Leipzig University
Faculty of Mathematics and Computer Science
Institute of Computer Science

Measuring the Energy Consumption of Software written in C on x86-64 Processors

Master's Thesis

Submitted by: Tom Strempel
Computer Science

Supervising professors: Prof. Dr. Ulrich Eisenecker
Information Systems Institute

Prof. Dr. Norbert Siegmund
Software Systems

Leipzig, 13.10.2021

Contents

1 Introduction **4**

2 Basic Knowledge and Literature Review **6**

2.1 Energy consumption . . . 6
2.2 The x86-64 instruction set architecture . . . 8
2.3 Hardware . . . 10
2.4 The C Programming language . . . 11
2.5 Granularity . . . 12
2.6 Measurement software . . . 13
2.6.1 RAPL . . . 14
2.6.2 CPU Energy Meter . . . 16
2.6.3 PowerTOP . . . 16
2.6.4 Scaphandre . . . 18
2.6.5 C-LEM and Serapis . . . 19
2.7 Validity . . . 20

3 Software **21**

3.1 Modifying Scaphandre . . . 21
3.1.1 Temperature measurement . . . 25
3.1.2 The importance of synchronization . . . 26
3.2 Test software . . . 28
3.2.1 sorting . . . 28
3.2.2 vpxenc . . . 30
3.3 Python scripts . . . 31
3.4 Bash scripts . . . 32
3.5 Intel Vtune . . . 34
3.6 Correlation between CPU load and energy consumption . . . 36
3.7 Validity of Scaphandre . . . 38
3.7.1 TDP showcase . . . 38
3.7.2 Cross-validation with PowerTOP . . . 39

4 Energy Consumption Comparison of Sorting Algorithms **40**

4.1 Experiment . . . 40
4.2 Temperature during the measurement . . . 42
4.3 Reproducibility . . . 43
4.4 Differences on desktop and server platforms . . . 44

4.5 Validation of process based Scaphandre measurements 45
4.6 Achieving function-level granularity 46

5 Reproduction and Replication Packages 48
5.1 Docker 49
5.2 Reproduction package 50
5.3 Replication package 52

6 Conclusion and Future Work 54

List of Abbreviations 56

List of Figures 57

List of Tables 58

Listings 59

Chapter 1

Introduction

In 2016 German data centers consumed 12.4 terawatt-hours (or $12.4 \cdot 10^9 kWh$) [1] of electrical energy, which accounts for about 2% of Germany's total energy consumption in that year. In 2020 this rose to 16 terawatt-hours (or $16 \cdot 10^9 kWh$) [2] or 2.9% of Germany's total energy consumption in that year. These figures do not account for the energy consumption of private devices like laptops, desktop PCs or smartphones, because they do not have large-scale dedicated and therefore easy to measure power supplies like data centers. But due to the widespread use of consumer computer devices in modern day-to-day life, it can be expected that the total power consumption of these will also rise further.

The ever-increasing energy consumption of computers consequently leads to considerations to reduce it to save energy, money and to protect the environment. Battery-powered devices like laptops will also have the benefit of increased battery life due to lower energy consumption. This basic notion is encapsulated in the GreenIT concept, which aims to reduce negative impacts (e. g. energy consumption, toxicity, water use) in the life cycle (e. g. design, manufacturing, distribution, use and disposal) of IT products [3]. Energy consumption in the life cycle of personal computers and server systems is the main focus of this thesis. It aims to answer fundamental questions related to the energy consumption of software to lay the groundwork for further studies in the future. The software scope is chosen because it does not require extra hardware, e. g. an ampere meter, to take measurements. Being able to use a readily available program to measure energy makes this solution more accessible. Fine-tuning may be necessary on individual devices since the same software will not run power optimal on all devices.

The following questions are to be answered in connection with the energy consumption of applications:

1. Which methods exist to measure energy consumption?
2. How granular, precise, valid and reproducible can a measurement be?
3. What is having an influence on energy consumption?
4. Does algorithm design influence it?
5. How relevant is the energy consumption of small functions?
6. Can other features like CPU load be used to approximate the energy consumption?
7. Is the time expenditure in optimizing programs for less energy use worth it?

Questions four to seven are specific cases of question three. The scope of this thesis is further narrowed down to only apply to the most widespread processor architecture used in laptops, desktop PCs and servers, namely x86-64. All measured programs used are written in the C programming language since it is a widespread and simple language that has a lot of third-party tools to work with. All data supplementing this thesis can be downloaded at `https://zenodo.org/record/5559595`.

Chapter 2

Basic Knowledge and Literature Review

2.1 Energy consumption

Measurements can be taken in Joule or Watt. These units can be used interchangeably when the time frame is known. It is generally advised to use Joule for small and time-limited measurements like the energy usage of a function. Watt is better suited for longer or indefinitely running programs.

$$Power = \frac{Energy}{Time}$$

Formula 1: Equation for power

Ardito et al. [4] describe how methods of energy measurement can be categorized. Three measurement techniques can be distinguished, namely instant power measurement, time measurement and model estimation. This categorization scheme (see tab. 2.1) will be used in this thesis. A similar structure is used by Rieget et al. [5], where the subdivisions included measurements with and without model derivation and power analysis attacks from the subject area of IT security.

Instant power measurements require some sort of physical instrumentation, e. g. an attached ampere meter or an internal measurement of a component, which is exposed via an interface. The accuracy of the read-out power consumption depends on the sampling frequency of the measurement device. Facilities for exposed internal measurements must be implemented by the hardware vendor of the corresponding component. Suitable software for reading out is also needed. Standards like the Advanced Configuration and Power Interface (ACPI) exist, but these do not provide suitable isolation techniques to get the power consumption of a single program. The complexity of the ACPI [6] standard further disqualifies it as a simple solution for application-level power measurement. Another solution is adding an ampere meter to the PC's power supply and measuring the energy consumption. Adding extra hardware for measurement has the disadvantage of further complexity due to the energy

Measurement technique	Advantages	Disadvantages
Instant power measurement	Precise if the sampling frequency is high	Physical instrumentation needed. Difficult to isolate a single software application's contribution.
Time measurement	Precise if the exact energy stored in the battery is known.	Requires many repetitions of long tasks. Difficult to isolate a single software application's contribution. Automatic power saving in low battery levels can skew results.
Model estimation	No instrumentation is required. Easy to isolate a single application's contribution.	Precision is not always declared.

Table 2.1: Categorization of energy measurement methods according to [4]

efficiency of the power supply. Furthermore, not every developer may have the means and expertise to acquire and install the required hardware. Later, a way to measure the consumption of the most power-consuming part of a PC, the processor [7], is being presented to circumvent these problems.

Time measurement is based on the drainage of the energy stored in a battery to calculate the consumed energy. Laptops are ideally suited for this technique because they are already equipped with an integrated battery. The battery's actual charge level can be read out by the Operating System (OS) via the ACPI interface. But on desktop and server systems this technique would mandate adding a battery to the power supply for measuring, which comes with the same problems as described before with adding an ampere meter. Provided one has access to a laptop, this technique is suitable to cross-validate findings from the other two measurement techniques.

Creating a model can remove the biggest disadvantage of the previous two techniques, namely the dependency on physical instrumentation. Furthermore, it becomes possible to isolate the energy consumption of a single application. Models can employ a combination of software analysis and energy modelling [8] or use only energy measurements to set the developed model into context [9]. Without some form of measurement, it is not possible to get a concrete energy value in Joule, since no frame of reference for energy consumption would exist. Hybrid versions between model estimation and the other two techniques are therefore possible.

2.2 The x86-64 instruction set architecture

Desrochers et al. states that *"CPUs are the focus of power optimization as they make up the largest single component of a system's energy budget. Next in line after the processors is the DRAM"* [7]. An example is given where on a server system under load the CPU consumes 130W of power while the DRAM only consumed 13W. Dedicated Graphics Processing Units (GPUs) can have a higher power consumption, e. g. an NVIDIA V100 can consume 250W [10]. Since dedicated GPUs are used for more specialized use cases, they are excluded from deeper analysis in this thesis. Due to their prominent role in energy consumption, processors used in laptop, desktop and server computers are more closely examined. Almost all processors in these systems are based on the x86-64 architecture and are produced by one of two major vendors, AMD[1] and Intel[2]. Intel has a bigger market share (see fig. 2.1), especially in the server market (see fig. 2.2), but AMD's market share is rising in recent years.

Overall x86 CPU Share

Overall x86 CPU Share	2021 Q2	2021 Q1	2020 Q2	Change	Change
Includes IoT and SoC	Share	Share	Share	Quarter	Year
Intel	77.5%	79.3%	81.7%	- 1.8	- 4.2
AMD	22.5%	20.7%	18.3%	+ 1.8	+ 4.2
VIA	0.0%	0.0%	0.0%	+ 0.0	- 0.0
Total	100.0%	100.0%	100%		

Figure 2.1: x86 market share [11]

Server CPU Share excluding IoT

Server CPU Share	2021 Q2	2021 Q1	2020 Q2	Change	Change
	Share	Share	Share	Quarter	Year
Intel	90.5%	91.1%	94.2%	- 0.6	- 3.7
AMD	9.5%	8.9%	5.8%	+ 0.6	+ 3.7
Total	100.0%	100.0%	100.0%		

Figure 2.2: Server CPU market share [11]

Definitions of the machine code instructions, available data types and data type size are all parts of the implemented Instruction Set Architecture (ISA). The hardware design (i.e. design documents, mechanical drawings, schematics and layout data) of x86-64 itself is closed source. Self-conducted tests and experiments are needed for a detailed examination of processor features and inner workings because no extensive information is available to the public. Processors implementing the x86-64 ISA, other names are x86_64, x64, amd64 and Intel64, are the most commonly used processors in notebooks, desktop computers and server systems to date. Smartphones and micro-controllers use different ISAs like ARM and AVR-RISC and thus are out of the scope of this work. ARM processors can also be used for desktop computers, but their market share is so negligibly low, that it is not listed in most figures.

[1] https://www.amd.com/en (accessed 04.10.2021)
[2] https://www.intel.com/content/www/us/en/homepage.html (accessed 04.10.2021)

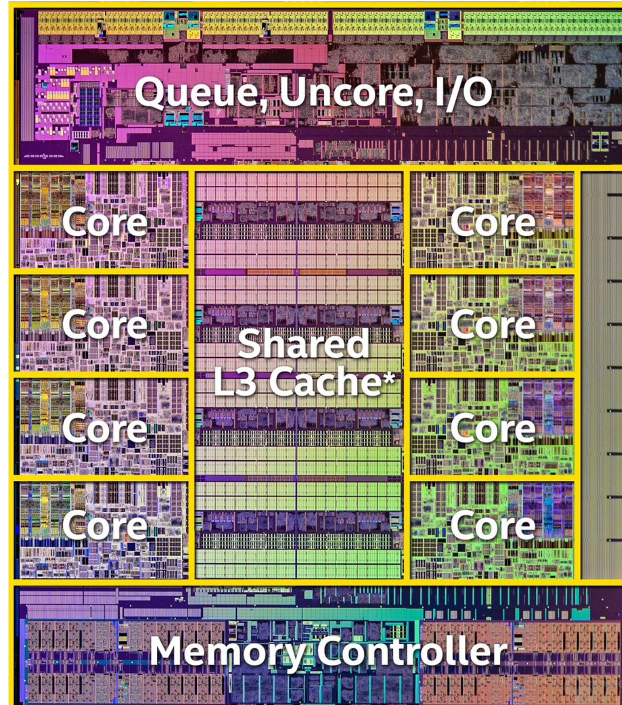


Figure 2.3: Intel Haswell processor die [12]

A processor (or Central Processing Unit (CPU)) can be divided into sub-domains (see fig. 2.3). Processing cores, where most calculations take place, are the central part of a processor. One basic core contains a control unit, arithmetic logic units and computing registers. Other components can also be included. Modern processors contain multiple cores and all these cores in combination form the core domain. Another name for a processor core is CPU core. Other domains include the memory controller for the DRAM. The makeup of the domains can change depending on the processor. Two common terms regarding processor utilization are "idle" and "under load". Idle means that the processor is not or only slightly used by programs. In contrast, a processor under load is used to a large amount or completely by programs.

2.3 Hardware

Three different computers and processor microarchitectures are used in this work (see tab. 2.2). Each of them is from a different segment and therefore their properties differ. The laptop is a *Toshiba Tecra Z50*, the desktop PC is custom build and the server is a *Dell PowerEdge R640*. A processor generation overview (including release dates) is given in figure 2.4. The differing number between cores and threads in table 2.2 is due to a technology called Hyperthreading [13], which enables running two threads or programs on one core. This feature is deactivated on the provided server system.

Processor	Architecture	Segment	Cores	Threads	TDP
i5-6200U	Intel Skylake	Laptop	2	4	15W
Ryzen 5 1600	AMD Zen	Desktop	6	12	65W
Xeon E5-2690 v4	Intel Broadwell	Server	14	14	135W

Table 2.2: Available hardware for testing

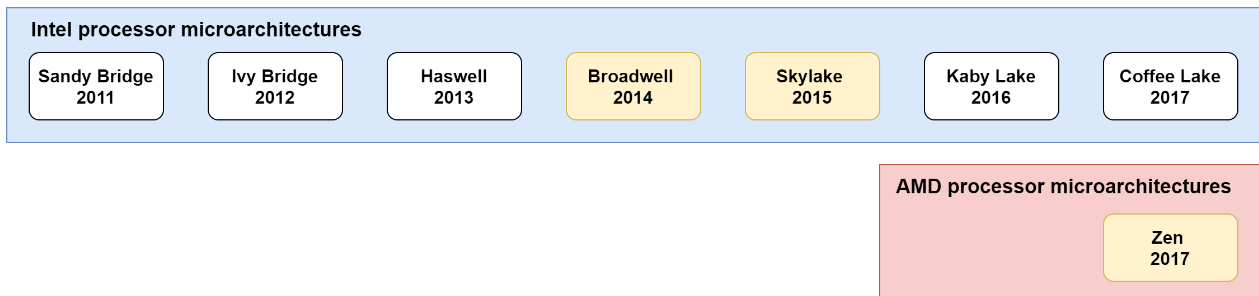


Figure 2.4: x86-64 CPU microarchitectures (yellow colored ones are used in this work)

Thermal Design Power is defined as *"the maximum power a processor can draw for a thermally significant period while running commercially useful software."* [14]. This means the Thermal Design Power (TDP) sets the dimensions of the cooling solution since it must dissipate the waste heat associated with a specific TDP. Laptops with limited space and air circulation can only afford processors with low TDPs like the Intel i5-6200U. Desktop PCs have larger dimensions and dedicated fan cooling systems for their components, therefore increasing their heat dissipation capabilities manifolds over laptops. Data centers have centrally managed airflow cooling thus they can afford the processors with the highest TDPs. One processor from each of these segments (see tab. 2.2) is chosen to spot potential unexpected differences in energy consumption behavior between them. If not further specified in the corresponding figure, the i5-6200U laptop system is meant. Figures from two other processors are only provided when new information can be showcased with them. Providing every figure in three variations is not done because of space concerns.

2.4 The C Programming language

C was developed by Dennis Ritchie for usage on the DEC PDP-11 computer running the UNIX operating system. Brian Kerninghan, who worked on UNIX development alongside Ritchie, is often credited for the creation of C too. He refuted such claims in an interview saying that *"I had no part in the birth of C, period. It's entirely Dennis Ritchie's work. I wrote a tutorial on how to use C for people at Bell Labs, and I twisted Dennis's arm into writing a book with me."* [15]. Despite first being used for programming operating systems, C is a general-purpose programming language [16, Page XI]. It features a static typing system, which means that the type of variables is declared at compile time and not dynamically at run time. C belongs to the imperative programming paradigm, specifically the procedural paradigm. Procedures or functions are the main structuring mechanism of the source code. C does not feature object-oriented programming.

C programs are compiled into executable binaries. Different compilers are available. If not further specified, the C compiler from the GNU Compiler Collection (GCC)[3] called `gcc` is used. Being a compiled language programs written in C run faster than corresponding programs in interpreted languages.

The process of creating an executable from source code with the GCC is split into four main steps: preprocessing, compilation, assembly and linking (see fig. 2.5) [17]. This process remains largely the same for other compiled languages such as C++ and Fortran, the subprograms for the preprocessor and compiler may change.

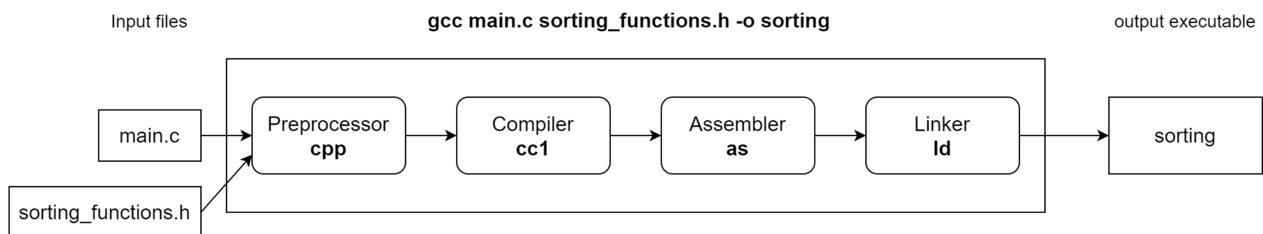


Figure 2.5: GCC compilation steps

`gcc` can be supplied with extra options to further customize the compile process and results. These options are also called compiler flags. The most important flags for optimization, which means internally restructuring the compiled code to gain a performance increase, are the *-O* flags. These will be later used in this thesis to compare the energy consumption of unoptimized and optimized programs.

It was contemplated using Java instead of C. But the Java Virtual Machine (JVM), on which Java runs, would introduce a further layer of complexity. Not only the characteristics of the written code would need to be considered, but those of the JVM as well. Since the goal of the thesis is to answer fundamental questions and not specific ones like how the JVM behaves, it was decided against using Java instead of C. Duarte et al. [18] also state that *"We used a Java program, which means that part of the energy*

[3] https://gcc.gnu.org/ (accessed 04.10.2021)

usage might be due to the Java Virtual Machine execution." [18]. It is further specified that *"Other programming languages with less execution overhead will probably generate results that can be actually attributed to the particular program."* [18]. This supports the decision of using the C programming language, which has less overhead due to not needing an intermediate platform like the JVM for execution.

2.5 Granularity

The Granularity of a measurement describes on which level of program abstraction it takes place (see fig. 2.6). This must be seen in the context of wanting to measure the energy consumption of a program and if possible its subroutines.

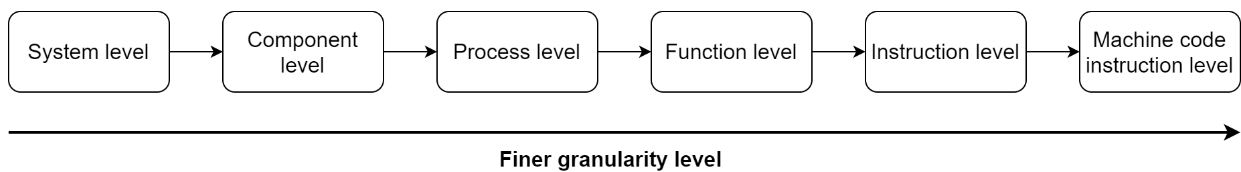


Figure 2.6: Granularity levels

The most coarse method would be adding an ampere meter to the power supply of a computer and measuring the difference between an idle state and the running program. Many disturbing factors, e. g. power supply efficiency or non-contributing power-hungry components, can influence this setup. Power supplies are used to transform the AV wall power to lower DC voltages more suitable for electrical components. The voltage is further transformed by the mainboard down for different components. Power supplies differ in their energy efficiency. This means the same program takes less energy to run on a PC with a more energy efficient power supply than on an otherwise equal PC with a less energy efficient PC.

Disturbance factors like these are eliminated by only measuring a component of interest, e. g. the processor. This only provides information about energy consumption in the context of the processor, therefore achieving component-level granularity. Other components such as storage or wifi cards can also be measured if a suitable interface exists. This approach also ignores the energy efficiency from the power supply, which makes the measurement less specific but more platform-independent.

By isolating or measuring the energy consumption of a single process (the running program) an even finer granularity can be achieved. On this level, the program is effectively isolated from all other processes running on the OS. Based on this even finer-grained measurements, namely, that of the program's subroutines, such as functions, can be achieved. Some programming languages besides C like Java also provide classes or modules, but these will be not looked into since only the imperative programming structures of C are of interest. It must be stated that further granularity sub-levels can exist depending on the used programming language. Granularity on the function level already enables finding the most energy-consuming parts of the

program. Measuring the consumption of all base instructions like addition, multiplication, memory allocation etc. would enable instruction-level granularity [19]. Energy modelling on machine code granularity would have the advantage of being independent of the used programming languages since only an already compiled executable would be analyzed. At the same time this has the downside of being forced to work with machine code instructions, which are complicated to read. Molka et al. [20] did achieve this granularity by measuring the energy consumption of arithmetic operators and data transfers. The used method is not suitable for the use case of isolated single applications and hence not further discussed.

The complete isolation of the target process of the rest of simultaneously running processes (process-level granularity) on the OS is necessary to eliminate disturbing factors. Achieving function-level granularity is judged to be enough to enable programmers to spot energy-intensive parts of their code. The efforts in this work are mainly concentrated on achieving process- and function-level granularity.

2.6 Measurement software

A survey conducted by Rieger et al. [5] in 2017 concluded there is a lack of adequate tools by stating *"The sobering result of this survey was that there are hardly any actual ready-to-use development tools; those that we found are all platform-specific."* [5]. Non x86-64 platforms such as microcontrollers [21] or smartphones were also included in this survey. An own survey focusing on methods working on x86-64 processors was conducted. In the following sections, an overview of the most promising tools is given. A big problem encountered during the research is, that code was either not available to the public or available but not maintained. Two sources of energy measurement software are used in this thesis. Either the software was developed during a research project [19, 22] or it was developed by other interested parties for administrative work [23, 24].

2.6.1 RAPL

Intel processors of the Sandy Bridge generation (released in 2011) and newer have an onboard power meter capability called Running Average Power Limit (RAPL). RAPL [25] was originally developed in combination with Dynamic Voltage and Frequency Scaling (DVFS), also called CPU throttling, and Turbo boost in mind. DVFS enables the processor to lower its frequency and voltage with it to conserve power when the system is idle or not under full load. Turbo boost technology enables the processor to boost its frequency over its normal maximum for a limited time because it leads to more waste heat energy. If this heat energy can not be dissipated quickly enough by the installed cooling solution the processor will overheat. This is a severe factor in laptops with their design-related smaller dimensions for radiators and heat pipes. Therefore RAPL was developed to keep track of the consumed energy and thus waste heat. The maximal power a processor can consume for significant periods of time without producing too much heat is the TDP, making it an important metric in developing cooling systems for a processor. This also means that over a thermally insignificant period the TDP can be exceeded, but on average its power limit must be adhered to. This is where the name Running Average Power Limit stems from. The first implementation in the Sandy Bridge processors used model estimation. In later processor generations (Haswell and newer) sensors measuring the actual electrical current were installed, putting RAPL in the instant power measurement category described in table 2.1.

RAPL measurements can encompass four domains:

- Package (total power consumption)
- Core
- Uncore
- DRAM

The Core domain contains the CPU cores used for the computation itself. Uncore is a catch all term for things that are neither Core or DRAM. What is part of the Uncore domain can change depending on the processor generation. The most power hungry component in the Uncore domain, if present, is the integrated graphics unit. Other components can include the memory controller, IO-devices and the shared L3 cache. DRAM refers to the power consumption of the DRAM DIMMs on the mainboard. DRAM is also often referred to as RAM and main memory.

For each supported power domain a Machine Specific Register (MSR) filled with an 32-bit integer is exposed. It does not measure energy in Joule, instead it uses *energy units*. These energy units correspond to a processor depended energy value in Joule. The theoretical possible energy resolution is one energy unit, e. g. 61μJ for a Skylake processor. The register as such contains the consumed energy since the last computer startup. To compute the energy consumption in a given timeframe two measurements are needed and the value of the earlier measurement is subtracted from the later one. The MSRs for RAPL are updated approximately every millisecond [26]. Although in practise a minimal interval of 100 milliseconds between measurements is chosen [26]. Under Linux the *rdmsr* command can be used to read out MSRs directly. For example

the `rdmsr` command allows to read out the size of the energy unit on a system (see tab. 2.3): `$ sudo rdmsr 0x606H`

Architecture	Energy unit size
Sandy Bridge	15.3μJ
Haswell	61μJ
Skylake	61μJ

Table 2.3: Energy unit sizes across different architectures

Reading out MSRs directly is error prone due to register numbers being specific to one processor generation. A suitable middleware is needed to read out RAPL on different systems without modification. This role is performed by the Power Capping Framework (PowerCap)[4] on Linux. RAPL was only supported on Intel processors for a long time since it was originally developed for them. In 2020 Google engineer Victor Ding extended the support for AMD processor generations based on the Zen architecture released in 2017. AMD supports the same interface just on different MSR numbers. This feature was included in the Linux kernel version 5.11 released on the 14th February 2021[5]. With that it is possible to cover all widely used x86-64 processors in one measurement software.

The package domain encompasses the total energy consumption of the processor; as such it is the sum of the three other domains. It is to be expected that computing intensive workloads will mainly increase the power consumption of the Core domain, graphical workloads should increase Uncore power consumption and memory intensive workloads should increase DRAM power consumption. These assumption will be tested later in the thesis.

RAPL has since its introduction developed into a popular way to measure energy consumption of x86-64 systems. Next to usage in the scientific community it is also incorporated into many libraries, projects and tools, e. g. jRAPL[6], pyRAPL[7] or rapl-tools[8]. A lot of these were not updated in years, so only projects with recent updates are looked into further. Promising projects will be presented in the following sections. Similar systems exist for other computer parts like GPUs. The NVIDIA Management Library (NVML)[9] also provides a power management interface. GPU-intensive workloads like graphics or crypto mining can provide cases for future study due to being energy-intensive tasks.

[4] https://www.kernel.org/doc/html/latest/power/powercap/powercap.html (accessed 04.10.2021)
[5] https://www.phoronix.com/scan.php?page=news_item&px=AMD-RAPL-Linux-Now-19h (accessed 10.10.2021)
[6] https://github.com/kliu20/jRAPL (accessed 04.10.2021)
[7] https://github.com/powerapi-ng/pyRAPL (accessed 04.10.2021)
[8] https://github.com/kentcz/rapl-tools (accessed 04.10.2021)
[9] https://developer.nvidia.com/nvidia-management-library-nvml (accessed 04.10.2021)

2.6.2 CPU Energy Meter

Beyer and Wendler developed CPU Energy Meter [22][10], a software that can measure the power consumption of a processor over time. Only support for Intel CPUs is included because the registers are read out directly and not over PowerCap. The program works by starting it and terminating it via CTRL+C (the key combination for program termination on Linux), after that an overview of consumption metrics is shown (see listing 2.1). In the Bash programming language, the $ is used to indicate a command execution and # is used for comments. The same notation is used for the listings containing source code in this thesis. Executables such as `./cpu-energy-meter` and functions are written in monospace font in the text.

Listing 2.1: CPU energy meter output

```
# shows output
$ sudo ./cpu-energy-meter
+--------------------------------------+
| CPU Energy Meter            Socket 0 |
+--------------------------------------+
Duration                  3.136108 s
Package                   1.797180 Joule
Core                      0.172546 Joule
Uncore                    0.029419 Joule
DRAM                      1.605530 Joule
```

Superuser privileges (Linux equivalent to Admin privileges on Windows) are needed to execute the program. This can be done by adding *sudo* (**s**uper**u**ser **do**) before the command itself (see listing 2.1). These privileges are needed because the MSRs containing the RAPL measurements require superuser access to be read out. There are no other output formats available and every measurement has to be terminated manually since no timer is available. CPU energy meter can thus not practically be used for extensive measurement experiments. It instead provides a quick-to-install and easy-to-read method to try out energy measurements to get an idea of the target CPU's normal energy consumption.

2.6.3 PowerTOP

PowerTOP [24] uses another measurement technique (see table 2.1) than the previously presented methods which used instant power measurement with RAPL, namely time measurement. By measuring the energy drained from the battery, the power consumption of the device and its subcomponents are calculated. As this method is only possible on devices with a directly attached battery, it can practically be used only on laptops. PowerTOP also has the added function of making suggestions for optimizations. Sodhro et al. [27] used PowerTOP in conjunction with an ampere meter to measure the energy consumption of various web and locally executed applications. Af-

[10] `https://github.com/sosy-lab/cpu-energy-meter` (accessed 04.10.2021)

ter installation PowerTOP must be calibrated first, before taking a measurement (see listing 2.2).

Listing 2.2: PowerTOP calibration and usage

```
# calibrate PowerTOP
$ sudo powertop --calibrate
# measure for 120s and write results in a html
$ sudo powertop --html=powertop.html --time=120s
```

Exporting to a CSV file is also supported. PowerTOP uses a different system to calculate the energy consumption of a process. The ACPI standard defines C- and P-states, which describe the powering down of subsystems and voltage reduction of a processor. For example, C0 describes an active CPU core, while in the C3-state the CPU core is completely inactive ("sleep"). The states in between describe mixed versions of these. If a CPU core is in the C0-state its frequency can be further regulated by using the P-states. A core in the P0-state runs at its maximum frequency. The other states describe cores running at lowered frequency. PowerTOP calculates the energy consumed by a running process by logging how many times the CPU core running it changes between different power states and setting it into context with the battery energy drainage. This has the disadvantage that running a program that fully utilizes a CPU core all the time without changing to a sleep state or lower power state, would lead to PowerTOP reporting that no energy consumption did take place. On an idle system the display of a laptop is its main power consumer (see fig. 2.7).

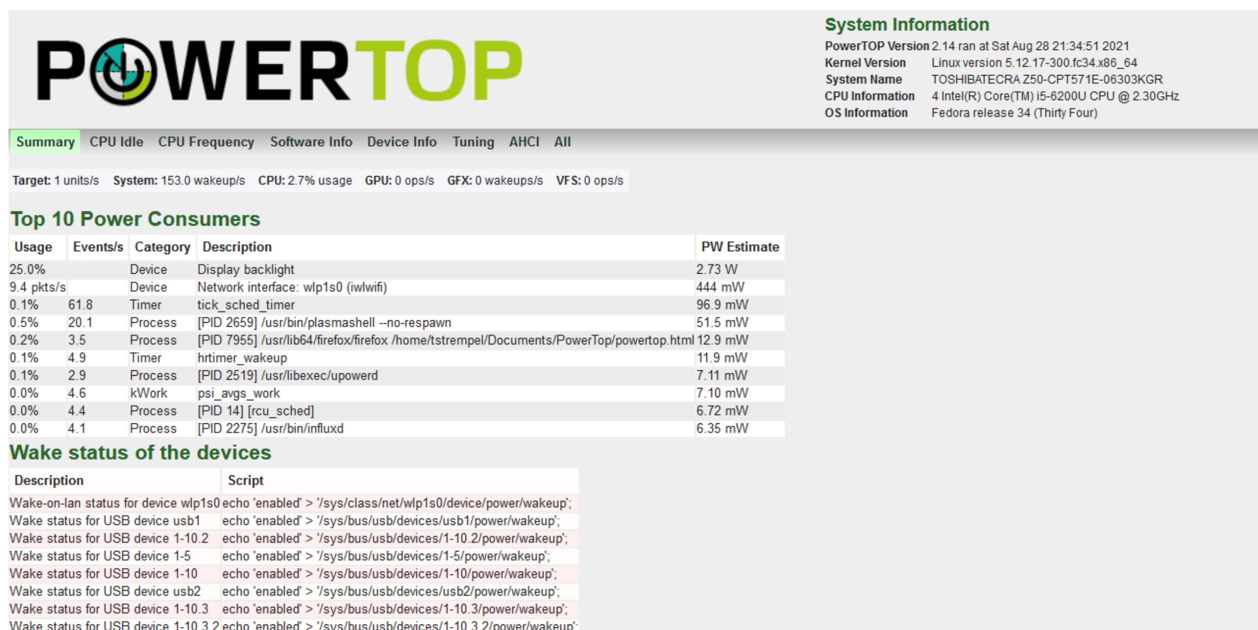


Figure 2.7: PowerTOP output of an idle laptop system

PowerTOP is better used for searching energy consumption hotspots on a whole system rather than inside a single application. Later in this thesis, PowerTOP will be used to cross-validate measurements taken with the other measurement techniques.

2.6.4 Scaphandre

Scaphandre [23] is a freshly developed RAPL based software. At the time of writing (09.06.2021), no scientific publications using Scaphandre for energy measurements could be found. The main differences to the previous programs are an export option to structure the read-out data and support to measure applications isolated from the rest of the system. Version v0.3.0 features different exporters. For this project, the JavaScript Object Notation (JSON) exporter will be used because most programming languages have extensive support to work with JSON. Other exporters include support for monitoring systems like Prometheus[11], Warp10[12] or riemann[13]. Listing 2.3 shows an example output of the scaphandre JSON exporter, it is made human readable with the *js-beautify* program. Normally the output is provided without any newlines and whitespaces.

Listing 2.3: Scaphandre JSON output (beautified)

```
$ scaphandre --no-header json  --step 2 --timeout 2 | js-beautify
{
    "host": 828016.0,
    "consumers": [{
        "exe": "/usr/bin/plasmashell",
        "pid": 2770,
        "consumption": 157792.55
    }, {
        "exe": "/opt/google/chrome/chrome",
        "pid": 3607,
        "consumption": 78896.27
    }],
    "sockets": [{
        "id": 0,
        "consumption": 828888.0,
        "domains": [{
            "name": "core",
            "consumption": 552902.0
        }, {
            "name": "uncore",
            "consumption": 266203.0
        }, {
            "name": "dram",
            "consumption": 10729.0
        }]
    }]
}
```

[11] https://prometheus.io/ (accessed 04.10.2021)
[12] https://warp10.io/ (accessed 04.10.2021)
[13] http://riemann.io/ (accessed 04.10.2021)

The output is structured into three parts: system-wide metrics (like the complete energy consumption of the host system), a list of the most energy-consuming applications and the energy consumption per domain per processor (also called CPU socket). Multi socket systems, e. g. servers with two or four CPU sockets on one mainboard are supported this way.

Scaphandre uses an internal model based on the CPU time (measured in *jiffies*) spend in processes to calculate per-process energy consumption metrics. A *jiffy* corresponds to 0.004 (the default value) CPU seconds on modern Linux kernels. It also states the maximal resolution of measuring the CPU time because only whole *jiffies* are counted. The CPU load is defined as the number of CPU seconds used in a second, e. g. two CPU seconds spend on a four-thread processor corresponds to a load of 0.5. The normalization with the processor thread, the maximal load can only ever be 1.0, is not done in every program or publication. Without normalization, the example earlier would lead to a CPU load of 2.0. In this work, only the CPU load with normalization is used. The amount of energy a process consumes in a time frame is calculated by dividing the CPU time (represented as *jiffies*) used by a process by the total amount of spend CPU time and multiplying the result with the total amount of energy consumed by the host.

$$ProcessEnergy = \frac{ProcessJiffies}{TotalJiffies} \cdot TotalEnergy$$

Formula 2: Equation for sorting efficiency [28]

This means a process consuming 10% of all CPU time in a time frame is assigned 10% of the total energy consumption in that same time frame. This approach only takes the CPU time into account, other metrics like memory consumption, temperature etc. are not. Scaphandre is the only software encountered achieving process-level granularity. Measuring power consumption on AMD processors is supported on new Linux kernels (> 5.11) over Powercap and measurements are hence not limited to Intel processors only. Since Scaphandre offers the largest set of features of all compared software, it will mainly be used in the experiments later on in this work.

2.6.5 C-LEM and Serapis

Couto et al. [19] developed a model named C-LEM[14] which estimates the energy usage of different base instructions of C, e. g. addition or requesting memory. This requires reading out RAPL to get measurements for these instructions. Another model called Serapis[15] then estimates energy consumption of a Software Product Line (SPL) based on the energy consumption of its base instructions. The concept of a SPL is being defined as the *"systematic development of products that can be deployed in a variable way, e.g., to include different features for different clients"*.

[14] https://gitlab.com/MarcoCouto/c-lem (accessed 04.10.2021)
[15] https://gitlab.com/MarcoCouto/serapis (accessed 04.10.2021)

This approach combines the instant power measurement and model estimation techniques (see tab. 2.1). No support for newer Intel CPUs like Skylake is available. Neither documentation is bundled with the developed software nor was it be provided at request. Therefore no own measurements could be taken and C-LEM and Serapis are not used further in this thesis. Nevertheless, the ground concept of measuring the energy of SPLs is worth exploring in future works since interest from the scientific community exists. For example, Siegmund et al. [6] describe a model of optimizing energy consumption that uses techniques gained from SPLs.

2.7 Validity

As the measurement methods used in this thesis are RAPL-based the validity of them must be proved. Desrochers et al. [7] developed an experimental setup to measure the DRAM energy with as little interference as possible, comparing it with the RAPL measurements afterward. These findings are important for the validation of the DRAM domain, especially since they were created using multiple different systems and benchmarks. In general, not more than a 20% difference exist between the physical and RAPL measurement [7]. For newer processors, including the Intel Haswell microarchitecture (see fig. 2.4), the precision is better than in previous generations. Because all processors used in this thesis are of the Broadwell generation or newer, the DRAM measurements can be assumed to be accurate.

Zhang et al. [29] took another approach by setting a limit for power consumption via RAPL (similar to the TDP limit) and measuring how accurate this limit was kept. 14 out of 16 different test benchmarks achieved a mean absolute percentage error (MAPE) of 2%, the other two achieved a worse result over 5% [29]. It is also shown that RAPL is more accurate when a lot of energy is consumed. Reduced accuracy can be expected when the processor runs no heavy workloads (idling). As before, newer processors achieve better results than older ones.

Khan et al. [26] evaluated RAPL by comparing it with the wall power measurements from servers inside the *Taito* supercomputer of the Finnish Center of Scientific Computing. A high correlation (around 99%) of RAPL to wall power is achieved. The estimation error (MAPE) amounts to 1.7%. The performance overhead of reading out RAPL is considered negligible (<1%) [26]. High temperatures are also able to influence energy consumption, therefore a constant temperature must be achieved or a varying temperature is taken into account within a model. Mazouz et al. [30] also put forward a methodology with which the validity of RAPL can be proven. At the time of writing this method was not used in practice. Due to these extensive tests, RAPL is considered to be a valid way of measuring the energy consumption of a processor.

Chapter 3

Software

3.1 Modifying Scaphandre

Scaphandre's most recent version v0.3.0 at the time of writing (09.06.2021) has missing features, e. g. reporting timestamps, CPU load, memory usage and CPU temperature, which need to be implemented. Some shortcomings like the missing timestamps are already addressed on unmerged development branches on the projects GitHub page[1]. Henceforth the branch `#75-shortcomings-of-current-json-exporter` is used as a basis for an own fork[2]. A fork describes a copy of an already existing git repository so that it can be modified by the new author without needing access to the original repository. All source code changes (see listing 3.1) are concentrated into the `src/exporters/json.rs` file because only the JSON exporter needs to be modified. Scaphandre is written in the programming language Rust which closely resembles C/C++ but does not use manual memory allocation. Rust is not further explained in detail due to the close resemblance to C/C++ and the easy-to-read code. Code portions in listing 3.1 are instead commented to give an overview about what is programmed.

Listing 3.1: json.rs code modifications

```
extern crate systemstat;
use systemstat::{System, Platform};
...
fn retrieve_metrics(&mut self, parameters: &ArgMatches) {
...
    let sys = System::new();
    let host_average_load = sys.load_average().unwrap().one;
    let mut host_cpu_temp = 0.0;
    let host_mem = sys.memory().unwrap();
    let host_mem_free = host_mem.free;
    let host_mem_total = host_mem.total;
    let host_stat = match self.topology.get_stats_diff() {
        Some(value) => value,
```

[1] https://github.com/hubblo-org/scaphandre (accessed 30.09.2021)
[2] https://github.com/tstrempel/scaphandre (accessed 30.09.2021)

```
        None => return,
    };
    let host_cpu_load = host_stat.cputime.user + host_stat.cputime.nice
        + host_stat.cputime.system;

    let top_consumers = consumers
            .iter()
            .map(|(process, value)| {
                let host_time = host_stat.total_time_jiffies();
                Consumer {
                    exe: process.exe().unwrap_or_default(),
                    pid: process.pid,
                    consumption: ((*value as f32
                        / (host_time
                        * procfs::ticks_per_second().unwrap() as f32))
                        * host_power as f32),
                    memory: process.stat.rss,
                }
            })
            .collect::<Vec<_>>();
    let host_report = Host {
        consumption: host_power as f32,
        timestamp: host_timestamp.as_secs_f64(),
        average_load: host_average_load as f32,
        cpu_load: host_cpu_load as f32,
        cpu_temp: host_cpu_temp as f32,
        mem_total: host_mem_total.as_u64(),
        mem_free: host_mem_free.as_u64(),
    };
...
}
struct Consumer {
    exe: PathBuf,
    pid: i32,
    consumption: f32,
    memory: i64,
}
struct Host {
    consumption: f32,
    timestamp: f64,
    average_load: f32,
    cpu_load: f32,
    cpu_temp: f32,
    mem_total: u64,
    mem_free: u64,
}
```

The JSON exporter takes a `Consumer`, `Host` and `Socket struct` (see the end of listing 3.1) and serializes it into the JSON format (see listing 3.2). `Host` contains system wide metrics like total power consumption or total memory. Process specific metrics are included into `Consumer`. Values for energy domains, e. g. Core, Uncore and DRAM, are listed in `Socket`. The fields in each `struct` are set in the `retrieve_metrics` function by variables named after the fields with an `host_` prefix.

Rust provided its own package manager `cargo` (the equivalent to `pip` in Python) from which the package `systemstat` package is installed. *Systemstat* provides functions for implementing the load average and memory usage metrics. Other than that only functions from standard or already imported libraries are used.

Average load is simple to implement by using the `load_average()` function which returns a `LoadAverage` object containing the average load of the last minute. Only the one-minute average load is used for the JSON output. Later measurements showed that a period of one minute is too large to be useful for explaining energy consumption, intervals of a couple seconds are better suited. Therefore this metric is not mentioned further. Analogous to `load_average()`, the `memory()` function is used to gather the total memory of the computer system and the currently free memory.

The CPU load metric is already implemented elsewhere because Scaphandre uses the CPU load to calculate the energy consumption of a process. It is gained by computing the difference between the CPU time in a single Scaphandre time step and can be accessed via the `get_stats_diff()` function. Three different CPU time values are added to compute the total CPU time (`host_cpu_load`). This is due to how the Linux Operating System (OS) categorizes CPU time into user (processes run in the context of the user), system (system calls, everything done by the kernel) and nice (processes run with less priority in scheduling than others). Scaphandre already provides per-process memory metrics with `process.stat.rss`, this is just added as an extra field to the `Consumer struct`. Temperature measurement is handled in an extra section. Listing 3.2 shows the beautified JSON output of the modified Scaphandre. Compared with the original Scaphandre (see listing 2.3) the timestamp, average load, memory used per process, CPU load, CPU temperature, total and free memory are added.

Listing 3.2: Scaphandre JSON output (beautified)

```
$ scaphandre --no-header json --step 2 --timeout 2 --max-top-consumers=3 \
| js-beautify
{
    "host": {
        "consumption": 751756.0,
        "timestamp": 1630407144.7413516,
        "average_load": 0.06,
        "cpu_load": 0.14984131,
        "cpu_temp": 31.0,
        "mem_total": 8252932096,
        "mem_free": 3610116096
    },
    "consumers": [{
        "exe": "",
        "pid": 2521,
        "consumption": 100340.29,
        "memory": 5480
    }, {
        "exe": "",
        "pid": 2661,
        "consumption": 50170.145,
        "memory": 45292
    }, {
        "exe": "",
        "pid": 1,
        "consumption": 0.0,
        "memory": 3344
    }],
    "sockets": [{
        "id": 0,
        "consumption": 751861.0,
        "domains": [{
            "name": "dram",
            "consumption": 554204.0
        }, {
            "name": "core",
            "consumption": 207308.0
        }, {
            "name": "uncore",
            "consumption": 3737.0
        }]
    }]
}
```

3.1.1 Temperature measurement

It is important to monitor the CPU temperature during the experiments conducted in chapter 4. Due to DVFS the processor frequency, and thus voltage, is lowered on a high CPU temperature. If thermal buildup (high temperatures) from one measurement is carried over to another it could skew the results. Khan et al. [26] also showed that an increased temperature leads to increased energy consumption. To check if a thermal buildup is taking place, a thermal measurement was implemented into Scaphandre. Thermal measurements can be taken by reading out the right thermal zone under the path `/sys/class/thermal/`. Multiple zones were available on the development machine (see listing 3.3).

Listing 3.3: Temperature measurement

```
$ ls /sys/class/thermal
thermal_zone0  thermal_zone2
thermal_zone1  thermal_zone3
$ cat /sys/class/thermal/thermal_zone*/type
acpitz
pch_skylake
iwlwifi_1
x86_pkg_temp
```

The `systemstat` package has a function `cpu_temp()` which returns a temperature. But this function takes the first thermal zone, which is the wrong one. The `x86_pkg_temp` zone needs to be used instead. For that an own implementation was created (see listing 3.4).

Listing 3.4: Temperature measurement in json.rs

```
let mut host_cpu_temp = 0.0;
if let Ok(lines) = read_lines("/sys/class/thermal/thermal_zone3/temp") {
    // Consumes the iterator, returns an (optional) string
    for line in lines {
        if let Ok(tmp) = line {
            host_cpu_temp = tmp.parse::<f32>().unwrap() / 1000.0
        }
        break;
    }
}
```

These thermal zones are platform-dependent, which means on another PC `x86_pkg_temp` may correspond to `thermal_zone1` instead of `thermal_zone3`. Before working with temperature measurements on a new system, the correct thermal zone should be identified beforehand as shown in listing 3.3. This needs then to be changed in Scaphandre (see listing 3.4) followed by a rebuild. The fallback temperature is 0.0, since this is the value assigned to `host_cpu_temp` in listing 3.4. This value only changes when a valid temperature is read. Implementing a platform-independent method would fall outside the scope of this work. Plots created from the tempera-

ture data are later used to look for irregularities. Normal temperature ranges are listed in table 3.1.

Segment	Idle	Under load
Desktop CPU	45°C - 55°C	70 - 80°C
Laptop CPU	60°C	82 - 88°C

Table 3.1: Typical temperatures for different processors [31]

3.1.2 The importance of synchronization

Two implementations of measuring the CPU load were evaluated (see fig. 3.1). Standard metrics refer to the already implemented ones by Scaphandre, e. g. timestamp, total energy consumption, consumption per domain and the most energy-intensive processes. Added metrics refer to metrics added for this thesis, e. g. average load, CPU load, memory usage.

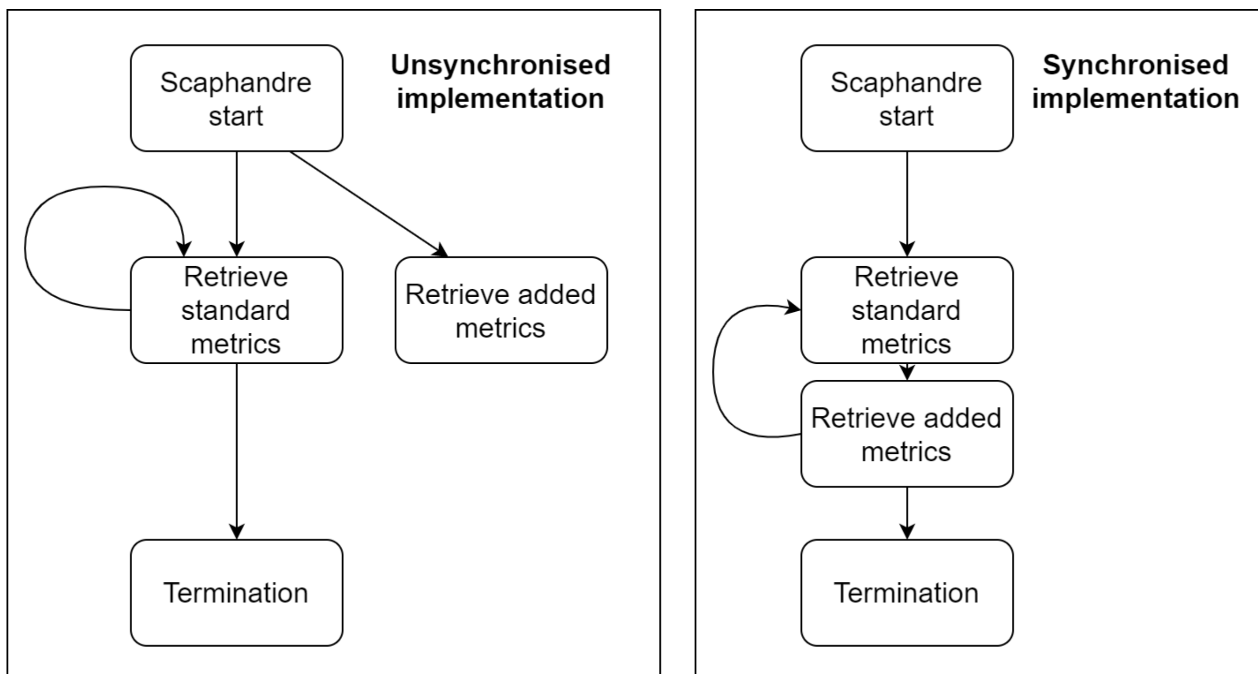


Figure 3.1: Visualization of the implementations

The first idea of implementation was to create an extra thread that retrieves the metrics independent from the standard metrics. This was needed since the *systemstat* rust library requires a one second long waiting period before a CPU load measurement could be taken. For example, an one second measuring period of Scaphandre and an extra one second period for the CPU load would add up to two seconds in total. Adding such a significant amount of time is unacceptable. Using an extra thread with its own timer, where this measurement takes place, would remedy the problem. The downside

of this is that measurements for the standard and added metrics now use two different timers. In later tests, it became clear that this implementation lead to unsynchronized timestamps. As can be seen in the last quarter of fig. 3.2, the spikes in CPU load do not coincide with the spikes in energy consumption. Instead, the CPU load spikes trail the energy consumption spikes. The `vpxenc` application, which will be examined in the next section, was measured to create these plots.

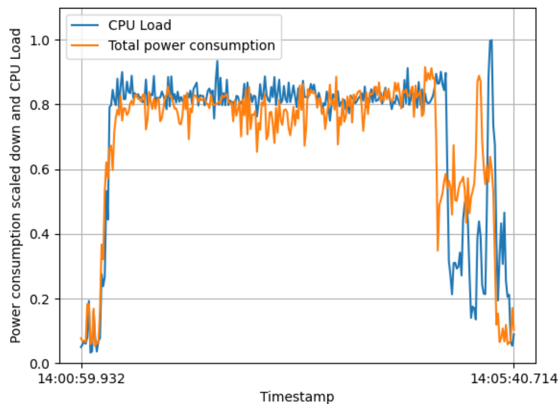


Figure 3.2: Graph of the unsynchronized implementation

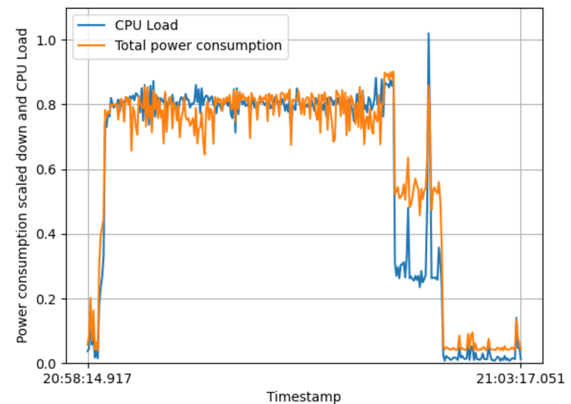


Figure 3.3: Graph of the synchronized implementation

This can be further showcased by looking at the data points of the graph (see fig. 3.4). It is to be expected that most points should lay on a straight line so that high CPU load corresponds to high energy consumption. Instead, there are a lot of irregular points spread out. Due to this problem, an alternative implementation was developed. The `systemstat` library will be retained for every added metric besides the CPU load. Scaphandre already uses the `procfs`[3] library for getting metrics on processes. There it is used to calculate the CPU time spend on a process and in total. The same results used there are also employed for delivering the CPU load metric in the new implementation. All the added measurements will be executed after the initial ones as shown in fig. 3.1.

[3] https://docs.rs/procfs/0.10.1/procfs/ (accessed 04.10.2021)

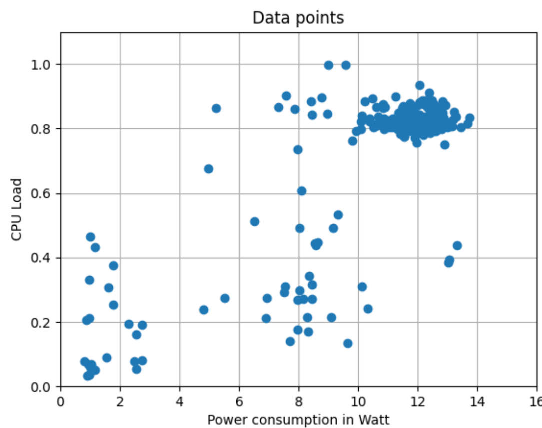


Figure 3.4: Data points of the unsynchronized implementation

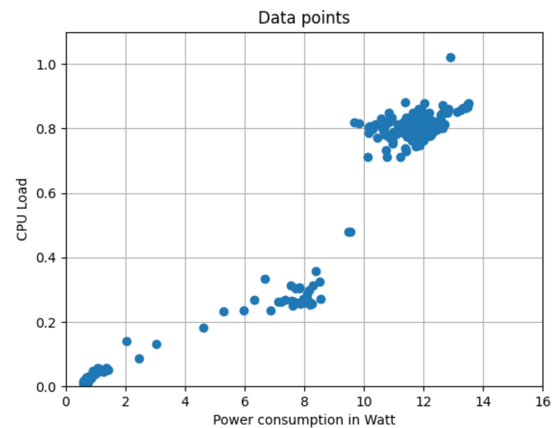


Figure 3.5: Data points of the synchronized implementation

Figure 3.3 now shows that spikes in CPU load and energy consumption coincide. By plotting the data points again (see fig. 3.5) this also becomes visible, because no outliers are visible. This example from development shows that properly synchronized timestamps are important to conducting experiments with energy consumption.

3.2 Test software

3.2.1 sorting

One fundamental question the thesis aims to answer is that whether algorithm design influences energy consumption. It shall be illustrated if and how energy can be saved by exchanging energy-intensive parts for less energy-intensive parts with the same function. Sorting algorithms are well suited for this use case because they all take the same input, an array of unsorted numbers, and deliver the same output, an array of sorted numbers. The principle of having exchangeable software components so that several programs can be created from it is further explored in SPLs.

Three sorting algorithms will be compared: selection sort, insertion sort and quick sort [32]. All three are self-programmed in C and stem from a programming assignment from 2017, they were checked for validity. These are used because they were already available and a good showcase for own algorithm development. For comparison, the quicksort function provided by the C standard libraries called `qsort()` will be used. The source code is hosted at GitHub[4] under the open-source *MIT* license. It would go beyond the scope of this thesis to explain the operation of these standard algorithms in depth. Important differences can be found in their time complexity (see tab. 3.2), which means the runtime of the algorithm in relation to the input size. Three cases can be distinguished based on the fact that the order of the input data of the

[4] https://github.com/tstrempel/sorting (accessed 09.10.2021)

algorithm can be advantageous or detrimental to its runtime: worst, average and best case.

Algorithm	**Worst-case**	**Best-case**	**Average-case**
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n^2)$	$O(n)$	$O(n^2)$
Quick sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$

Table 3.2: Time complexity of sorting algorithms

It is to be expected that selection sort will take the most time to run of the three algorithms because in every case it has a time complexity of $O(n^2)$. Insertion sort has a better best-case time complexity, its runtime will be comparable to selection sort since the average case remains the same. The average case of quicksort only takes $O(n \log n)$, a shorter runtime is to be expected.

The self-implemented sorting algorithms are published under the MIT license in a Github repository[5]. The code can be compiled by running the `make` program in the downloaded repository which contains the compiling instruction in a `Makefile`. `gcc` and `make` need to be installed for a successful compilation. Listing 3.5 shows how the program can be executed.

Listing 3.5: The sorting application

```
$ ./sorting <algorithm> <array_size>
$ ./sorting-O2 <algorithm> <array_size>
```

The type of algorithm can be selected by specifying a number for `<algorithm>`:

- 0: selection sort
- 1: insertion sort
- 2: quick sort
- 3: qsort

The array size can be freely chosen. One hundred arrays of the specified size will be created and filled by a random number generator. To enable reproducibility a fixed seed value of 42 is chosen for the random number generator. Sorting utilizes a single CPU core. The sorting functions are contained in the `sort_functions.h` file and are called `selection_sort`, `insertion_sort` and `quick_sort`. The creation of the arrays, parsing the user input and starting the calculations happen in `main.c`. Since most computation takes place in the sorting function it is to be expected that these also take the most time.

No compiler optimization is enabled at first so that only the efficiency of the code as it is can be taken. GCC can explicitly disable optimization via the *-O0* flag. When no optimization flags are provided GCC does not optimize the code, which is analogous to providing *-O0* [17]. A separate binary `sorting-O2` (see fig. 3.5) is created with the `-O2`

[5] https://github.com/tstrempel/sorting (accessed 04.10.2021)

flag. This is the recommended optimization flag [33] for most use cases. Comparing the results of sorting algorithms compiled with `-O0` or `-O2` is done to simulate which impact optimization has on energy consumption. This approach is chosen because optimizing a program manually is out of the scope of this thesis.

3.2.2 vpxenc

Besides `sorting` another application is introduced to have a bigger variety of applications. An application named `vpxenc`[6], which is used for video encoding, is chosen for the test. `vpxenc` is written in C and can utilize up to five CPU cores. The standard video encoding performance test is used. Since it is difficult to replicate a specific benchmark with all its setting a benchmark framework called *phoronix-test-suite* is used. After the installation of the Phoronix test suite [7] the benchmark can be installed and run (see listing 3.6).

Listing 3.6: vpxenc benchmark

```
$ phoronix-test-suite benchmark pts/vpxenc-3.1.0

...installation...

VP9 libvpx Encoding 1.10.0:
    pts/vpxenc-3.1.0
    Processor Test Configuration
        1: Speed 5 [Fastest - Default]
        2: Speed 0 [Slowest]
        3: Test All Options
        ** Multiple items can be selected, delimit by a comma. **
        Speed: 1

        1: Bosphorus 1080p
        2: Bosphorus 4K
        3: Test All Options
        ** Multiple items can be selected, delimit by a comma. **
        Input: 2

... overview of computer hardware and software settings ...
```

[6] `https://openbenchmarking.org/test/pts/vpxenc` (accessed 04.10.2021)
[7] `https://www.phoronix-test-suite.com/?k=downloads` (accessed 04.10.2021)

3.3 Python scripts

The programming language Python [34] is used to process the data provided by Scaphandre to calculate statistics and render plots. A GitHub repository[8] hosts the Python and Bash scripts. The script is tested to run under Python versions 3.8 and 3.9. The code is split into two files. `processing_functions.py` contains often used functions like reading in JSON files or processing the energy metrics. In `evaluation.py` the plots are generated together with important statistics of the data set, e. g. the correlation between CPU time and energy consumption. `jq`[9] is used as the JSON parser. Five input parameters are required (see listing 3.7), the input data (beautified JSON file), an output directory for the plots, the interval between measurements that is used by Scaphandre, the TDP and the number of threads of the processor and the name of an application of interest for which an extra plot and extra information is provided.

Listing 3.7: Executing the evaluation.py script

```
# argument 1: input file
# argument 2: output directory
# argument 3: scaphandre step size
# argument 4: TDP
# argument 5: application name
python evaluation.py data/energy_data_beautified.json data 1 15 4 sorting
```

Since all three platforms support different RAPL energy domains, each of those has a separate script. Due to that three directories (`laptop`, `desktop` and `server`) are created. An extra directory `legacy` is created for an old version of the Python scripts to work with older data where not all metrics, e. g. temperature or memory usage, are supported. Reading in old data with the normal `evaluation.py` script would lead to errors. The same was done for the server energy measurements (`server` directory), since the RAPL Uncore domain was missing and two processors were used simultaneously (dual socket system).

[8] https://github.com/tstrempel/masterthesis-code (accessed 10.10.2021)
[9] https://stedolan.github.io/jq/ (accessed 10.10.2021)

3.4 Bash scripts

A major drawback of Scaphandre is that only a timeout, which means the time Scaphandre runs before termination, can be specified. There is no built-in way to run Scaphandre alongside another program. Therefore a Bash program (see listing 3.8) was written. Bash is an interpreted language used in Linux environments. The main idea behind writing the script is to start the Scaphandre process in the background and save its process ID provided by Linux, so the process can be killed later. Running a command in the background via the & operator makes it possible to execute other commands after it without needing to wait until the previous command has finished. `$!` returns the process ID of the previously executed program. Waiting periods are introduced via `sleep` to give the processor time to cool down in between measurements.

Listing 3.8: Excerpt from wrapper.sh

```
# sleep 120 seconds
sleep 120
mkdir "data"

# start scaphandre in the background for a long time period (one day)
$ scaphandre --no-header json --timeout 86400 --step 1 --step_nano 0 \
   --max-top-consumers=200 > data/energy_data.json &
# get the processid of the running scaphandre
$ processid_scaphandre=$!
# wait some time
$ sleep 5
# start the application which should be measured
$ { time sorting 0 32000; } 2> data/log.txt &
# get the processid of the running application
$ processid_sorting=$!

# send the script into an endless queue until the application is done
$ while [ -d "/proc/$processid_sorting" ]; do sleep 1; done
# wait some time
$ sleep 1
# and then kill the scaphandre process
$ kill $processid_scaphandre

$ nice js-beautify "data/energy_data.json" > "data/energy_data_beautified.json"
$ python evaluation.py "data/energy_data_beautified.json" "data" 1 15 4 '
   sorting' > "data/stats.txt"
```

After collecting the Scaphandre data in `energy_data.json` it needs to be restructured ("beautified") for the Python script. This can be achieved with the `js-beautify` program (see listing 3.8). After that, the Python script described in the previous section is used to create statistics and plots. The Bash script `wrapper.sh` encapsulates the code in listing 3.8 into a function which can be executed with different parameters.

This enables running `sorting` measurements with different parameters (algorithm and array size) without pasting the code multiple times. `wrapper-O2.sh` contains the same program logic but adapted for the `sorting-O2` application.

The `time` command [35] reports the clock and CPU time a program spent. Alternatively, VTune, which will be explained in the next section, can be used. The data collected in this script is written to multiple files via the `> output_file.txt` pipeline. `log.txt` contains the output of the `time` command whereas `stats.txt` contains the computed key figures like how much energy was consumed by `sorting`. These values are then manually written into the `masterthesis.ods` file. This wasn't automated by an export to CSV because it was judged to take more time to automate than it would safe. Parsing output of applications like VTune would also add to this.

Listing 3.9: The time command (CPU time is user and sys combined)

```
# time prefixes the original command
$ time sorting 0 4000
real    0m0.618s
user    0m0.614s
sys     0m0.003s
```

3.5 Intel Vtune

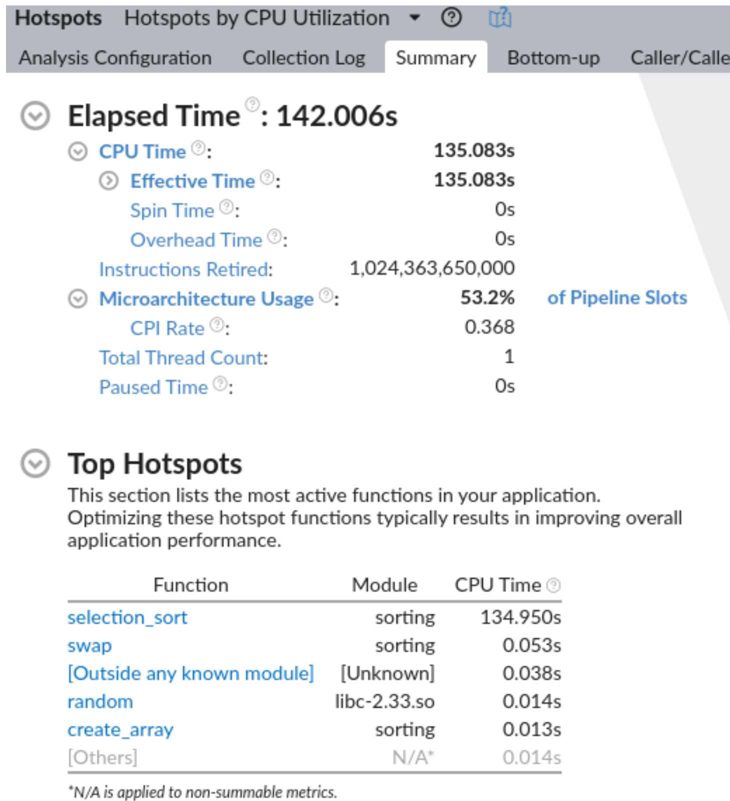


Figure 3.6: VTune summary

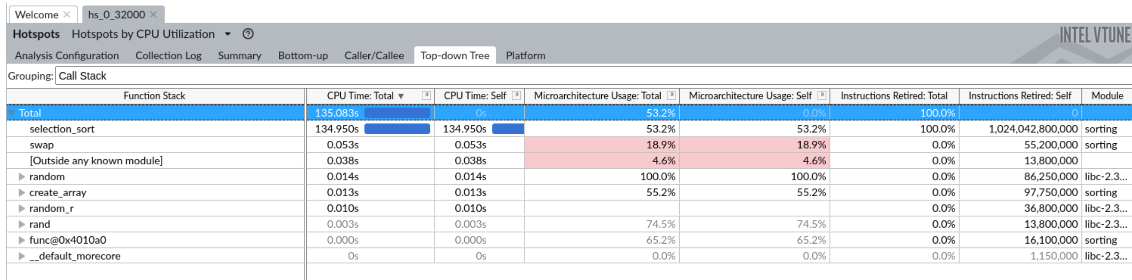
VTune [36] is a profiler developed by Intel. A profiler analyses an application at runtime. VTune supports a wide variety of programming languages (C is also included) and functionalities, e. g. finding bottlenecks or reports on the utilization of the processor. A big drawback of VTune is that only Intel CPUs are supported. This makes it impossible to use it with the AMD desktop PC. On the server system, there were a couple of problems with the installation which would need changes to the Linux OS configuration. This was not possible due to being a borrowed system. Other important features are an overview of how much CPU time was spend in each function (see fig. 3.7), the CPI rate and microarchitecture usage. The Cycles per Instruction Retired (CPI) rate approximates how long one instruction takes, in units of processor cycles. A CPI of one is considered acceptable, the lower the CPI the better since more instructions can take place in one cycle. CPI is a metric *"for judging an overall potential for application performance tuning"* [37]. Mircoarchitecture usage describes (in %) how many features of the processor are used by the application [38]. For example, using processor vector operations with Advanced Vector Extensions (AVX)[10] increases the mircoarchitecture usage. Since the `-O2` option activates such features, the programs compiled with it have a higher mircoarchitecture usage than not optimized ones. VTune also includes functionalities to optimize the program for a lower runtime and parallelism. An idea for a future work would be to compare the energy consumption of a program before and after optimization via VTune, since this is outside the scope of this thesis.

[10] https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html (accessed 10.10.2021)

Listing 3.10: Scaphandre

```
$ vtune -collect hotspots -knob sampling-mode=hw -knob sampling-interval=0.5 -r
    selection_sort_32000/hs_0_32000 sorting 0 32000

$ vtune-gui laptop-O0/selection_sort_32000/hs_0_32000/
```



Function Stack	CPU Time: Total	CPU Time: Self	Microarchitecture Usage: Total	Microarchitecture Usage: Self	Instructions Retired: Total	Instructions Retired: Self	Module
Total	135.083s	0s	53.2%	0.0%	100.0%	0	
selection_sort	134.950s	134.950s	53.2%	53.2%	100.0%	1,024,042,800,000	sorting
swap	0.053s	0.053s	18.9%	18.9%	0.0%	55,200,000	sorting
[Outside any known module]	0.038s	0.038s	4.6%	4.6%	0.0%	13,800,000	
random	0.014s	0.014s	100.0%	100.0%	0.0%	86,250,000	libc-2.3...
create_array	0.013s	0.013s	55.2%	55.2%	0.0%	97,750,000	sorting
random_r	0.010s	0.010s			0.0%	36,800,000	libc-2.3...
rand	0.003s	0.003s	74.5%	74.5%	0.0%	13,800,000	libc-2.3...
func@0x4010a0	0.000s	0.000s	65.2%	65.2%	0.0%	16,100,000	sorting
__default_morecore	0s	0s	0.0%	0.0%	0.0%	1,150,000	libc-2.3...

Figure 3.7: VTune top-down tree

An alternative to using VTune is gprof which also has the CPU time per function feature but has a high overhead (30-260%) [39]. VTune has built-in features to reduce overhead [40]. When only the CPU time spent of a program is needed the `time` program [35] can be used instead.

3.6 Correlation between CPU load and energy consumption

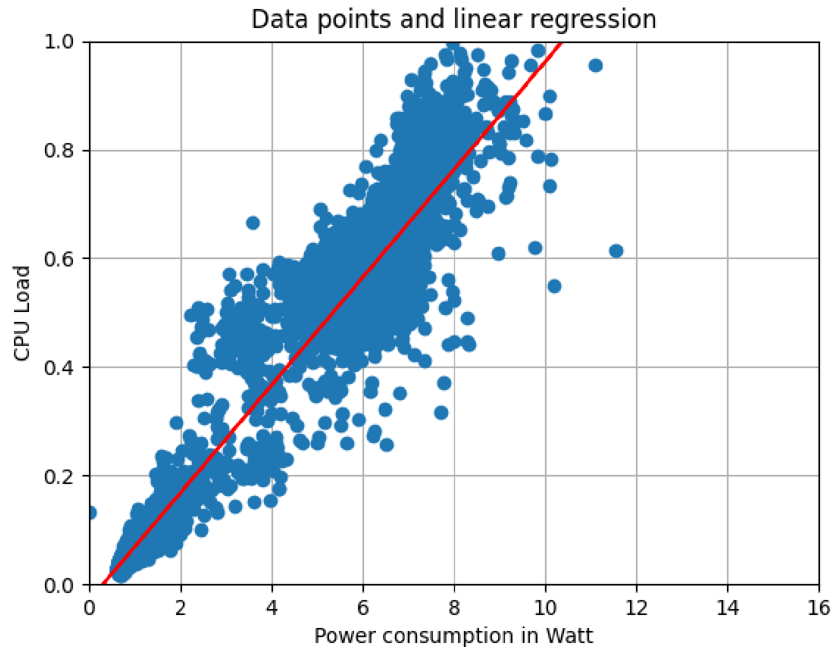


Figure 3.8: Data points with linear regression line

One of the questions to be answered in this thesis is if the energy consumption is correlated to the CPU load. If so the CPU load can be used in place of the real energy consumption for energy optimization. Since it is assumed that the relation between CPU load and the energy consumption is linear, linear regression is used. The Python package `scipy` provides the function `stats.linregress`[11] for this. The data set used consists of a three hour long measurement of work activities, e. g. browser usage, software development or video conferences. With this mixture of idle, middle and heavy workloads a linear regression is made (see fig. 3.8).

The determination coefficient R^2 (R-squared) indicates how much variance for a dependent variable (the energy consumption) is explained by an independent variable (CPU load) in a regression model. Using R^2 instead of R leads to a higher weighting of outliers far away from the regression line. The resulting R^2 value of 0.95 (see 3.11) is excellent. Only 5 percent of the variance can not be explained by the CPU load. Every measurement taken in this work had its R value, from which R^2 can be calculated, calculated and saved into the corresponding `log.txt` or `stats.txt` file.

[11] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.linregress.html (accessed 10.10.2021)

Listing 3.11: Printed out linear regression results

```
# in evaluation.py
print("Linregress:")
linregress = stats.linregress(energy_data['consumption'], energy_data['cpu_load
    '])
print(linregress)

# in output file (log.txt or stats.txt)
Linregress:
LinregressResult(slope=0.39632862733135393, intercept=-0.12055882254904082,
rvalue=0.9754433456052097, pvalue=0.0, stderr=0.0011588784223008895,
intercept_stderr=0.004678310356445275)

# R^2
0.95148972048548457375517178097409
```

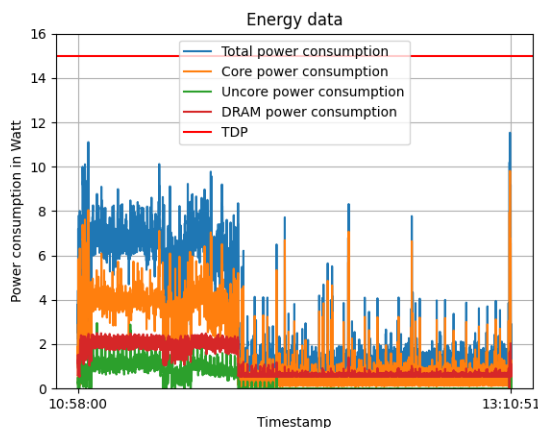


Figure 3.9: Energy domains

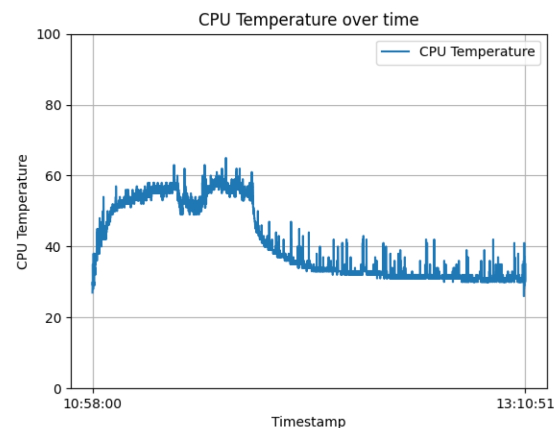


Figure 3.10: Temperature

`Vpxenc` has a R^2 value of 0.994, while the worst value in a `sorting` measurement encountered in this thesis was 0.716 (selection sort with an array size of 256,000). With that it is shown that energy consumption and CPU load strongly correlate. The created linear regression is platform-specific and partly software specific because the maximal amount of energy (TDP) can be drawn at a partial utilization of Core, Uncore and DRAM or at the maximal utilization of core (see fig. 3.9). The unexplained variance can stem from faulty measurements, CPU temperature (see fig. 3.10), memory utilization and code efficiency.

3.7 Validity of Scaphandre

In this section, the validity of Scaphandre (and therefore RAPL) is tested further to supplement the findings from other works (see section 2.7). There is also a section in the next chapter dedicated to checking Scaphandre's internal models, using data collected in the next chapter.

3.7.1 TDP showcase

Since it is difficult to estimate if measurements are plausible without prior experience or a frame of reference, a known external factor should be used for validation. In this case, the TDP is used. If the power consumption of the processor under maximal load, i. e. all processor cores are fully utilized, matches its TDP, it would support the assumption that the measurements are correct and set a frame of reference. A similar approach of checking the validity of power limits was used by Zhang et al. [29]. Running the processor under full load is also called "stress testing" the processor and is used in other fields to determine the stability, e. g. thermal throttling or system shutdowns, under excessive load. Prime95[12] is a popular program for stress testing and is easy to use, therefore it will be used in this work.

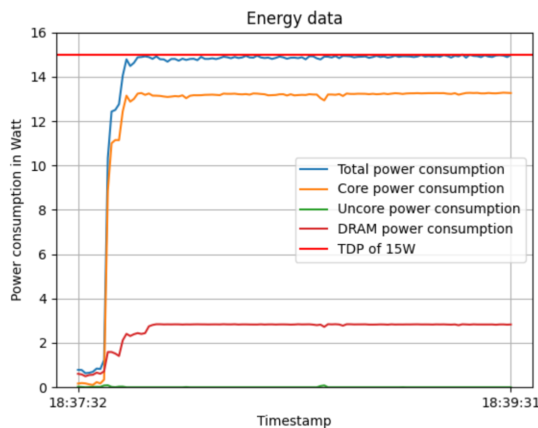


Figure 3.11: Energy consumption of RAPL domains

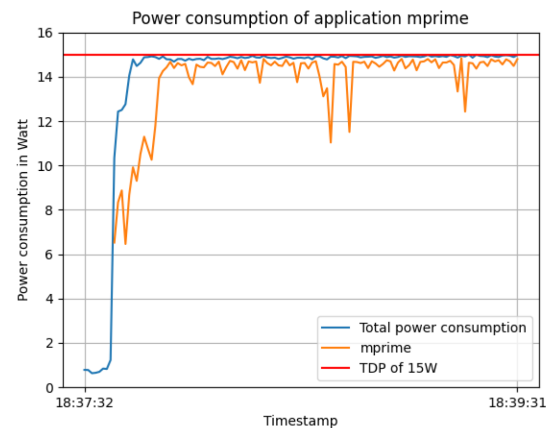


Figure 3.12: Power consumption of mprime (Prime95)

Fig. 3.11 shows the power consumption broken down into four RAPL domains (package/total, Core, Uncore, DRAM). It becomes visible that the greatest share of power consumption is in the Core domain, followed by the DRAM. Almost the entire main memory is used, which in turn leads to the maximal DRAM power consumption of around 2.92W. The Uncore power consumption is close to zero, because Prime95 does not do heavy IO or graphical workloads, it only stresses the CPU cores and the main memory. There are other dedicated programs for stress testing IO and graphics, which are outside of this thesis as it is limited to non-graphical applications.

[12] https://www.mersenne.org/ (accessed 04.10.2021)

The TDP of 15W is marked in the figures. As can be seen in figure 3.12 under a full stress test from the *mprime* (from Prime95) program the total power consumption is close to the TDP of 15W. With this, it is shown that, since it revolves around the reference point of the TDP, the RAPL measurements are plausible.

3.7.2 Cross-validation with PowerTOP

As it was stated in section 2.6.3, PowerTOP can not correctly measure the energy consumption of applications that do not change the power state of the CPU cores they are running on. An example for such an application is the `sorting` application because no energy consumption is shown for it by PowerTOP. Therefore `vpxenc` which was introduced in section 3.2.2 is used instead.

Three power measurements are taken with PowerTOP, which has an average of 8.95W. Scaphandre reports an energy consumption of 10.83W for the same workload. While 21% is a large difference, the reported power consumptions are not orders of magnitude apart from each other. This lends further credibility to the validity of RAPL measurements because two tools with differing techniques report roughly similar values. The associated calculations for this section can be found in the `PowerTOP` table in the `masterthesis.ods` file.

Not every measurement returns an entry specific to `vpxenc`, thus these are discarded. This means that PowerTOP provides a poor reliability and reproducibility and is not recommended for measuring single applications.

Chapter 4

Energy Consumption Comparison of Sorting Algorithms

In this chapter it will be illustrated how energy consumption behaves with different sorting algorithms and on different processors. It is similar to an experiment conducted by Johann et al. [28], where the topic was also the energy consumption of sorting algorithms.

4.1 Experiment

In the experiment the `sorting` program described in section 3.2.1 is used in different configurations (see table 4.1). The array sizes for the two quick sort implementations are far higher because of the lower time complexity of the quicksort algorithm. Therefore larger arrays can be sorted more quickly. Comparison between these and selection and insertion sort is enabled by an overlap in the array sizes of 128,000 and 256,000. Practical experience showed that trying to measure the energy programs with runtimes shorter than a couple of seconds is either not possible or inaccurate. Therefore no energy consumption is listed in these cases.

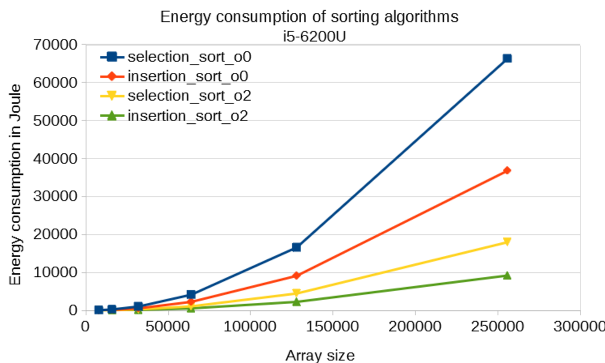


Figure 4.1: Energy consumption of selection and insertion sort

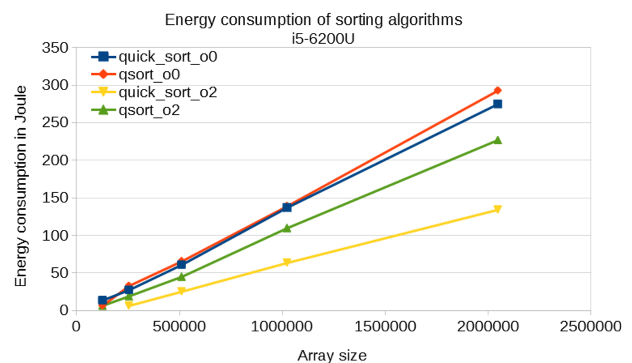


Figure 4.2: Energy consumption of the two quicksort implementations

Algorithm	Array size	Runtime	CPU time	Energy consumption
Selection sort	1,000	0.163s	0.162s	NA
	2,000	0.605s	0.604s	NA
	4,000	2.37s	2.366s	NA
	8,000	9.469s	9.008s	56.988J
	16,000	36.767s	34.956s	255.198J
	32,000	142.006s	135.083s	1016.562J
	64,000	558.261s	530.764s	4135.403J
	128,000	2223.103s	2113.553s	16592.252J
	256,000	8862.997s	8436.781s	66328.557J
Insertion sort	1,000	0.085s	0.082s	NA
	2,000	0.327s	0.326s	NA
	4,000	1.283s	1.281s	NA
	8,000	5.295s	5.036s	26.488J
	16,000	21.124s	20.057s	126.855J
	32,000	84.194s	80.169s	547.016J
	64,000	336.282s	320.082s	2245.272J
	128,000	1345.458s	1280.139s	9112.377J
	256,000	5387.913s	5123s	36804.944J
Quick sort	32,000	0.565s	0.563s	NA
	64,000	1.177s	1.75s	NA
	128,000	2.521s	2.381s	13.511J
	256,000	5.136s	4.877s	27.087J
	512,000	10.353s	9.853s	60.677J
	1,024,000	20.827s	19.862s	136.805J
	2,048,000	41.86s	39.891s	274.618J
qsort	32,000	0.558s	NA	NA
	64,000	1.187s	1.184s	NA
	128,000	2.616s	2.465s	6.399J
	256,000	5.466s	5.165s	32.639J
	512,000	11.267s	10.691s	65.356J
	1,024,000	23.034s	21.936s	138.721J
	2,048,000	46.94s	44.749s	292.595J

Table 4.1: Measurements taken for sorting (`-O0`) algorithms (NA = not available) on the laptop system (i5-6200U)

Two plots were created to showcase the energy consumption (see table 4.1) of the previously described sorting algorithms. Selection and insertion sort were grouped into one plot while the two quicksort implementations were grouped into another to achieve a proper scaling, e. g. selection sort with an array size of 128,000 takes 16592J to complete and quicksort only takes 13.5J. Behavior like this is explained by the differing average time complexity for these algorithms, $O(n^2)$ for selection and insertion sort and $O(n \log n)$ for the quicksort implementations. The drastically longer run times (see tab. 4.1) also lead to larger energy consumption. In figure 4.1 the energy con-

sumption curve resembles a quadratic function, the same as the time complexity curve. Therefore the term *energy complexity* will be used to describe the amount of energy an algorithm uses in relation to its input size, e. g. the array size in `sorting` or the length and resolution of a video in `vpxenc`. This new metric, like time complexity, can not deliver specific values because it is platform depended. It remains to be seen in future works if the energy complexity always resembles the time complexity.

Figure 4.2 closely resembles a linear function, since on larger input sizes the relative influence of the logarithmic portion of $O(n \log n)$ diminishes. In both figures, the optimized algorithms (compiled with the `-O2` flag) perform significantly better than the not optimized ones (`-O0` flag).

In both figures it also becomes visible, that insertion sort outperforms selection sort even though they have the same average time complexity. The self-implemented quicksort algorithm performs slightly better in the experiment than the provided `qsort` one. This showcases that the implementation of a specific algorithm and its data can influence its runtime and energy consumption. For example, a program using `quick_sort` instead of `qsort` and sorting 100 arrays of a size of 256,000 would save 5.6J. This is only a slight improvement since the laptop's processor can consume up to 15J per second. But changing from `selection_sort` to `quick_sort` would save 66,301 Joule and a lot of runtime. This circumstance illustrates the need to develop efficient algorithms not only to reduce runtime but also to reduce energy consumption. All figures and the data they are based on can be viewed in the `masterthesis.odt` file in the `sorting` table.

4.2 Temperature during the measurement

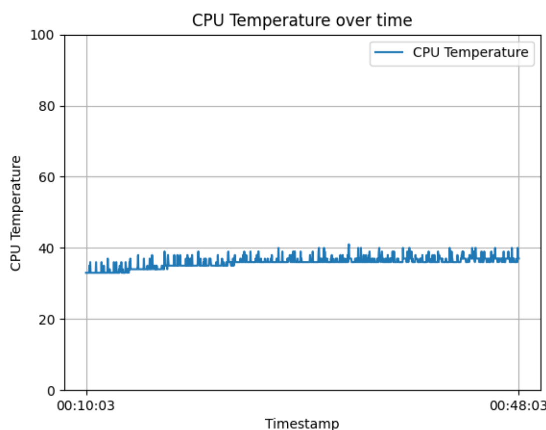


Figure 4.3: CPU temperature during selection sort with an array size of 128,000

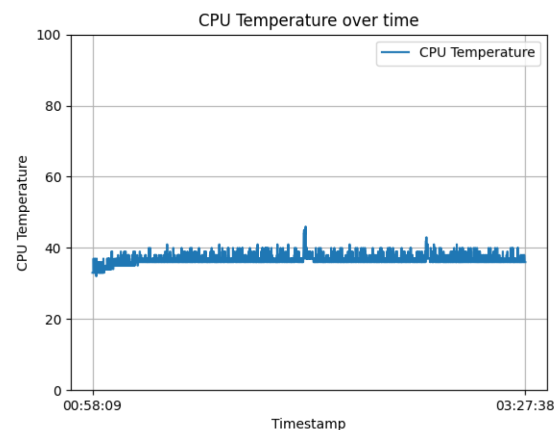


Figure 4.4: CPU temperature during selection sort with an array size of 256,000

Long and heavy CPU utilization combined with the small cooling systems of laptops thus leads to high temperatures. To monitor this, every benchmark run on the laptop platform has temperature metrics enabled. Temperature plots are created for every

single benchmark to enable checking for high temperatures. A two minute gap was used between consecutive benchmarks. The most likely place for a thermal buildup is between two long-running benchmarks. Figures 4.3 and 4.4 show such a point. As can be seen, the light thermal buildup made in 4.3 has dissipated in the next benchmark. Therefore a gap of 2 minutes for cooling is judged to be sufficient. The temperature never went high enough to be of concern (see table 3.1).

4.3 Reproducibility

Reproducibility means executing the same workflow, in this case measuring a program's energy consumption, and getting the same results every time, e. g. the same energy consumption. This can only be done in approximation, as it was stated in section 2.7 that measurements are subject to a certain degree of uncertainty. The Coefficient of Variation (CV), or relative standard deviation, is defined as the normalized dispersion. By scaling with the average value μ it becomes possible to compare the deviation across measurements with different array sizes.

$$CV = \frac{\sigma}{\mu}$$

For this test the `sorting` applications, specifically `selection_sort` (with an array size of 8,000 and 16,000) and `quick_sort` (with an array size of 128000), are used. Each of these programs are executed and measured ten times (with a two minute offset between measurements). The CV is then computed with the mean and standard deviation taken over the ten measurements. Results are listed in table 4.2, also see the `reproducibility` sheet in `masterthesis.ods` for further information. The CV stays below 1% in all three cases, this amount of deviation is deemed acceptable. Therefore reproducibility is achieved.

Algorithm	**Array size**	**Coefficient of variation**
Selection sort	8,000	0.8%
Selection sort	16,000	0.35%
Quick sort	128,000	0.86%

Table 4.2: Variation coefficients

4.4 Differences on desktop and server platforms

Conclusions in the previous sections are based on the measurements taken on the laptop system. All sorting benchmarks were also conducted in the desktop and server platform. Process-based measurements are not possible on the server system, the probable cause for this is the old Linux kernel version on it. Instead the idle (see tab. 4.3) power consumption times the amount of measurements is subtracted from the total energy consumption to get the `sorting` energy consumption.

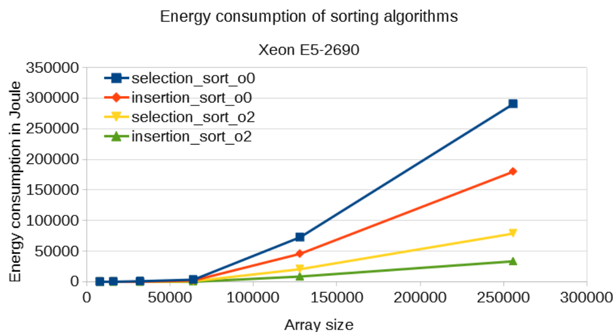


Figure 4.5: selection and insertion sort on the server system

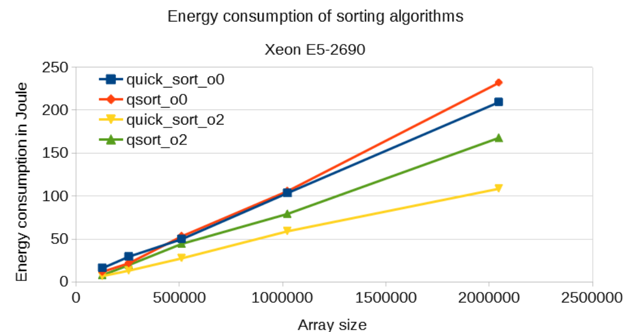


Figure 4.6: quicksort and qsort on the server system

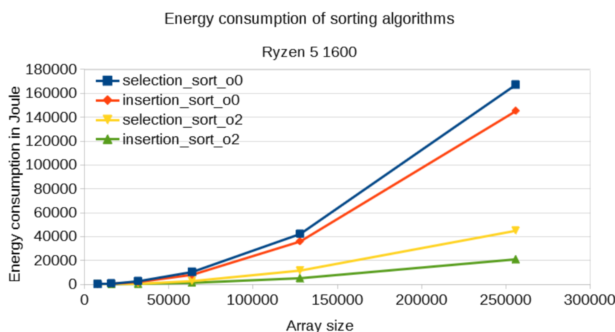


Figure 4.7: selection and insertion sort on the desktop system

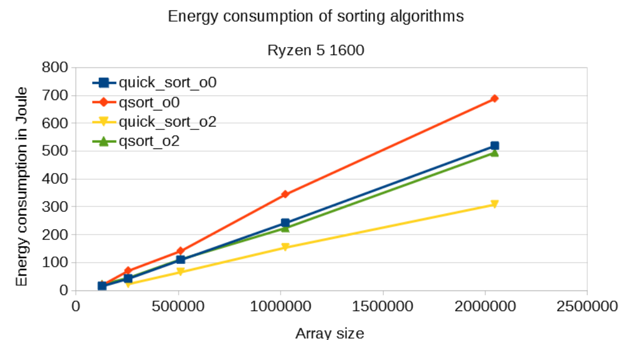


Figure 4.8: quicksort and qsort on the desktop system

The energy complexity, $O(n^2)$ (see fig. 4.5 and 4.7) and $O(n \log n)$ (see fig. 4.6 and 4.8), on the desktop and server systems remain the same as on the laptop system. This confirms the platform independence of energy complexity. Ranked by energy efficiency, the laptop platform performs best followed by the desktop and server platforms. But the computation on the laptop also took the longest amount of time while the server system was the quickest. This is to be expected since the *i5-6200U* processor the laptop uses is developed to be more energy-efficient and thus slower than normal processors. All plots and calculations for this section can be found in the `sorting` table of the `masterthesis.ods` file.

4.5 Validation of process based Scaphandre measurements

Validation of Scaphandre's internal model for process-level energy consumption metrics can be achieved by first measuring the idle power consumption of the system. Then the program is run and the difference between that run and idle is calculated. Both runs are measured with Scaphandre.

Processor	**Idle power draw**
i5-6200U	0.55W
Ryzen 5 1600	12.96W
2x Xeon E5-2690 v4	70W

Table 4.3: Idle power draw

The idle power consumption is determined by running Scaphandre on a freshly started computer to minimize the power draw of background processes. This takes place over 10min with one measurement every second. From the resulting data set, the median total power consumption is computed and used as the idle power draw. Median is used instead of the average to minimize the influence of outliers, e. g. power consumption spikes due to a program starting. This idle power draw is compared with the remaining power draw. If the remaining power draw matched the idle power draw Scaphandre's process-based measurements are valid. In section 2.7 it was established that RAPL is less accurate near the idle load. Therefore an exact match can not be expected.

$$RemainingPowerDraw = \frac{TotalConsumedEnergy - SortingConsumedEnergy}{MeasurementsTaken \cdot Interval}$$

The remaining power draw is calculated for every platform and every optimization level (`-O0` and `-O2`) separately. Interval is defined as the time between two measurements. An interval of one second is used as the default in all measurements in this thesis. The only exceptions are the measurements for selection and insertion sort with an array size of 128,000 and 256,000 where two seconds are used as the interval. Table 4.4 shows that on the laptop processor the remaining power draw is almost equal to the idle power draw (see table 4.3). With optimization, the remaining power draw is higher than previously. Possible causes for this are processor features, e. g. AVX, enabled by the `-O2` flag. Since the code got more energy efficient a bigger remaining power draw remains.

Processor	Flag	Remaining power draw
i5-6200U	-O0	0.56W
	-O2	0.75W
Ryzen 5 1600	-O0	7.72W
	-O2	7.04W

Table 4.4: Remaining power draw (see `masterthesis.ods`)

The server system is not mentioned since the idle consumption was used to compute the energy consumption of `sorting` because it could not be measured otherwise. On the Ryzen processor, the remaining power draw is a lot smaller than the idle power draw. Causes for this can not be stated since no evaluation of the RAPL implementation on AMD Ryzen was available at the time of writing. Regardless of that, since the main system of this thesis (a i5-6200U based laptop) is valid, the findings can be also considered valid. For other platforms, a larger margin of error needs to be expected.

4.6 Achieving function-level granularity

The previous sections established that CPU time strongly correlates to energy consumption. Therefore the energy consumption of functions can be approximated:

$$FunctionEnergyConsumption = \frac{FunctionCPUTime}{TotalCPUTime} \cdot TotalEnergyConsumption$$

Using this method the function energy consumption of `vpxenc` (see table 4.5) and `sorting` (see table 4.6) are calculated. VTune (see section 3.5) is used to determine to CPU time spend per function.

Function	CPU Time	Consumed Energy
Total	607.108s	2297.78J
vpx_sad64x64_avx2	101.251s	383.21J
vpx_sad32x32_avx2	62.941s	238.22J
vp9_pattern_search	32.175s	121.78J
sub_pix_var32	31.14s	117.86J
vp9_rd_pick_inter_mode_sb	29.28s	110.82J
The rest	350.321s	1325.89J

Table 4.5: vpxenc function energy consumption

Table 4.5 shows that small functions must sometimes be taken into account when trying to optimize energy consumption. Over half the CPU Time of `vpxenc` was spend in small functions (5% or less of the total CPU Time). In contrast the `sorting` application spends almost all CPU time in the function `selection_sort`. Therefore the decision to take small functions into account must be made depending on the application.

Function	CPU Time	Consumed Energy
Total	135.083s	1016.56J
selection_sort	134.95s	1015.56J
swap	0.053s	0.4J
unknown	0.038s	0.29J
random	0.014s	0.11J
create_array	0.013s	0.1J
The rest	0.014s	0.1J

Table 4.6: sorting (selection sort (with an array size of 32000) function energy consumption

The validation can only take place by measuring the energy consumption differences between two otherwise similar programs where one function was exchanged for another. This will need a more sophisticated SPL based approach to verify. The most ideal application would use an established community benchmark, the CPU load would be concentrated in a couple of big functions, which is not the case with `sorting`, and these need to be easy to swap with others, which is not the case with `vpxenc`. No such application was found after an initial search and developing one would be outside of this work.

Chapter 5

Reproduction and Replication Packages

To improve the quality of scientific work, findings need to be able to be reproduced. It is important to provide a bundle or package of software called a reproduction package, for enabling other researchers to understand, check and possibly enhance the results. Reproducing without appropriate provisions by the original developers is difficult and oftentimes impossible. This chapter explains the thoughts, reasons and approaches that make up the reproduction package of this thesis. The goal is to reliably reproduce every finding, statistical test and plot made in this thesis.

Krafczyk et al. [41] differentiates between reproduction and replication packages. Reproducibility is being defined as *"Obtaining consistent computational results using the same input data, computational steps, methods, code, and conditions of analysis"* [41, Page 2] while replicability means *"Obtaining consistent results across studies aimed at answering the same scientific question, each of which has obtained its own data."* [41, Page 2]. With that, the main distinction between them lies in the source of the input data. Researchers must thus publish their data to enable reproducibility or describe methods to acquire new data for replication.

While it may seem better to always aim for replicability, it cannot always be assumed to be possible. Experimental setups in the natural sciences can be so extensive that rebuilding a similar one is not viable (e. g. particle accelerator). Enabling replicability for this thesis would mean providing the same programs and instructions used to get the metrics shown in the previous chapters to obtain a new set of input data. It was decided to provide a replication package as one is needed anyway to make measurements on multiple systems. Platform-dependent parameters such as the processor's TDP and core count must be found out beforehand and provided as arguments. A reproduction package is also provided to recreate key figures and plots from the JSON data.

Principles and more direct practical guidelines for making a reproduction package were provided by Krafczyk et al. [41] and were shortened to their essence for this work.

Reproduction principles: [41]

1. Provide transparency regarding how computational results are produced.
2. When writing and releasing research code, aim for ease of (re-)executability
3. Make any code upon which the results rely as deterministic as possible.

Reproduction guidelines: [41]

1. Make all artifacts that support published results available, up to legal and ethical barriers.
2. Connect published scientific claims to the underlying computational steps and data
3. Specify versions and unique persistent identifiers for all artifacts.
4. Declare software dependencies and their versions
5. Refrain from using hard coded parameters in code
6. Avoid using absolute or hard-coded file paths in code.
7. Provide clear mechanisms to set and report random seed values
8. Report expected errors and tolerances with any published result that include any uncertainty from software or computational environments
9. Give implementations for any competing approaches or methods relied upon in the article.
10. Use build systems for complex software.
11. Provide scripts to reproduce visualizations of results.
12. Disclose resource requirements for computational experiments.

Every reproduction guideline was taken into account in this and previous chapters, e. g. a fixed seed value for sorting, explaining program parameters, always using relative paths.

5.1 Docker

Docker containers are the de facto standard for reproduction packages and are highly recommended for reproduction in the scientific community [41]. They can encapsulate the used program from the underlying OS. This is achieved by using OS functionalities to monitor and isolate resource usage (processor, memory etc.) for a process, to produce an isolated environment to run a process separated from the rest of the system. Operating systems need built-in support for the technologies used by Docker which all modern versions of Windows and Linux provide. To run the reproduction and replication packages a docker installation[1], which can run as a non-root user[2], is required.

[1] https://docs.docker.com/get-docker/ (accessed 30.09.2021)
[2] https://docs.docker.com/engine/install/linux-postinstall/ (accessed 30.09.2021)

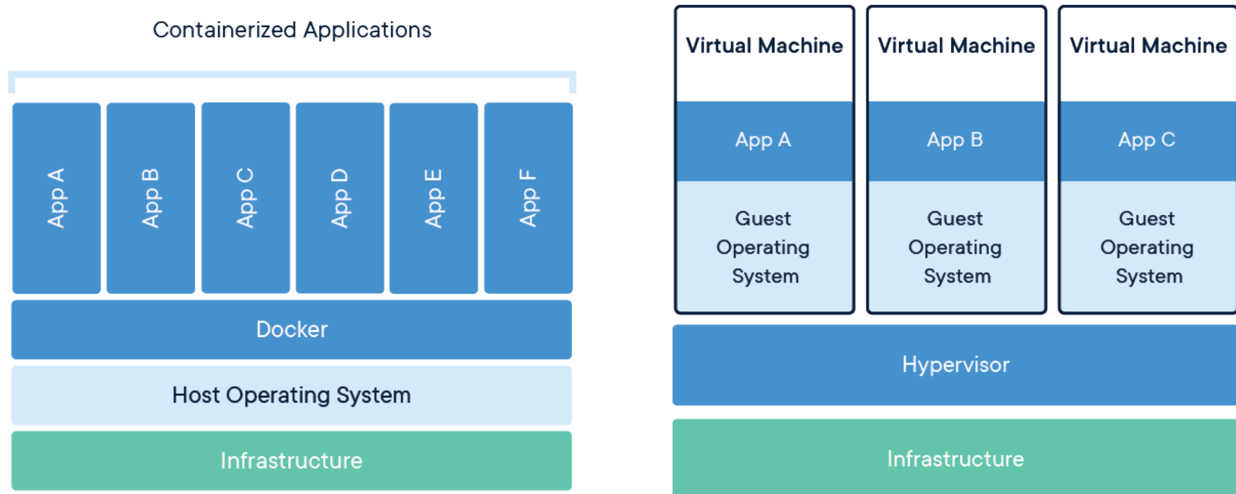


Figure 5.1: Structure of Docker and virtual machine based systems [42]

Alternatively to using Docker a Virtual Machine (VM) image could be provided. Unlike Docker, VMs require running a fully featured guest OS. This makes Docker Images smaller, faster and easier to share in comparison. One further alternative to Docker would have been to provide a script running all necessary commands in the right order with the right parameters. Scripts have the disadvantage that they are not platform-independent, e. g. Windows uses Powershell as the standard scripting language while Linux uses Bash. Thus Docker will be used as the basis for the reproduction package due to its simplicity and platform independence. All created Docker images are publicly available on DockerHub[3].

5.2 Reproduction package

The first step in creating a reproduction package is stating which results should be reproduced from which input data. VTune and PowerTOP automatically create reports after their execution. No modifications are made to these programs for this thesis. Therefore they can be excluded from the reproduction package, screenshots and result files of these are provided in `complete-data.zip`. Reproducing the plots and key figures from the JSON data collected by Scaphandre is the main focus of this package. The `Dockerfile`, a recipe for building the Docker image, is provided in a GitHub repository[4]. Python and Bash scripts needed for producing plots and key figures are also included. The following steps are programmed (also see the `Dockerfile`):

[3]https://hub.docker.com/ (accessed 09.10.2021)
[4]https://github.com/tstrempel/masterthesis-code (accessed 06.10.2021)

1. Provide base image (Fedora Linux)
2. Install tools and Python
3. Download scripts from GitHub
4. Install Python packages based on the bundled requirements.txt
5. Download data[5]
6. Run the reproduction and python scripts

For writing own Docker images a base image is required. The most used base images are official images of Linux distribution provided by their developers, these only include the most basic set of software to keep the storage requirements low. Because the development system used for this thesis was Fedora Linux[6], it will also be used as the base image for the reproduction package. Required packages like `python` or `zip` are installed via the system package manager `dnf`. All required Python packages are listed in the `requirements.txt` file from where they can be installed by `pip`. This ensures that exactly the same packages and software which were originally used are used for reproduction. Zenodo[7] [43] is used to host the input data needed for the production package. Zip compression is used to reduce the size of the data set. The resulting zip archive is downloaded by the `wget` application. All data is licensed on the *Creative Commons Attribution 4.0 International licence* while all source code is licensed under the *MIT* license (this fulfills guideline 1). The script `reproduction-docker.sh` is set as the entry point of the docker image, this means that executing the Docker image will execute this script automatically (fulfilling guidelines 3 and 4). `reproduction-docker.sh` contains the commands to create plots and figures. The script takes approximately 1h 30min to run through. The Docker imagine can be downloaded or self build (see listing 5.1).

Listing 5.1: Executing the reproduction package

```
# download the repository
$ docker pull tstrempel/reproduction:1.0
# or build it yourself
$ git clone https://github.com/tstrempel/masterthesis-code
$ cd masterthesis-code
$ docker build -t tstrempel/reproduction:1.0 - < Dockerfile

# create the data directory in your home directory before
# or choose a directory to your liking
# by changing --volume <your_directory>:/data to your liking
$ docker run --volume $HOME/data:/data -ti tstrempel/reproduction:1.0
```

After the download the Docker image is executed with `docker run` (see listing 5.1). Per default, no data from the docker container can be put onto the host system. But the created plots and files need to be put on the host system. This problem is solved by using Docker volumes via `-volume` (or the `-v` shorthand). These mount a path of the

[5]https://zenodo.org/record/5552510 (accessed 06.10.2021)
[6]https://getfedora.org/ (accessed 04.10.2021)
[7]https://zenodo.org/ (accessed 04.10.2021)

host system to the docker image which enables an exchange of files through that path. In the example in listing 5.1 the `$HOME/data` directory on the host system is mounted at the `/data` directory in the Docker container. After the execution is complete the `$HOME/data` directory contains the reproduced data. It may be necessary to set the correct permissions via `sudo chown -R $USER $HOME/data` manually after the work is completed since data put out by Docker containers are only accessible with superuser privileges. The naming scheme of the created directories is `<platform>-<specifier>` where platform is either `desktop`, `laptop` or `server`. The specifier describes what was measured:

- idle: idle power draw
- O0: sorting benchmarks
- O2: sorting-O2 benchmarks
- legacy: older measurements
- series: measurements for the reproducibility chapter
- vc: video conference
- vpxenc: vpxenc measurements

5.3 Replication package

The replication Docker image consists of the modified Scaphandre, the `sorting` application and a script `replication.sh`. It aims to replicate the measurements taken in chapter 4. As was the case with the reproduction package, the Docker image can be downloaded or build from the `Dockerfile` (see listing 5.2). This provides a version of the wrapper script described in section 3.2.1 called `replication.sh` as an entry point.

Listing 5.2: The replication Docker image

```
# download the repository
$ docker pull tstrempel/replication:1.0
# or build it yourself
$ git clone https://github.com/tstrempel/replication
$ cd replication
$ docker build -t tstrempel/replication:1.0 - < Dockerfile

# create the data directory in your home directory before
# or choose a directory to your liking
# by changing --volume <your_directory>:/data to your liking
$ docker run --volume $HOME/replication:/data -ti tstrempel/replication:1.0 <
    TDP> <threads> <application>
```

Three parameters must be provided to the Docker image, the TDP and thread count of the processor and the application to execute (either `sorting` or `sorting-O2`). Depending on the used application one execution can take up to five and a half hours. Scaphandre in version v0.3.0 contains a bug where on some platforms only the process IDs are listed but not the process names. Therefore the replication script can not

contain all functionality because the program logic in the Python scripts depends on knowing the process name. The newest release v0.4.1 (published 05.10.2021) fixes this bug but not enough time was left to incorporate these changes in the forked project. Therefore some features are omitted in the `replication/evaluation.py` script to work around the bug.

If a smaller demonstration is desired, another container that just contains the modified Scaphandre is provided (see 5.3). The unmodified v0.4.1 version of Scaphandre can also be used.

Listing 5.3: The Scaphandre Docker image

```
# download the modified repository
$ docker pull tstrempel/scaphandre:1.0
# or build it yourself
$ git clone https://github.com/tstrempel/scaphandre
$ cd scaphandre
$ docker build -t tstrempel/scaphandre:1.0 - < Dockerfile

# create the data directory in your home directory before
$ docker run -v /sys/class/powercap:/sys/class/powercap -v /proc:/proc -ti
   tstrempel/scaphandre:1.0 json -t 10 -s 1

# Download and run the v0.4.1 version
$ docker pull hubblo/scaphandre
$ docker run -v /sys/class/powercap:/sys/class/powercap -v /proc:/proc -ti
   hubblo/scaphandre json -t 10 -s
```

Here the mounted volumes on `powercap` and `/proc` allows Scaphandre to read out all metrics from the host system. Since Scaphandre can only run on Linux the replication and Scaphandre docker images can also only be executed on a Linux host system.

Chapter 6

Conclusion and Future Work

The thesis answered fundamental questions about energy consumption on x86-64 processors, namely the seven questions stated in the introduction:

1. Which methods exist to measure energy consumption?
2. How granular, precise, valid and reproducible can a measurement be?
3. What is having an influence on energy consumption?
4. Does algorithm design influence it?
5. How relevant is the energy consumption of small methods?
6. Can other features like CPU load be used to approximate the energy consumption?
7. Is the time expenditure in optimizing programs for less energy use worth it?

The seven questions stated in the introduction are answered. Different methods of measuring energy consumption were shown and the most promising, RAPL, was chosen for further study. A selection of RAPL-based software was analyzed and Scaphandre was selected to be the main measurement software of this thesis. RAPL was cross-validated by comparing it with PowerTOP which uses a different measurement technique. The validity of RAPL was further confirmed by checking whether the maximal power consumption corresponds with the theoretical limit (TDP). CPU load is the main contributor (in absence of special hardware like GPUs) to the energy consumption of software, this was proven by using linear regression. Other smaller factors include the memory consumption, temperature, CPI rate and microarchitecture utilization. Scaphandre was modified to report these metrics, the incorporation of these into Scaphandre's internal model is a possible subject for future works. The current model is used to calculate the energy consumption (using only the CPU load) of all processes. Incorporating the other metrics would make this model even more precise. Regardless of slight derivations due to only using CPU load, process-level granularity is achieved. Function-level granularity can be achieved the same way. CPU time measurements per function are provided by VTune.

In chapter 4 an example application, `sorting`, was benchmarked to measure the energy consumption of different sorting algorithms with different optimization on different platforms. An important finding is that an optimized (via `-O2`) `sorting` con-

sumes a lot less energy. Therefore trying to optimize a program for less execution time or CPU time respectively, can also reduce energy consumption. Further studies using a broad range of programs are needed to ascertain that. A reproduction package was provided to recreate metrics and plots from the data put out by Scaphandre. The replication package includes the modified Scaphandre and a way to execute a measurement similar to what was done in chapter 4. With these, the findings of this thesis can be reproduced and expanded upon in the future.

List of Abbreviations

ACPI	Advanced Configuration and Power Interface
AVX	Advanced Vector Extensions
CPI	Cycles per Instruction Retired
CPU	Central Processing Unit
CV	Coefficient of Variation
DVFS	Dynamic Voltage and Frequency Scaling
GCC	GNU Compiler Collection
GPU	Graphics Processing Unit
ISA	Instruction Set Architecture
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MSR	Machine Specific Register
NVML	NVIDIA Management Library
OS	Operating System
RAPL	Running Average Power Limit
SPL	Software Product Line
TDP	Thermal Design Power
VM	Virtual Machine

List of Figures

2.1 x86 market share [11] 8
2.2 Server CPU market share [11] 8
2.3 Intel Haswell processor die [12] 9
2.4 x86-64 CPU microarchitectures (yellow colored ones are used in this work) 10
2.5 GCC compilation steps 11
2.6 Granularity levels 12
2.7 PowerTOP output of an idle laptop system 17

3.1 Visualization of the implementations 26
3.2 Graph of the unsynchronized implementation 27
3.3 Graph of the synchronized implementation 27
3.4 Data points of the unsynchronized implementation 28
3.5 Data points of the synchronized implementation 28
3.6 VTune summary 34
3.7 VTune top-down tree 35
3.8 Data points with linear regression line 36
3.9 Energy domains 37
3.10 Temperature 37
3.11 Energy consumption of RAPL domains 38
3.12 Power consumption of mprime (Prime95) 38

4.1 Energy consumption of selection and insertion sort 40
4.2 Energy consumption of the two quicksort implementations 40
4.3 CPU temperature during selection sort with an array size of 128,000 . . . 42
4.4 CPU temperature during selection sort with an array size of 256,000 . . . 42
4.5 selection and insertion sort on the server system 44
4.6 quicksort and qsort on the server system 44
4.7 selection and insertion sort on the desktop system 44
4.8 quicksort and qsort on the desktop system 44

5.1 Structure of Docker and virtual machine based systems [42] 50

List of Tables

2.1 Categorization of energy measurement methods according to [4] 7
2.2 Available hardware for testing 10
2.3 Energy unit sizes across different architectures 15

3.1 Typical temperatures for different processors [31] 26
3.2 Time complexity of sorting algorithms 29

4.1 Measurements taken for sorting (`-O0`) algorithms (NA = not available) on the laptop system (i5-6200U) 41
4.2 Variation coefficients 43
4.3 Idle power draw 45
4.4 Remaining power draw (see `masterthesis.ods`) 46
4.5 vpxenc function energy consumption 46
4.6 sorting (selection sort (with an array size of 32000) function energy consumption 47

Listings

2.1 CPU energy meter output . . . 16
2.2 PowerTOP calibration and usage . . . 17
2.3 Scaphandre JSON output (beautified) . . . 18
3.1 json.rs code modifications . . . 21
3.2 Scaphandre JSON output (beautified) . . . 24
3.3 Temperature measurement . . . 25
3.4 Temperature measurement in json.rs . . . 25
3.5 The sorting application . . . 29
3.6 vpxenc benchmark . . . 30
3.7 Executing the evaluation.py script . . . 31
3.8 Excerpt from wrapper.sh . . . 32
3.9 The time command (CPU time is user and sys combined) . . . 33
3.10 Scaphandre . . . 35
3.11 Printed out linear regression results . . . 37
5.1 Executing the reproduction package . . . 51
5.2 The replication Docker image . . . 52
5.3 The Scaphandre Docker image . . . 53

Bibliography

[1] Ralph Hintemann, Severin Beucker, and Simon Hinterholzer. *Energieeffizienz und Rechenzentren in Deutschland*. Borderstep Institut für Innovation und Nachhaltigkeit gemeinnützige GmbH, Mar. 2018, p. 12. URL: https://ne-rz.de/wp-content/uploads/2018/04/NeRZ-Kurzstudie-Stand-20180327.pdf.

[2] Ralph Hintemann. *Energiebedarf der Rechenzentren steigt trotz Corona weiter an*. Borderstep Institut für Innovation und Nachhaltigkeit gemeinnützige GmbH, Mar. 2021, p. 1.

[3] OECD. "Towards Green ICT Strategies: Assessing Policies and Programmes on ICT and the Environment". In: *Organization for Economic Co-operation and Development (OECD)* (June 2009), p. 7. ISSN: 2071-6826.

[4] Luca Ardito et al. "Methodological Guidelines for Measuring Energy Consumption of Software Applications". In: *Scientific Programming* 2019 (2019). ISSN: 10589244. DOI: 10.1155/2019/5284645.

[5] Felix Rieger and Christoph Bockisch. "Survey of approaches for assessing software energy consumption". In: *CoCoS 2017 - Proceedings of the 2nd ACM SIGPLAN International Workshop on Comprehension of Complex Systems, co-located with SPLASH 2017* (2017). DOI: 10.1145/3141842.3141846.

[6] Norbert Siegmund, Marko Rosenmüller, and Sven Apel. "Automating energy optimization with features". In: *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development, FOSD'10* (2010). DOI: 10.1145/1868688.1868690.

[7] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. "A validation of DRAM RAPL power measurements". In: *ACM International Conference Proceeding Series* 03-06-October-2016 (2016). DOI: 10.1145/2989081.2989088.

[8] Shuai Hao et al. "Estimating mobile application energy consumption using program analysis". In: *Proceedings - International Conference on Software Engineering* (2013). ISSN: 02705257. DOI: 10.1109/ICSE.2013.6606555.

[9] Chiyoung Seo, Sam Malek, and Nenad Medvidovic. "Component-level energy consumption estimation for distributed java-based software systems". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5282 LNCS (2008). ISSN: 16113349. DOI: 10.1007/978-3-540-87891-9_7.

[10] *NVIDIA Tesla V100 data sheet*. Accessed 05.09.2021. Sept. 2021. URL: https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf.

[11] Joel Hruska. *AMD x86 CPU Market Share Soars, Hits 14-Year High*. Accessed 30.09.2021. Aug. 2021. URL: https://www.extremetech.com/computing/325848-amd-x86-cpu-market-share-soars-hits-14-year-high.

[12] Volker Rißka. *Core i7-5820K und 5960X im Test: Intel Haswell-E mit sechs und acht Kernen*. Accessed 05.09.2021. Aug. 2014. URL: https://www.computerbase.de/2014-08/intel-core-i7-5820k-5960x-haswell-e-test/.

[13] *Intel Hyperthreading*. Accessed 05.09.2021. May 2021. URL: https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading.html.

[14] *AMD Family 10h Server and Workstation Processor Power and Thermal Data Sheet*. June 2010.

[15] Aleksey Dolya. *Interview with Brian Kernighan*. Accessed 05.05.2021. Aug. 2003. URL: https://www.linuxjournal.com/article/7035.

[16] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language - 2nd Edition*. Prentice Hall, 1988.

[17] *gcc(1) — Linux manual page*. Accessed 05.10.2021. Oct. 2021. URL: https://man7.org/linux/man-pages/man1/gcc.1.html.

[18] Lucio Mauro Duarte et al. “A Model-based Framework for the Analysis of Software Energy Consumption”. In: *ACM International Conference Proceeding Series* (2019). DOI: 10.1145/3350768.3353813.

[19] Marco Couto, João Paulo Fernandes, and João Saraiva. “Statically analyzing the energy efficiency of software product lines”. In: *Journal of Low Power Electronics and Applications* 11 (1 2021). ISSN: 20799268. DOI: 10.3390/jlpea11010013.

[20] Daniel Molka et al. “Characterizing the energy consumption of data transfers and arithmetic operations on x86-64 processors”. In: *2010 International Conference on Green Computing, Green Comp 2010* (2010). DOI: 10.1109/GREENCOMP.2010.5598316.

[21] *Energy Debugging Tools for Embedded Applications*. Silicon Labs. URL: https://www.silabs.com/documents/public/white-papers/energy-debugging-tools.pdf.

[22] Dirk Beyer and Philipp Wendler. “CPU Energy Meter: A Tool for Energy-Aware Algorithms Engineering”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 12079 LNCS (2020). ISSN: 16113349. DOI: 10.1007/978-3-030-45237-7_8.

[23] Benoit Petit. *scaphandre*. Accessed 06.09.2021. 2021. URL: https://github.com/hubblo-org/scaphandre.

[24] *Powertop*. Accessed 06.09.2021. 2021. URL: https://github.com/fenrus75/powertop.

[25] Srinivas Pandruvada. *RUNNING AVERAGE POWER LIMIT – RAPL*. Accessed 06.09.2021. June 2014. URL: https://01.org/blogs/2014/running-average-power-limit-%E2%80%93-rapl.

[26] Kashif Nizam Khan et al. "RAPL in action: Experiences in using RAPL for power measurements". In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* 3 (2 2018). ISSN: 23763647. DOI: 10.1145/3177754.

[27] Ali Hassan Sodhro et al. "Energy-efficiency of tools and applications on internet". In: *Computational Intelligence for Multimedia Big Data on the Cloud with Engineering Applications* (2018). DOI: 10.1016/B978-0-12-813314-9.00014-1.

[28] Timo Johann et al. "How to measure energy-efficiency of software: Metrics and measurement results". In: *2012 1st International Workshop on Green and Sustainable Software, GREENS 2012 - Proceedings* (2012). DOI: 10.1109/GREENS.2012.6224256.

[29] Zhang Huazhe and Hoffman H. "A quantitative evaluation of the RAPL power control system". In: *Feedback Computing* (2015).

[30] Abdelhafid Mazouz, Benoît Pradelle, and William Jalby. "Statistical validation methodology of CPU power probes". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8805 (Part 1 2014). ISSN: 16113349. DOI: 10.1007/978-3-319-14325-5_42.

[31] M. Mateen. *Ideal CPU Temp Ranges: Max Safe, Normal CPU Temperature (Gaming vs Idle)*. Accessed 10.09.2021. Sept. 2021. URL: https://www.cputemper.com/maximum-and-normal-cpu-temperature/.

[32] C. A. R. Hoare. "Quicksort". In: *The Computer Journal* 5 (1 1962). ISSN: 0010-4620. DOI: 10.1093/comjnl/5.1.10.

[33] *Gentoo Wiki - GCC optimization*. Accessed 08.10.2021. Aug. 2021. URL: https://wiki.gentoo.org/wiki/GCC_optimization.

[34] *Python*. Accessed 06.09.2021. June 2021. URL: https://github.com/python/cpython.

[35] *time(7) — Linux manual page*. Accessed 05.10.2021. Oct. 2021. URL: https://man7.org/linux/man-pages/man7/time.7.html.

[36] *Intel VTune Profiler*. Accessed 08.10.2021. Oct. 2021. URL: https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html.

[37] *Intel VTune Profiler User Guide - CPI Rate*. Accessed 08.10.2021. Oct. 2021. URL: https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/reference/cpu-metrics-reference/cpi-rate.html.

[38] *Intel VTune Profiler User Guide - Microarchitecture Usage*. Accessed 08.10.2021. Oct. 2021. URL: https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/reference/cpu-metrics-reference/microarchitecture-usage.html.

[39] Gernot Klingler. *gprof, Valgrind and gperftools - an evaluation of some tools for application level CPU profiling on Linux*. Accessed 05.10.2021. Jan. 2015. URL: https://gernotklingler.com/blog/gprof-valgrind-gperftools-evaluation-tools-application-level-cpu-profiling-linux/.

[40] *Minimize Collection Overhead*. Accessed 05.10.2021. URL: https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/analyze-performance/control-data-collection/minimizing-collection-overhead.html.

[41] M. S. Krafczyk et al. "Learning from reproducing computational results: Introducing three principles and the Reproduction Package". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 379 (2197 2021). ISSN: 1364503X. DOI: 10.1098/rsta.2020.0069.

[42] *Docker web page*. Accessed 08.10.2021. Oct. 2021. URL: https://www.docker.com/resources/what-container.

[43] Strempel. Tom. "Dataset for master's thesis reproduction package". In: (Oct. 2021). DOI: 10.5281/ZENODO.5559595. URL: https://zenodo.org/record/5559595.

Ich versichere, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann. Ich versichere, dass das elektronische Exemplar mit den gedruckten Exemplaren übereinstimmt.

Leipzig, 13.10.2021