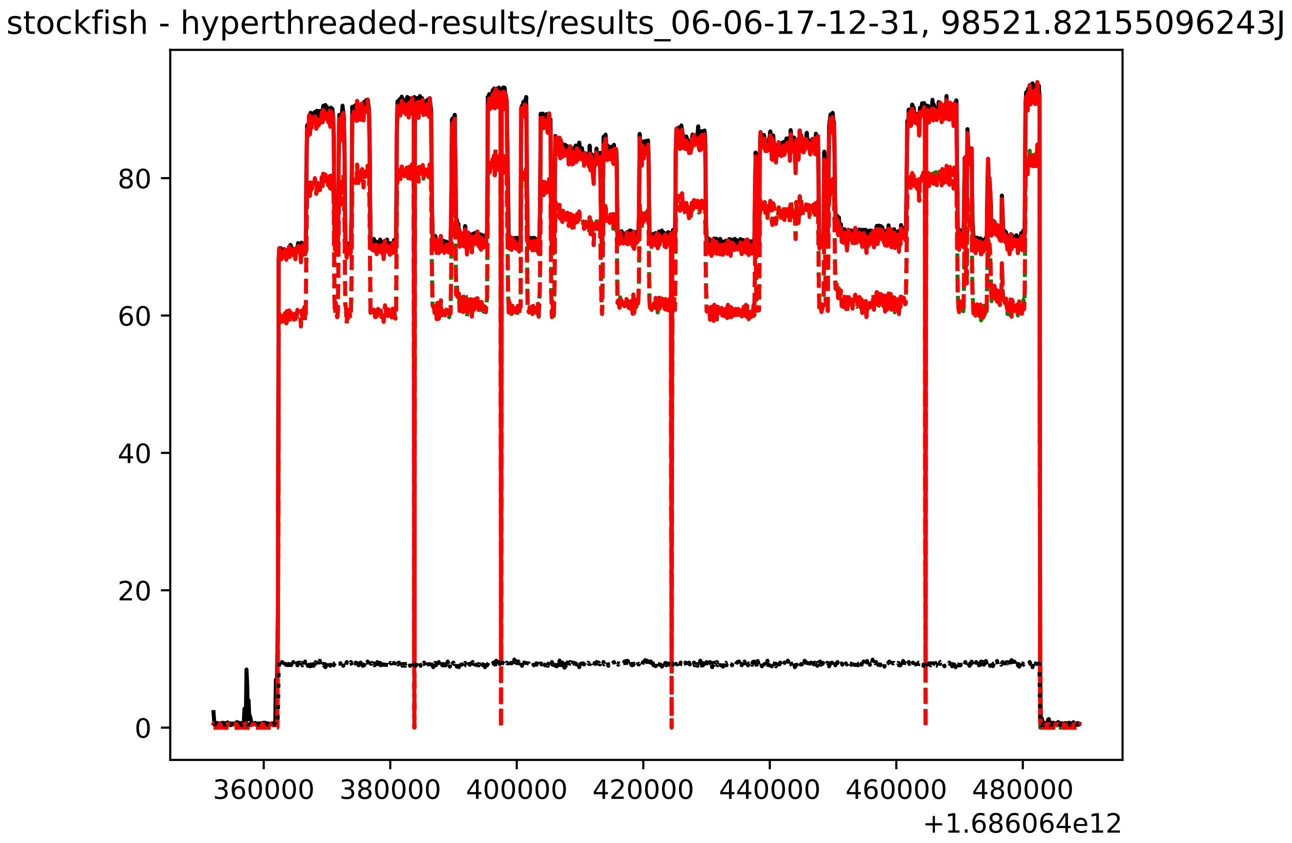

Figure B.3: An example of a good Stockfish test.

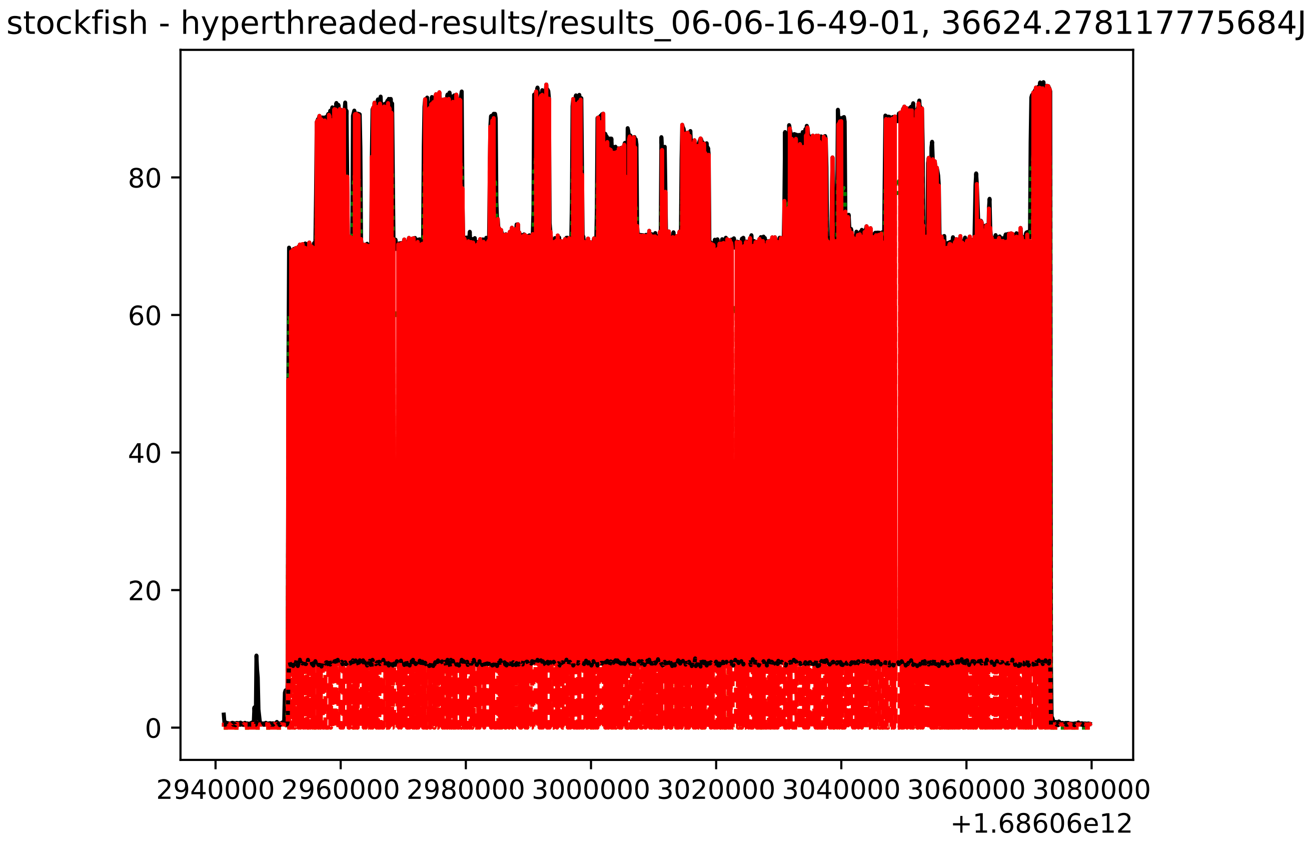

Figure B.4: An example of a bad Stockfish test.