

PROJECT REPORT

DEPARTMENT OF COMPUTING

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

---

**PowerKap - A tool for Improving Energy  
Transparency for Software Developers on  
GNU/Linux (x86) platforms**

---

*Author:*  
Krish De Souza

*Supervisor:*  
Dr. Anandha Gopalan

Submitted in partial fulfilment of the requirements for the M.Eng Computing 4 of  
Imperial College London

# Contents

|          |  |          |
|----------|--|----------|
| <b>1</b> | <b>Introduction</b>  | <b>6</b> |
| 1.1      | Motivation . . . . .   | 6        |
| 1.2      | Objectives . . . . .   | 7        |
| 1.3      | Achievements . . . . .   | 7        |
| <b>2</b> | <b>Background</b>  | <b>9</b> |
| 2.1      | The relationship between power and energy. . . . .             | 9        |
| 2.2      | Power controls on x86 platforms . . . . .                      | 9        |
| 2.3      | Improving software for power efficiency . . . . .              | 10       |
| 2.3.1    | Algorithm . . . . .  | 10       |
| 2.3.2    | Multithreading . . . . .                                       | 10       |
| 2.3.3    | Vectorisation . . . . .  | 10       |
| 2.3.4    | Improper sleep loops . . . . .                                 | 12       |
| 2.3.5    | OS Timers . . . . .  | 13       |
| 2.3.6    | Context aware programming . . . . .                            | 13       |
| 2.4      | Current methods of monitoring energy. . . . .                  | 14       |
| 2.4.1    | Out of Band Energy Monitor . . . . .                           | 14       |
| 2.4.2    | In-Band Energy Monitor . . . . .                               | 14       |
| 2.4.2.1  | Powertop . . . . .   | 15       |
| 2.4.2.2  | Turbostat . . . . .  | 16       |
| 2.5      | Related Work . . . . .   | 16       |
| 2.5.1    | ENTRA 2012-2015 . . . . .                                      | 16       |
| 2.5.1.1  | Common Assertion Language . . . . .                            | 16       |
| 2.5.1.2  | Compiler Optimisation and Power Trade-offs . . . . .           | 18       |
| 2.5.1.3  | Superoptimization . . . . .                                    | 18       |
| 2.5.1.4  | Thermal trade-off . . . . .                                    | 20       |
| 2.5.2    | eProf . . . . .  | 20       |
| 2.5.2.1  | Asynchronous vs Synchronous . . . . .                          | 20       |
| 2.5.2.2  | Profiling implementation . . . . .                             | 21       |
| 2.5.3    | Energy Formal Definitions . . . . .                            | 21       |
| 2.5.3.1  | Java Based Energy Formalism . . . . .                          | 22       |
| 2.5.3.2  | Energy Application Model . . . . .                             | 22       |
| 2.5.4    | Impact of language, Compiler, Optimisations . . . . .          | 22       |
| 2.5.4.1  | Choice of Language . . . . .                                   | 22       |
| 2.5.4.2  | Relation of execution time and energy consumption . . . . .    | 23       |
| 2.5.4.3  | Impact of Optimisation flags . . . . .                         | 23       |
| 2.5.4.4  | Choice in Algorithm . . . . .                                  | 25       |
| 2.6      | Similar Tools . . . . .  | 25       |
| 2.6.1    | AEON . . . . .   | 25       |
| 2.6.2    | Visual Studio . . . . .  | 26       |
| 2.7      | Gathering Energy Measurements . . . . .                        | 27       |
| 2.7.1    | Module Specific Registers (MSR) . . . . .                      | 27       |
| 2.7.1.1  | Running Average Power Limits (RAPL) Interfaces . . . . .       | 27       |
| 2.7.2    | Perf . . . . .   | 30       |
| 2.7.3    | PAPI (Performance Application Programming Interface) . . . . . | 30       |

|          |   |           |
|----------|---|-----------|
| 2.7.4    | HWMON . . . . .   | 30        |
| 2.7.5    | Intel Powercap . . . . .  | 30        |
| 2.7.6    | PowerAPI . . . . .  | 31        |
| 2.7.7    | Summary . . . . .   | 31        |
| 2.8      | Interacting with these interfaces from user space . . . . .     | 33        |
| 2.8.1    | Sysfs . . . . .   | 33        |
| 2.8.2    | NetLink . . . . .   | 33        |
| 2.8.3    | Procfs . . . . .  | 33        |
| <b>3</b> | <b>Profiling a program</b>                                      | <b>35</b> |
| 3.1      | Initial Ideas . . . . .   | 35        |
| 3.2      | The Profiler . . . . .  | 35        |
| 3.2.1    | Design Ideas . . . . .  | 35        |
| 3.2.2    | Chosen Design . . . . .   | 36        |
| 3.2.3    | Why not use current profilers? . . . . .                        | 38        |
| 3.2.4    | Choice of energy interface . . . . .                            | 38        |
| 3.2.4.1  | Energy Consumption of CPU and Memory . . . . .                  | 38        |
| 3.2.5    | Thermal Information . . . . .                                   | 39        |
| 3.2.5.1  | Battery Information . . . . .                                   | 39        |
| 3.2.5.2  | DiskIO . . . . .  | 39        |
| 3.2.5.3  | Network . . . . .   | 40        |
| 3.2.5.4  | Choice of language . . . . .                                    | 41        |
| 3.2.6    | Profiler Design . . . . .                                       | 41        |
| 3.2.6.1  | Forker . . . . .  | 41        |
| 3.2.6.2  | Profiler . . . . .  | 43        |
| 3.2.6.3  | Sysfs, Procfs and Energy Interfaces . . . . .                   | 43        |
| 3.2.6.4  | Printer . . . . .   | 45        |
| 3.2.6.5  | Measurement and Energy Structure . . . . .                      | 45        |
| 3.2.6.6  | Networking Script . . . . .                                     | 49        |
| 3.2.7    | Implementation Details . . . . .                                | 49        |
| 3.2.7.1  | Steps taken to minimise the overhead introduced by the profiler | 49        |
| 3.2.7.2  | Avoiding the impact of the user environment . . . . .           | 51        |
| 3.3      | Linux Java Energy Assessment (LJEA) plugin . . . . .            | 51        |
| 3.3.1    | Choice of IDE . . . . .   | 52        |
| 3.3.2    | EnergyPoints . . . . .  | 52        |
| 3.3.2.1  | The profiling code . . . . .                                    | 52        |
| 3.3.2.2  | StackTrace . . . . .  | 54        |
| 3.3.2.3  | Energy Graphs . . . . .   | 54        |
| 3.3.3    | Implementation Details . . . . .                                | 54        |
| 3.3.3.1  | The UI design . . . . .   | 54        |
| 3.3.3.2  | Action Classes . . . . .  | 57        |
| <b>4</b> | <b>Project Evaluation</b>                                       | <b>59</b> |
| 4.1      | The hardware and methodology . . . . .                          | 59        |
| 4.2      | The Profiler . . . . .  | 62        |
| 4.2.1    | The Results . . . . .   | 62        |
| 4.3      | The Battery Measurements . . . . .                              | 62        |
| 4.4      | CPU Measurements . . . . .                                      | 66        |

|          |   |            |
|----------|---|------------|
| 4.4.1    | Battery vs CPU measurements . . . . .             | 66         |
| 4.4.2    | CPU Stress Test . . . . .                         | 66         |
| 4.4.3    | Desktop vs Laptop . . . . .                       | 69         |
| 4.4.3.1  | Gaming Desktop . . . . .                          | 69         |
| 4.4.3.2  | Modern Laptop . . . . .                           | 72         |
| 4.4.4    | Governor Choice . . . . .                         | 74         |
| 4.4.5    | BigBuckBunny Mplayer Test . . . . .               | 76         |
| 4.4.6    | Choice of Algorithm . . . . .                     | 79         |
| 4.4.7    | Asynchronous vs Busywait . . . . .                | 81         |
| 4.4.8    | Effects of Timers . . . . .                       | 83         |
| 4.4.9    | Reproducibility of the results gathered . . . . . | 83         |
| 4.4.9.1  | John the Ripper . . . . .                         | 85         |
| 4.4.9.2  | OpenSSL . . . . .                                 | 86         |
| 4.4.9.3  | STREAM Benchmark . . . . .                        | 86         |
| 4.4.9.4  | Sunflow benchmark . . . . .                       | 87         |
| 4.4.9.5  | MPlayer . . . . .                                 | 87         |
| 4.4.9.6  | Summary of Benchmark Findings . . . . .           | 89         |
| 4.4.10   | Temperature Sensor Data . . . . .                 | 89         |
| 4.4.11   | IO Capturing Capability . . . . .                 | 91         |
| 4.4.11.1 | Ping Test . . . . .                               | 91         |
| 4.4.12   | DiskIO Capturing Technique . . . . .              | 94         |
| 4.4.13   | Case Study: Browser Comparison . . . . .          | 96         |
| 4.4.14   | LJEA . . . . .                                    | 100        |
| 4.4.14.1 | Graphing Module . . . . .                         | 100        |
| 4.4.14.2 | Energy Trace . . . . .                            | 100        |
| <b>5</b> | <b>Conclusion</b>                                 | <b>103</b> |
| <b>6</b> | <b>Recommendations for Future Work</b>            | <b>104</b> |
| 6.1      | PowerKap . . . . .                                | 104        |
| 6.1.1    | Expanding the interfaces . . . . .                | 104        |
| 6.1.2    | Sysfs/Procfs/Linux Interfaces . . . . .           | 104        |
| 6.1.3    | Asynchronous computing . . . . .                  | 104        |
| 6.1.4    | Machine Learning and Model Generating . . . . .   | 105        |
| 6.1.5    | Handling Thermal Spikes . . . . .                 | 105        |
| 6.2      | LJEA . . . . .                                    | 105        |
| 6.2.1    | Introducing code suggestions . . . . .            | 105        |
| 6.2.2    | Expanding to other IDEs and Languages . . . . .   | 106        |
| <b>7</b> | <b>Appendix</b>                                   | <b>112</b> |
| 7.1      | User Guide . . . . .                              | 112        |
| 7.1.1    | Setup . . . . .                                   | 112        |
| 7.1.2    | How to use PowerKap . . . . .                     | 112        |
| 7.1.3    | How to use LJEA . . . . .                         | 113        |



## **Abstract**

This report covers an exploration of a technique for evaluating the energy consumption of programs on GNU/Linux (x86) based platforms. The project in particular, focuses on capturing energy information based on physical counters present in user space. In addition, the project explores some of the interfaces, and evaluates their reproducibility across different platforms.

This project introduces a series of tools that can be used to evaluate the energy consumption of a program. These tools integrate directly into developer environments to improve energy behaviour for programs. Various techniques are evaluated in this project across a series of benchmarks over different hardware, including a case study comparing Mozilla Firefox 53 and Google Chrome 59. From our findings, it was determined that Firefox can be more efficient than Chrome when watching a YouTube video.

Our approach makes use of a novel technique for capturing network traffic information by utilising user namespaces. This is a new feature introduced in modern kernels (Linux Kernel 3.8 and above). This is useful as the bytes sent and received scale linearly with energy use.

### **Acknowledgements**

I would like to thank Dr. Anandha Gopalan who helped supervise this project even on his time off. I would also like to express my gratitude to my family for their moral support and warm encouragements.

# 1 Introduction

## 1.1 Motivation

Measuring energy usage is a topic of significant importance in our energy-aware society. In 2012, the energy consumption of the ICT industry corresponded to 4.7% of the world's energy consumption [1]. This corresponds to approximately 530Mt of  $CO_2$  released into the atmosphere [1]. Climate change targets necessitate reduction of energy consumption in all aspects including the IT industry. In particular, data centers have been viewed as particularly inhibitive towards climate goals [2]. A large data center can consume around 30GWh (Gigawatt hours) of power per year according to IT trade association TechUK. A mismanaged poorly operated datahall (data center) can consume approximately 60% more energy relative to a well managed one. This can be increasingly problematic as more data centers are built to cope with increased Internet demand. In Figure 1, we can see some predictions for energy usage over the next decades. Realistically, the worst case scenario is unlikely to occur for economic reasons. However, it clearly emphasises the need for energy efficiency.

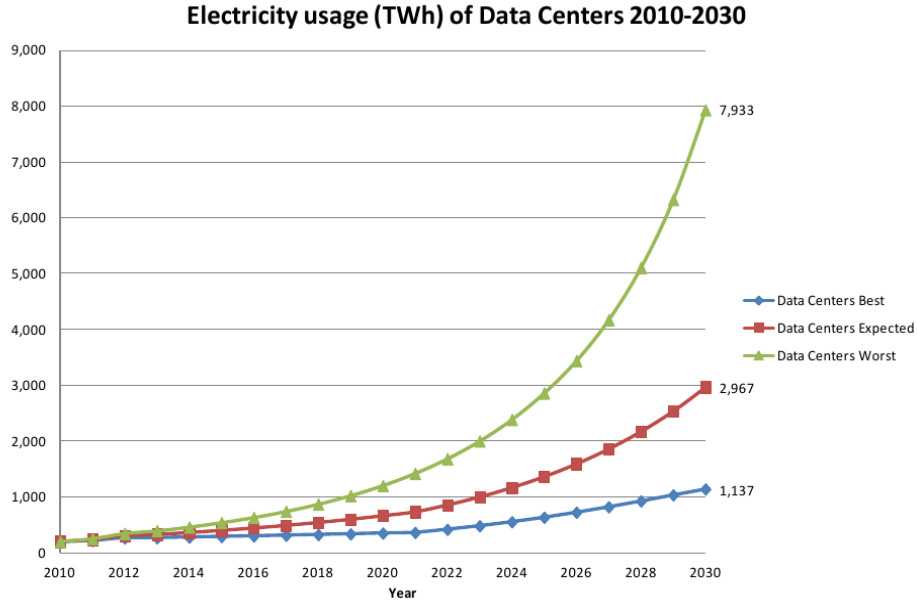


Figure 1: Global Energy demand from Data Centers 2010-2030 [3]

Traditionally, industry have achieved better power efficiency through gains in hardware improvements. In contrast, the software side of energy consumption is often overlooked. According to Gadi Singer, vice-president of the IAG (Intel Architecture Group) and general manager of the SoC (System on a Chip) enabling group at Intel, by better optimising software to be in control of power states, it is possible to achieve up to 3-5x better power consumption [4].

Such a task is quite laborious with current technologies. This is because current methods of profiling and optimising software energy rely upon deep inspection throughout the

software stack. In particular, energy information present in hardware does not translate directly to developer tools such as the compiler or Integrated Developer Environment (IDE). This can make it difficult for developers to know the effects of their design choices with respect to energy consumption. Even the effects of different hardware platforms for the same code can produce drastically different energy profiles, which can be unclear to a developer.

In terms of current methodologies, there are few commercial tools for estimating power consumption of software on GNU/Linux. At present, there are various power estimation and calibration tools provided by Intel for x86. This includes two open source programs, Powertop [5] and Turbostat [6] which enables the ability to measure the overall power consumption. These tools, provide a system overview for energy consumption without particular granularity in information related to the codebase. For example, it is not possible to use these tools to find the energy cost of a specific function within a program. In addition, there are a lot of interfaces present on GNU/Linux that enable power and energy profiling. However, these require some form of calibration and none are integrated into common developer tools such as IDEs. There is a need to rectify this situation by creating a user facing program to estimate how much energy is being consumed. In this way, developers can capture energy information during the execution of a specific function or test. This can be useful for capturing unexpected energy usage. The idea is that developers could design specific tests that enable the estimation of the energy consumed by programs such as Google Chrome. This is to ensure optimum energy behaviour. An example could be watching a YouTube video in the background. In such a situation where the video cannot be seen, is it particularly necessary for the GPU to be running and consuming energy?

## 1.2 Objectives

The goal of this project is to improve energy transparency during development. This will be achieved by completing the following objectives.

1. Create a more granular mechanism for developers to profile specific portions of code using current hardware energy metrics.
2. Design a plugin or extension that enables developers to easily profile the code within their current development environments.
3. Display the results of profiling the code back to the user in an informative form.
4. Evaluate the tool's reproducibility and accuracy against other commercial tools.
5. Assign system information and energy metrics to specific portions of code. Based on this, create guarantees for the energy cost for a given behaviour on a specific hardware platform.

## 1.3 Achievements

The following contributions have been made in this project.

1. Created a set of tools to measure the CPU and memory energy consumption of a generic program all within user space. This is without any further modifications to the user's system or additional permanent configuration.

2. Created an user intuitive interface that enables users to interpret results directly within the IDE.
3. Enabled an ability for users to highlight specific code aspects within their code just by annotating the code.
4. Created an unique method of capturing per process network bandwidth. This enables estimation of networks without relying on expensive user space programs.
5. Used PowerKap to research to evaluate the energy efficiency of Mozilla Firefox 53 and Google Chrome 59.
6. Used PowerKap to explore the energy consequences of various benchmarks on various platforms and governors. In particular, researching the key differences in energy handling for laptops and desktop processors.
7. Evaluated the reproducibility of the measurement data. In particular, demonstrating that the energy behaviour of programs is not necessarily the same across different hardware. Demonstrated this by benchmarking across identical hardware.

## 2 Background

This section is designed to explore some of the relationships between software and power.

### 2.1 The relationship between power and energy.

Power is defined as the rate of doing work, measured in Watts [7]. On computers, this value is recorded over a small finite value of time. For this reason, it is still defined as the average power consumed over a period of time. This is in contrast to instantaneous power which is the rate of energy consumption at a particular moment [8].

Within this report, power and average power are used interchangeably. The goal of PowerKap is to reduce the overall energy consumption.

### 2.2 Power controls on x86 platforms

When considering software efficiency on x86 platforms the first concept to note is that of hardware states. These states were introduced through an open standard called ACPI (Advanced configuration and power interface) [9]. This standard was created in 1999 through a consortium consisting of Hewlett-Packard, Intel, Microsoft, Phoenix and Toshiba.

Within the standard, particular hardware behaviour is defined for specified states. In general, the higher the state for a given device, the lower its energy consumption. This is because under higher states, the device can attain lower energy consumption by disabling or turning off sub-components as required. For example, a state of C0 indicates that a CPU core is operating normally. In a state of C3, the core is allowed access to caches, but main memory access for the core is disabled. The core also stops generating clocks and disables “snooping activity”. The expected behaviour of states C0-C3 are described further in section 8.1 of the specification, with ACPI v2.0 adding capabilities for further C states. In addition, C states apply both per core and for the processor as a whole. When the entire processor achieves a high C state, the power consumption can be reduced significantly [8]. The ACPI standard also provides other state requirements such as specific device states D0-D3 or further control of processor states P0-Pn. The D states perform similarly to C states whilst P states differ as they control processor frequency and voltages [10]. For external components that are connected through the PCI Express interface, a different standard is used for power management called ASPM (Active State Power Management). This interface grants additional power states (Link states) which can improve power efficiency in exchange for greater latency [11].

Operating Systems are able to adjust scheduling and various power states to optimise for different environments via these various states. This is achieved by implementing an ACPI compliant module called an OSPM (Operating System-directed configuration and Power Management) [9]. This allows the Operating System to control the CPU sleep states for a given device. For example, when a laptop is plugged in, the OSPM may optimise the hardware behaviour for performance. In general, the longer the CPU spends in higher C states and P states, the more energy is saved. However, frequent transitions from high and low C states can also be expensive. This is because these transitions can introduce latency. As such, the machine must stay in active states longer resulting in energy penalties.

## 2.3 Improving software for power efficiency

This section corresponds to a series of suggested improvements to software that are based on green technical manuals by Intel [12, 10]. These improvements are important as they relate to specific behaviours that this project is trying to distinguish. The goal of many of these techniques is to reduce and consolidate the number of high power C state activities.

### 2.3.1 Algorithm

Algorithm and data structure choice can have massive impact on power profile of a program. When deciding software design, it is important to tailor the program to handle average use cases. This is generally achieved through manual inspection and understanding the context in which a program is being used. An algorithmic choice can optimise a problem to use less time or less space. In turn, such benefits would result in greater energy efficiency by reducing the time a CPU is active allowing it to return to idle faster. Similarly, optimising an algorithm for less space would have the advantage of reducing costly cycles spent shifting data from main memory or disk storage.

### 2.3.2 Multithreading

**Definition 2.1.** Threads are a basic unit of CPU utilisation. They consist, of a program counter, stack and a set of registers [13].

Introducing parallelism can be another approach to maximise the power efficiency of a program. To do so, one could split a program into multiple smaller sets of computation called threads. These threads can then be run concurrently across multiple cores of CPU. By doing this, the program can achieve greater efficiency as more actions can be completed during the same cycle. This means that the task can be completed faster and the CPU can return to idle power efficient states quicker. This efficiency can be seen in Figure 2 which compares the power efficiency gains when completing a task using a multithreaded versus single threaded approach. According to the original paper [10], the 8 threaded example in Figure 2 consumed 25% less power relative to the single threaded benchmark.

### 2.3.3 Vectorisation

If multithreading is not possible, another approach to introducing parallelism within individual programs is to take advantage of the instruction set of a given platform. On x86 platforms, Intel and AMD provide advanced instructions such as SIMD (Single instruction multiple data). By doing this, computations can be completed much quicker. This enables the CPU to return to idle faster therefore saving energy. Such speed optimisation can be explained by Figure 3. The availability and choice of vectorization instructions, depends upon the type of compiler. For example, to enable most vectorization instructions on GCC simply compile the program with -O3 flag [15].

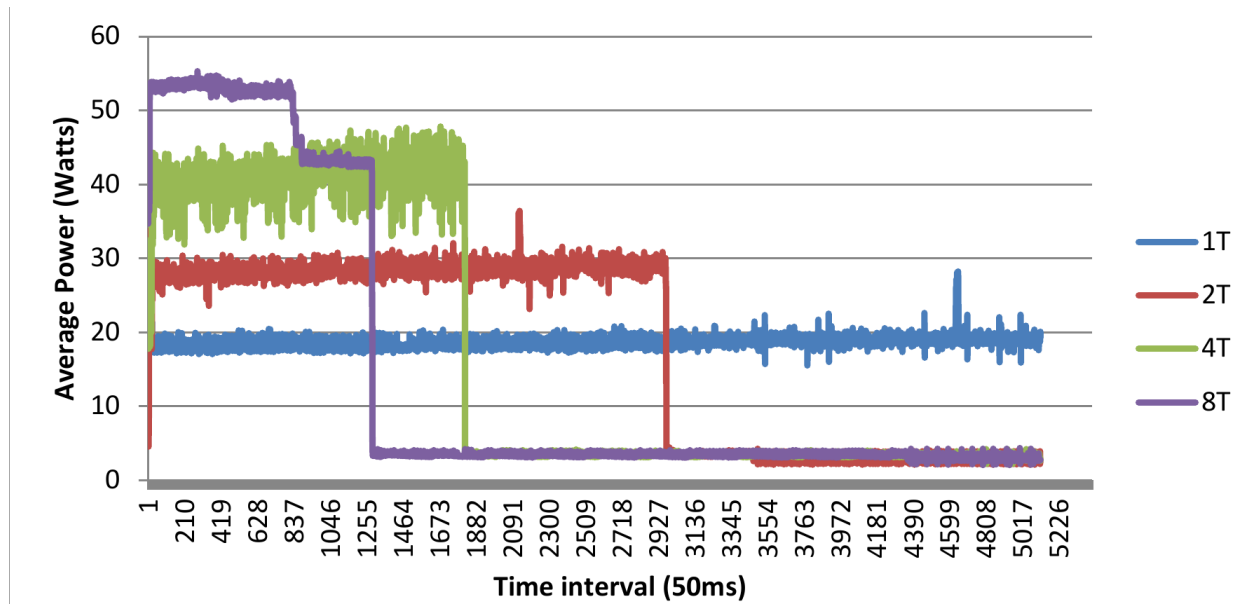


Figure 2: The above benchmark was run by Intel [10]. The test ran Maxon's Cinebench 11.5 [14] on an Intel i7 processor.

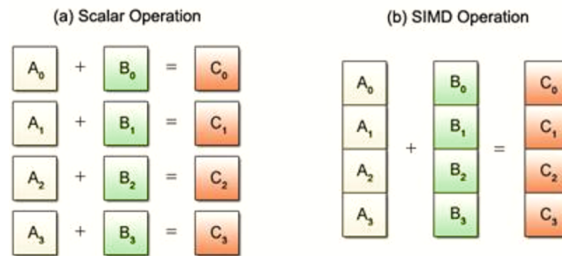


Figure 3: The above diagram is an example of a SIMD (single instruction multiple data) instruction. In the left, four add operations are necessary to complete the task. The case on the right can instead be completed using just one operation using the correct SIMD instruction. In this way, parallelism is introduced at an instruction level.

The diagram above was sourced from kernel.org [16]



```

while (!acquired_lock)
{
    sleep(0);
}
do_work();
release_lock();

```

**Figure 4: An example of an energy inefficient tight loop from Intel [12].**

```

if (!acquired_lock)
{
    for(int i = 0; i < max_spin_count; ++i)
    {
        _mm_pause();
        if (read_volatile_lock())
        {
            if (acquire_lock())
            {
                goto PROTECTED_CODE;
            }
        }
    }
    Sleep(0);
    goto ATTEMPT_AGAIN;
}
PROTECTED_CODE:
do_work();
release_lock();

```

**Figure 5: An example of a better optimised tight loop from Intel [12].**

#### 2.3.4 Improper sleep loops

Figure 4 illustrates an example tight loop. These loops are typically used when waiting for a device or signal. This loop can cause a processor to run in a highly active state. This is because the program causes the Operating System to perform multiple expensive operations called context switches. In each case, when the program calls sleep, the active thread is switched out. This requires the saving of the current program state which can be expensive. With a time set to 0, the thread is immediately switched back resulting in another context switch.

Instead, it is recommended by Intel to reduce the times in which the code is busy-waiting by adding pause instructions. An illustration of optimised code can be seen in Figure 5. According to Intel, the code from Figure 5 allows waiting tasks to acquire the CPU more easily. It also reduces the number of context switches by reducing the calls to sleep(0). In

addition, the CPU can reach higher C states by reducing running power components within the pipeline. According to Intel, such practices can result in significant savings of up to 21% power saving at the CPU level [12].

Another alternate approach to avoid such loops is to adopt an event driven paradigm. This paradigm works by only performing the `do_work()` function when notified by an external program. This saves power as a program simply reacts upon notification. Such mechanisms depend on the choice of Operating System and language.

### 2.3.5 OS Timers

For programs that rely on repeated tasks occurring at scheduled intervals, it is normally a convention to specify some time slice in which the task is repeated. Such tasks can be made more power efficient by ensuring such activities repeat at a similar time to that of other activities. Such a mechanism is called timer coalescing.

This mechanism provides power benefits through the fact that in between timer intervals, the CPU can idle. If a program is not synchronised with the system timer or has an unnecessarily low timer resolution (say 1ms), this can cause unnecessary power draw. This is because the CPU performs a “wakeup” where it shifts from idle states to active states to perform the task. On Windows, the normal System timer occurs at 15.6 ms. For this reason, it is recommended for repeated timer tasks at an interval larger than this value [10]. According to Microsoft’s documentation, by utilising high resolution timers smaller than this value, a given program could cause a reduction of battery life by up to 25% [17].

**Definition 2.2.** A Jiffy is a unit of time, that represents the number of timer interrupts since the system has booted. On each timer interrupt by the CPU, the number of Jiffies are incremented by one. The relationship between Jiffy and seconds is defined by the constant HZ within the kernel. This value defines the number of Jiffies each second [18].

On GNU/Linux platforms, the default rate for a given program is a complex topic with massive variability depending on choice of kernel and hardware. For example, with kernel versions before 2.4, the tick rate between Jiffies would be 10ms. This was changed from 2.4 to a value such that each tick occurred at 1ms [19]. From kernel version 3.10, the kernel has introduced more dynamic capabilities. In such cases, when idle the tick rate can be slowed such that it may occur once a second. This occurs if the flag “CONFIG\_NO\_HZ\_FULL” is enabled within the kernel configuration. From a developer perspective, such variances can be difficult to code for power efficiency. For this reason, it is recommended to utilise a kernel mechanism called “timer\_slack” [20]. This mechanism allows the developer to specify how long each call be delayed for. In this way, the kernel can coalesce the activities of multiple timers so that multiple actions can occur whilst the processor is already active.

### 2.3.6 Context aware programming

In relation to the earlier OSPM mechanism stated earlier, most operating systems have the capability of detecting whether certain hardware and devices are enabled or not. This information can be utilised by software developers to improve the energy efficiency of their programs. This is because if the feature or hardware device is not required, it should be possible for users to disable certain features of the code. For example, if the device has

disabled a web-cam, it should not be necessary to initialise or use any of the data structures relevant for the camera within the program.

In addition to the above, the OSPM mechanism on most Operating systems provide the capability of allowing developers to be aware of the current power schemes adopted by the Operating System. For example, Windows programs are capable of knowing whether the machine is in a “High performance mode” or other “Battery saving mode”. This concept could be leveraged to better improve battery performance. For example, for a game, when running on battery, it would be worthwhile to cap the framerate of the program to the refresh rate of the monitor. Additional frames beyond this would provide minimal gain as it’s unlikely to be noticeable to the user.

## **2.4 Current methods of monitoring energy.**

In order to physically monitor the energy usage of a given program or function, there are two categories of energy monitoring methodologies, each with advantages and disadvantages depending on the context of the purpose [21].

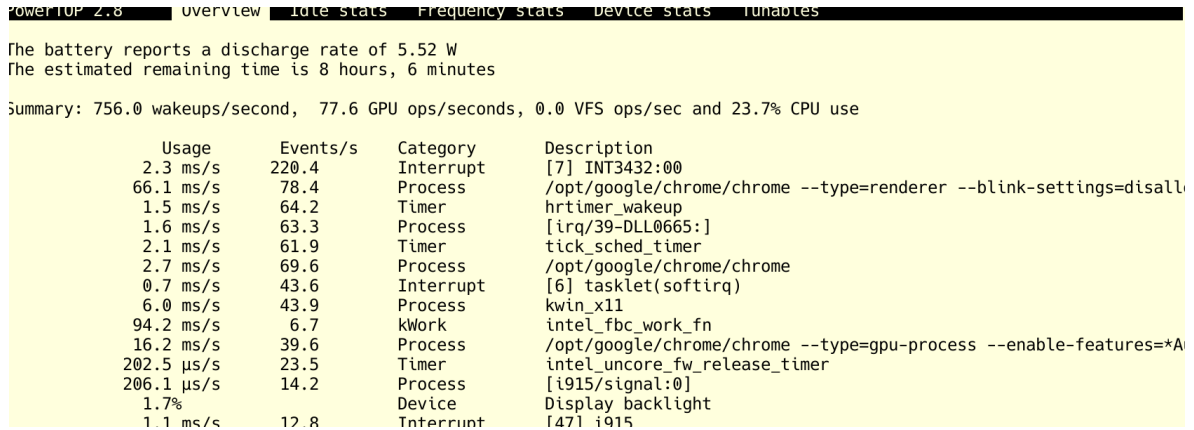
### **2.4.1 Out of Band Energy Monitor**

These types of energy monitors, are used to measure the energy by using methodologies external to the system. Examples of such a system include the use of physical readings such as ammeters, voltmeter and stop watch. These kinds of metrics have certain advantages when evaluating a system as a whole. This is because sampling external readings would not affect the performance of the system. For a developer however, such metrics have limited utility for diagnosing the energy usage of individual programs. This is because they are not granular enough to distinguish the energy usage of specific components within a system. In some cases, such external metrics can be turned to in-band energy monitoring techniques. For example, with a laptop running GNU/Linux on battery, it is possible to measure the energy usage by reading current drawings written to the file `“/sys/class/power_supply/BAT0/current_now”`.

### **2.4.2 In-Band Energy Monitor**

Another way to measure the energy use involves integrating power monitoring within the system. This method is described as an “In-band” power monitor. This works by measuring specific performance counters such as CPU state transitions, process wake ups or reading specific energy registers. On newer Intel platforms such as SandyBridge, these metrics have been improved due to the introduction of Running Average Power Limit (RAPL) metrics [22]. The energy usage provided by RAPL is localised to specific power domains such as the cores or DRAM. Recent x86 processors from AMD have similar capabilities that are available using `fam15h_power` drivers [23]. They provide similar energy consumption statistics for the processor as a whole. The main disadvantage of this approach is that by sampling such metrics within the system, the power measurements themselves affect the energy usage, which reduces the accuracy of the measurements.

On GNU/Linux there are a variety of free and open tools that are used to currently measure the energy usage of a program and a system.



**Figure 6:** This picture demonstrates some of the per program statistics offered by Powertop [5].

#### 2.4.2.1 Powertop

This free and open software was designed by Intel to optimise power behaviour on Intel Linux based platforms. The program takes advantage of current metrics provided by the OSPM and hardware counters to provide a general overview of energy consumption. This is achieved by taking advantage of the RAPL registers and ACPI statistics to measure aspects such as wakeups, C state transitions and specific device features. The program is also able to read and provide specific kernel and device configurations to optimise device power usage on GNU/Linux platforms.

Figure 6, shows a snapshot of Powertop in use. The Figure shows a breakdown of the number of wakeup operations as well as CPU usage per program. The tabs at the top lead to further details such as C state transitions per core and frequency information. In addition, Powertop reads the data from the battery discharge file stated previously and is able to estimate the power consumption in Watts. Such a feature is only available when the battery is discharging.

Similarly, in Figure 7, the additional capabilities offered by the program can be observed. In this case, Powertop offers the ability to set and tune specific kernel and device configuration to optimise power consumption.

At present, the tool is open source and is capable of running on a variety of platforms and hardware including both Android and GNU/Linux [5].

By using the “-workload” flag, it is possible to profile a specific binary file to analyse the power usage of the program relative to the rest of the system. This is designed to be used to benchmark a program against others. For accuracy, Powertop recommends the program to be used for multiple iterations to improve the accuracy of the model. Such information however, has limited value for an energy conscientious developers as the information provided by the tool does not tie directly against code. This makes it difficult to estimate which specific portions of code lead to problematic power behaviour.

```

PowerTOP 2.8      Overview  Idle stats  Frequency stats  Device stats  Tunables
>> Bad           VM writeback timeout
Bad              Autosuspend for USB device Touchscreen [ELAN]
Good             NMI watchdog should be turned off
Good             Enable SATA link power management for host0
Good             Enable SATA link power management for host3
Good             Enable SATA link power management for host1
Good             Enable SATA link power management for host2
Good             Enable Audio codec power management
Good             Bluetooth device interface status
Good             Autosuspend for unknown USB device 1-1 (8087:8001)
Good             Autosuspend for USB device Integrated_Webcam_HD [CN09GTFM72487526B88DA00]

```

Figure 7: A snapshot of some of the tuning capabilities present within Powertop.

```

20 * 100 = 2900 MHz max turbo 5 active cores
20 * 100 = 2900 MHz max turbo 4 active cores
20 * 100 = 2900 MHz max turbo 3 active cores
20 * 100 = 2900 MHz max turbo 2 active cores
30 * 100 = 3000 MHz max turbo 1 active cores
cpu2: MSR_CONFIG_TDP_NOMINAL: 0x00000018 (base_ratio=24)
cpu2: MSR_CONFIG_TDP_LEVEL_1: 0x0000003c (PKG_MIN_PWR_LVL1=0 PKG_MAX_PWR_LVL1=0 LVL1_RATIO=6 PKG_TDP_LVL1=60)
cpu2: MSR_CONFIG_TDP_LEVEL_2: 0x00000000 ()
cpu2: MSR_CONFIG_TDP_CONTROL: 0x00000000 (lock=1)
cpu2: MSR_TURBO_ACTIVATION_RATIO: 0x80000017 (MAX_NON_TURBO_RATIO=23 lock=1)
cpu2: MSR_NHM_SNB_PKG_CST_CFG_CTL: 0x1e008408 (Undemote-C3, Undemote-C1, demote-C3, demote-C1, locked: pkg-cstate-limit=8: unlimited)
cpu8: MSR_IA32_ENERGY_PERF_BIAS: 0x0000000f (powersave)
cpu8: MSR_RAPL_POWER_UNIT: 0x000a0e03 (0.125000 Watts, 0.000061 Joules, 0.000977 sec.)
cpu8: MSR_PKG_POWER_INFO: 0x00000078 (15 W TDP, RAPL 0 - 0 W, 0.000000 sec.)
cpu8: MSR_PKG_POWER_LIMIT: 0x00420c0800000078 (locked)
cpu1: PKG Limit #1: Enabled (15.000000 Watts, 28.000000 sec, clamp Enabled)
cpu8: PKG Limit #2: Enabled (25.000000 Watts, 0.002441* sec, clamp Disabled)
cpu8: MSR_PP0_POLICY: 9
cpu8: MSR_PP0_POWER_LIMIT: 0x00000000 (Unlocked)
cpu8: Cores Limit: Disabled (0.000000 Watts, 0.000977 sec, clamp Disabled)
cpu8: MSR_PP1_POLICY: 13
cpu8: MSR_PP1_POWER_LIMIT: 0x00000000 (Unlocked)
cpu8: GFX Limit: Disabled (0.000000 Watts, 0.000977 sec, clamp Disabled)
cpu8: MSR_IA32_TEMPERATURE_TARGET: 0x05600000 (105 C)
cpu8: MSR_IA32_PACKAGE_THERM_STATUS: 0x08430000 (38 C)
cpu1: MSR_IA32_THERM_STATUS: 0x08440000 (37 C +/- 1)
cpu1: MSR_IA32_THERM_STATUS: 0x08430000 (38 C +/- 1)
cpu2: MSR_PKG_C3_IRTLL: 0x000000842 (valid, 67584 ns)
cpu2: MSR_PKG_C6_IRTLL: 0x000000073 (valid, 117760 ns)
cpu2: MSR_PKG_C7_IRTLL: 0x000000891 (valid, 148480 ns)
cpu2: MSR_PKG_C8_IRTLL: 0x000000804 (valid, 233472 ns)
cpu2: MSR_PKG_C9_IRTLL: 0x000000945 (valid, 332000 ns)
cpu2: MSR_PKG_C10_IRTLL: 0x00000090f (valid, 506880 ns)

```

| Core | CPU Avg_MHz | Busyn% Bzyn_MHz | TSC_MHz | IRQ  | SMI  | CPUc1 | CPUc3 | CPUc6 | CPUc7 | CoreTmp | PkgTmp | GFXHz | GFXHzc6 | GFXHzc7 | PkgHzc2 | PkgHzc3 | PkgHzc6 | PkgHzc7 | PkgHzc8 | PkgHzc9 | PkgHzc10 | PkgWatt | CorWatt | GFXWatt |      |
|------|-------------|-----------------|---------|------|------|-------|-------|-------|-------|---------|--------|-------|---------|---------|---------|---------|---------|---------|---------|---------|----------|---------|---------|---------|------|
| -    | -           | 171             | 6.97    | 2455 | 2395 | 36382 | 4     | 36.67 | 3.32  | 0.75    | 52.28  | 37    | 38      | 89.65   | 600     | 10.07   | 1.52    | 0.09    | 9.88    | 0.00    | 0.00     | 0.00    | 3.42    | 1.50    | 0.08 |
| 0    | 0           | 277             | 13.62   | 2404 | 2395 | 33905 | 1     | 62.31 | 1.20  | 0.33    | 22.45  | 37    | 38      | 89.65   | 600     | 10.07   | 1.52    | 0.09    | 9.88    | 0.00    | 0.00     | 0.00    | 3.42    | 1.50    | 0.08 |
| 0    | 2           | 114             | 4.49    | 2537 | 2395 | 739   | 1     | 71.44 |       |         |        |       |         |         |         |         |         |         |         |         |          |         |         |         |      |
| 1    | 1           | 96              | 3.91    | 2456 | 2395 | 1316  | 1     | 7.46  | 5.36  | 1.17    | 82.10  | 36    |         |         |         |         |         |         |         |         |          |         |         |         |      |
| 1    | 3           | 147             | 5.87    | 2509 | 2394 | 522   | 1     | 5.49  |       |         |        |       |         |         |         |         |         |         |         |         |          |         |         |         |      |

Figure 8: Turbostat is a simple program that provides information relating to C state transitions and power consumption for the CPU and GPU.

### 2.4.2.2 Turbostat

The Turbostat tool is limited in capability relative to the Powertop tool listed previously. It was designed by Intel and provides accurate information regarding the CPU. In particular, the program provides specific information of the C state and power consumption of a given CPU core. Figure 8 shows the statistics offered by the program that are generated from reading CPU information [6].

## 2.5 Related Work

### 2.5.1 ENTRA 2012-2015

In addition to the above physical measurements and utilities used to profile power for x86 systems, new strategies are actively researched to form generic modelling solutions that are not tied to hardware. For example, the Whole Systems ENergy TRAnsparency project (ENTRA) [24] was designed to explore some of these techniques. Below are some of the findings.

#### 2.5.1.1 Common Assertion Language

This Common Assertion Language [25] was designed to introduce a concept of extending information relating to energy and resource consumption throughout the software stack such

```

// assert energy < cost(20)
// assert a >= 0
int factorial(int a) {
    Int result = a; //(Line 1)
    // assert max_loop iterations < 1000
    while(a > 0) { //(Line 2)
        result = result * (a - 1); //(Line 3)
        a = a - 1; //(Line 4)
    }
    return result;
}

```

**Figure 9: A simple program to compute factorial.**

as at instruction level or source level. The theory is to create a language agnostic approach where software developers can provide a specification with respect to resource consumption for their program. The developer would achieve this by restricting program behaviour using pre-conditions, post-conditions and assertions. In addition, the paper refers to additional environmental assertions that specify the power consumption, accuracy, bound restrictions and time restrictions.

For the code in Figure 9, a developer can specify a set of energy bounds for the instructions such as, lines (1), (2) having a cost of 1 whilst (3) and (4) having a cost 2. Based on this information, it is possible to produce a lower and upper bound for inputs that satisfy the assertion for energy cost. For example, in this case, for an input value of 0, the minimum cost of the above function would simply be 2. This is because the above code would simply run instructions 1 and 2 once respectively. Additionally, for an input value of 3, the cost would be the following:

$$1 + 1 + 2 + 2 + 1 + 2 + 2 + 1 + 2 + 2 + 1 = 17$$

In the above example, line 1 is run once, line 2 four times and the lines 3 and 4 three times each. This is the largest input value that satisfies the above cost estimate. As such, this is the upper bound in terms of input to the function that satisfies the cost assertion. This information could subsequently be fed back to the developer to estimate the utility of the function for certain inputs.

In practice, the above procedure occurs at a lower level in the software stack by performing the code analysis at an instruction level. This is done whilst retaining the assertions made in the original higher source code. The developer in addition would provide explicitly the cost of each instruction for a given platform. By using these assertions to form constraints, a static analyser can check whether a specific function satisfies its energy constraints or assertions. If the input is unspecified, it would also be able to return specific bounds to the input that satisfy the given energy budget. The reason the process provides an upper-bound is because the cost of an instruction can vary depending on the circumstances and environment. For example, if a function calls an external or non-analysed function, the developer would need to provide an estimate of the energy cost of the given call.

```

a = b * c + g;
d = b * c * e;

```

(a) Non-optimised common sub-expression elimination

```

f = b * c;
a = f + g;
d = f * e;

```

(b) Optimised common sub-expression elimination

**Figure 10:** In the example, the code  $b*c$  is optimised to a new variable. This avoids recomputing  $b*c$  multiple times saving energy.

This approach using static analysis has a lot of benefits from a developer perspective. This is because the developer can specify and guarantee the energy consumption of a given program within a certain constraint by using assertions and a contract. On the other hand, the tool does present disadvantages as it requires calibration from developer to estimate a per instruction energy cost for the platform.

### 2.5.1.2 Compiler Optimisation and Power Trade-offs

Most current compiler optimisation improve the energy consumption of a given program [26]. These reduce energy usage by decreasing the amount of work performed. Examples of such optimisation include Dead-code elimination and common sub-expression elimination. This can be observed in Figure 10. However, not all optimisation necessarily follow this principle. A few optimisations require context of the problem in order to be efficient for energy consumption. One example includes the use of compilers that optimise the code layout that take advantage of memory hierarchies. By introducing greater spatial and temporal locality in data accesses, it is possible to improve power usage by reducing memory transitions and improving the speed of the program. Such techniques however, do introduce additional code which can result in energy penalties.

An example of such a case is demonstrated in Figure 11. In this case, a standard for loop is broken down into smaller sub loops. This is so that data used within the inner loop remains in the cache longer. However, this can introduce more branching which can result in an overhead. For small input sizes, this additional branching can increase the power consumption for the given program.

### 2.5.1.3 Superoptimization

Another strategy for optimising code for energy includes a technique called “Superoptimisation” [26]. This technique was developed by Dr. Massalin in 1987 [28] to optimise instruction code. The basic idea of this approach is to take a portion of code and to enumerate all viable options of instruction code and checking whether it is equivalent to that produced from a given piece of code. This equivalence is achieved by first testing the function with test values and seeing if it produces the correct output. If the test passes, the function is verified using program verification techniques such as SMT solvers.

```

int A[MAX_VAL, MAX_VAL], B[MAX_VAL, MAX_VAL];
for (int i = 0; i < MAX_VAL; i++) {
    for (int j = 0; j < MAX_VAL; j++) {
        A[i, j] = A[i, j] + B[i, j];
    }
}

```

(a) Unoptimized loop

```

int A[MAX_VAL, MAX_VAL], B[MAX_VAL, MAX_VAL];
for (int i=0; i< MAX_VAL; i+=BLOCK_SIZE) {
    for (int j=0; j< MAX_VAL; j+=BLOCK_SIZE) {
        for (int k=i; k<i+BLOCK_SIZE; k++) {
            for (int l=j; l<j+BLOCK_SIZE; l++) {
                A[k,l] = A[k,l] + B[l, k];
            }
        }
    }
}

```

(b) Optimized loop

**Figure 11: The above code is an example of loop tiling from Intel [27].**

This principle is demonstrated by Figure 12. In the original paper on Superoptimization [28], Dr. Massalin translated the small function written listed in the figure. When compiled with a SUN-C compiler, the resulting code was achieved in approximately 9 instructions. Instead, Dr. Massalin found a functionally equivalent program using just 4 instructions. This was achieved by exhaustively exploring the instruction set space to find functionally equivalent code. The code at the bottom of the figure shows the generated code which achieved the result in simply 4 instructions. This was achieved by exploiting features of two's complement. The main advantage of this approach is that it completely eliminated costly jump instructions.

This technique currently is designed for optimising code for memory and speed; however, by applying a cost function for the instructions produced, the technique can be extended to optimise small code pieces for energy efficiency.

In terms of limitations, the above technique can be quite impractical for anything but simple programs. This is because the search space can become impractically large for platforms with complex instruction sets and large programs. In addition, the technique according to Dr. Massalin does not work for pointer based languages [28]. This is because the pointer to memory can point to anywhere in memory. As such, pointer based operations can exponentially increase the search space for the instructions to the point of impracticality. This is especially the case on modern systems which use larger memory sizes and caches.



```

int signum(int x)
{
    if (x > 0)
        return 1;
    else if (x < 0)
        return -1;
    else
        return 0;
}

(x in d0)
add.1 d0, d0 | add d0 to itself
subx.1 d1, d1 | subtract (d1 + Carry) from d1
negx.1 d0      | put (0 - d0 - Carry) into d0
addx.1 d1, d1 | add (d1 + Carry) to d1
(signum(x) in d1)

```

**Figure 12:** The above code is the example presented by Dr. Massalin in his original paper on Superoptimization [28].

#### 2.5.1.4 Thermal trade-off

When people consider programs and optimise behaviour, it can be easy to ignore other sources of energy side effects that are consequences of their program. One of the major uses of energy within data-centers are the cooling systems necessary to manage the immense heat generated by the hardware. Software developers, are often unaware of the thermal consequences of the software we write. One example of such a trade-off includes register allocation in compilers. Traditionally, when deciding which register to allocate data, compilers often choose the earliest free register. This introduces potential thermal consequences as the early registers get frequently used and flushed. This in itself could cause more energy inefficiency relative to using other registers as more power is required to cool the device. According to the paper, in a sample system with 12 registers with heavy usage, 5% of energy could have been saved by following a even register allocation as opposed to simply picking the last free register [29]. Interestingly, this result contrasts with traditional advice with respect to idle hardware. In this case, it can be more energy efficient to schedule use between active components to reduce thermal energy use.

#### 2.5.2 eProf

The eProf academic project was designed to estimate energy use of programs for Linux and Android based platforms [30]. It achieves this without annotating the original source code.

##### 2.5.2.1 Asynchronous vs Synchronous

Within the paper, the authors observe the nature of energy consumption of a system and breaks them down towards synchronous and asynchronous components. For example, with the CPU and memory hierarchy systems of a computer are synchronous components. This

means that as code executes, both components consume energy as the instructions are executed.

In contrast, asynchronous devices are far more complex. These are components such as the network, disk and other IO forms. When a program interacts with such components, the program execution becomes complex. In the case of Linux the method of IO works as following:

1. Code executes some system call to the kernel and requests some device resource.
2. The kernel allocates a device request and places the request on a request queue.
3. The thread is blocked and put to sleep until the resource is returned.
4. The device at this point has it's own schedule for the packets and eventually executed.
5. The device processes the request synchronously consuming energy.
6. Upon finishing, the device sends the data back to the kernel and the thread is woken.

The nature of steps 3-6 can make it difficult to estimate how much energy is actually being consumed directly by the program. This is for several reasons including the fact that there is energy consumed when the device is servicing a different request. Equally, the device may consume additional energy due to the process being switched out whilst the requested process is sleeping.

#### **2.5.2.2 Profiling implementation**

eProf aims to alleviate some of the main limitations of the above synchronous and asynchronous devices by creating an energy model that estimates how much energy each device consumes. This model is trained initially by a series of micro-benchmarks such as SPEC CPU 2006. In this way, a linear model is generated of the energy consumed by the CPU.

For asynchronous devices such as disks and networks, the model is generated by stressing the disk and generating data between request duration and energy. This can be then used to estimate the energy consumption of these various devices by relating it to how much data is consumed from the process. According to the paper, the relationship between data accessed and energy consumes is linear.

Based on this approach, eProf manages to break down an application and function by gathering specifically how much energy each function consumes based on the models they generate.

The main limitation of this approach involves the calibration step required for the profiler. In this case, benchmarks need to be run on the machine to generate an accurate model for the energy consumption of a target platform. It also requires specific modification of the Linux kernel.

#### **2.5.3 Energy Formal Definitions**

Another project explored in academia is the concept of formalising definitions for estimating the energy consumption. Some of these models are explored in this section.

### 2.5.3.1 Java Based Energy Formalism

An example of such a framework includes [31] which specifies a model for estimating the energy consumption of a java-based application. This is broken down as follows.

$$E_{Component} = E_{Computational} + E_{Communication} + E_{Infrastructure}$$

In this case, computational costs are the energy costs consumed by the CPU, memory and IO operations. The communication cost is the amount of energy estimated due to exchanging data over the network. The final infrastructure cost which is the sum of the overhead incurred by the OS and JVM.

Within the paper, the author relies on previous research which has shown that the energy consumed for wireless communication is proportional to the size of transmitted and received data.

### 2.5.3.2 Energy Application Model

Another project [32] explored a more generic approach for specifying the energy consumption of a program. Within the paper, the author breaks down an application's energy consumption into the following formula.

$$E_{App} = E_{Active} + E_{Wait} + E_{Idle}$$

In this case,  $E_{Active}$  corresponds to the time in which the application is running on the underlying system.  $E_{wait}$  corresponds to the time spent waiting for another component of subsystem. Finally,  $E_{Idle}$  corresponds to the time in which the equipment is not performing any work for the application.

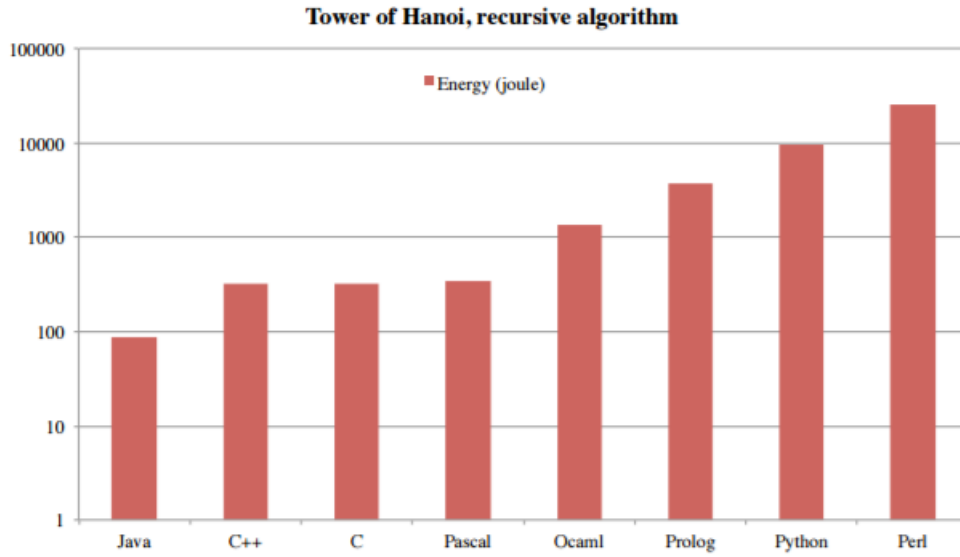
## 2.5.4 Impact of language, Compiler, Optimisations

Some of the differences in energy cost by language and algorithm choice are explored here. The author originally calculated these impacts by comparing various programs in different languages using a recursive implementation of the Tower of Hanoi problem. These measurements were gathered on a Dell Precision T3400 workstation computer with an Intel Core 2 Quad processor (Q6600), and running Ubuntu Linux version 11.04. Measurements were gathered using PowerAPI. Further details of this tool is linked in Section 2.7.6.

### 2.5.4.1 Choice of Language

From the results presented in Figure 13, it is clear that there is a drastic difference between energy consumption across scripting languages such as Perl and Python. The difference between the energy consumption of Perl, Python and Prolog can be up to 1000% higher than that of native code. The reason for this is additional energy consumed simply to interpret the scripting language.

Another interesting aspect is the difference between Virtual Machine languages such as OCaml, Java and C/C++. In some cases, such as with Java, the code compiled is significantly more energy efficient than native code. This is because of optimisations and code prediction provided by the Java Virtual Machine. One optimisation in this case, is the use



**Figure 13: A comparison of Tower of Hanoi recursive implementation across various languages. The energy consumed (Joules) is marked in a  $\log_{10}$  scale. This graph is from a PhD thesis on the effects of energy consumption on software systems [33].**

of JIT (just-in-time) to detect repetitive code which is further compiled down to native code.

#### 2.5.4.2 Relation of execution time and energy consumption

Another interesting observation from the benchmark was the comparison of energy consumption against time. In this case, the author performed a series of benchmarks using the Tower of Hanoi problem as a sample program. This is a CPU bound benchmark. This linear relationship can be observed in Figure 14. The author however, stresses that this relationship is not universal across programs. With other more complex programs such as video decoding and the like having more complex energy to execution time. This is because in more advanced cases, the CPU is not the limiting factor and could run at variable clock speeds. At times, it is beneficial to run at lower clock speeds for a larger execution time as this would consume less energy than running at a higher clock speed for a shorter execution time.

#### 2.5.4.3 Impact of Optimisation flags

As observed earlier, the choice of optimisation flags can greatly impact the energy consumption of a program. When compiling with the O2 flag for example, the author noted a saving of approximately 27% relative to an non-optimised version. Similarly, the choice in O3 flag also produces a significant saving in energy over O2. In this case producing approximately 87% energy consumption relative to O2.

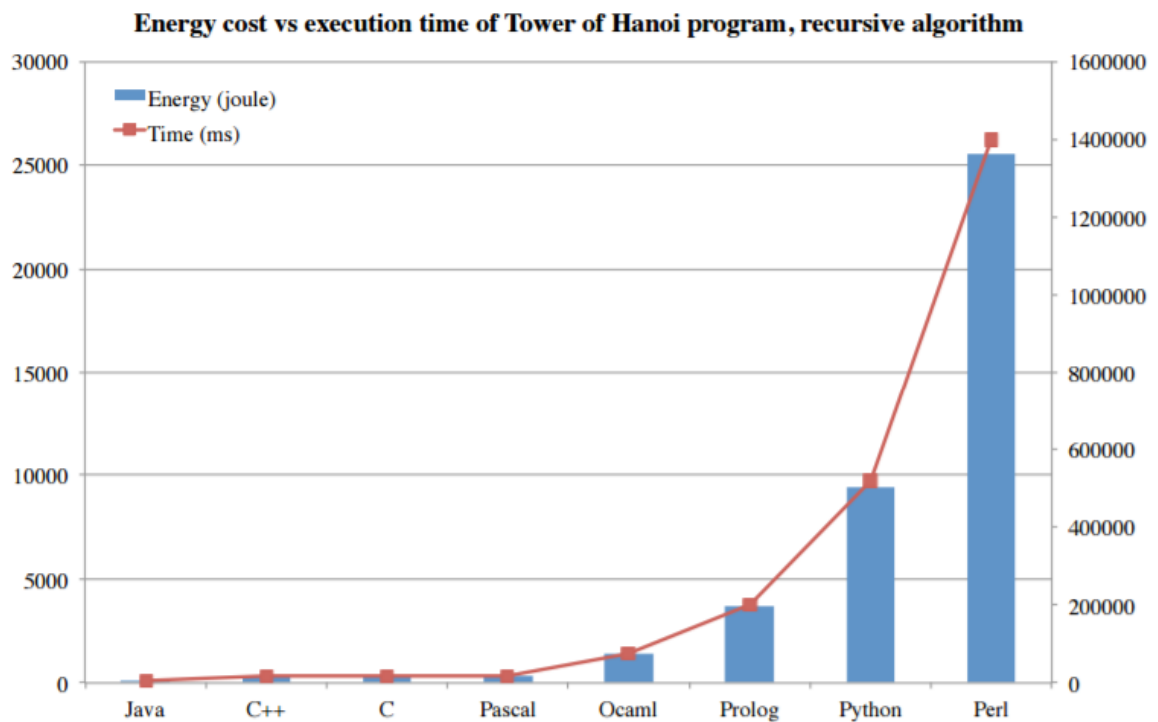
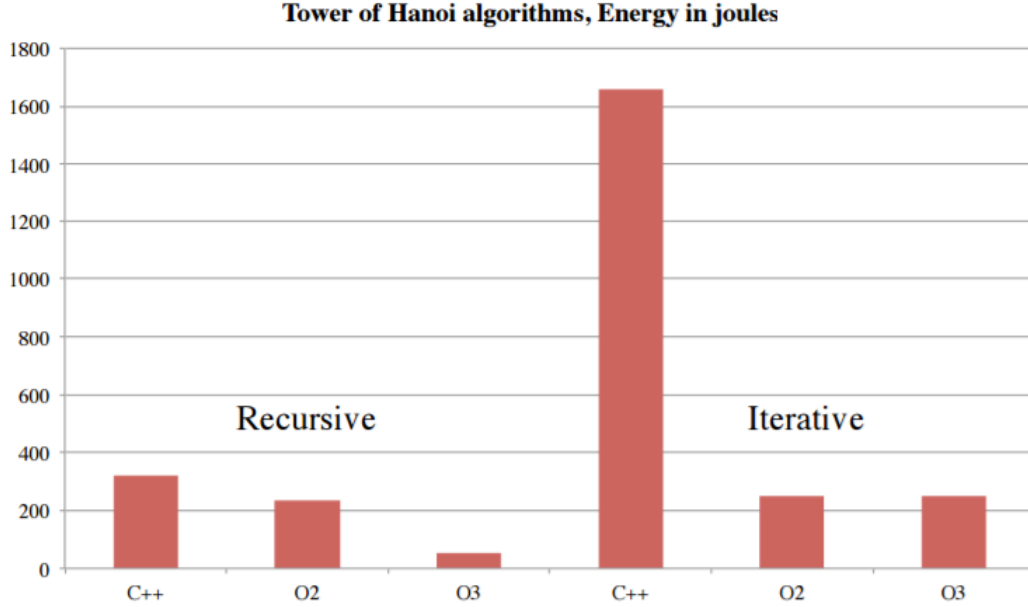


Figure 14: A comparison of Tower of Hanoi energy consumption across various languages. This graph is from a PhD thesis on the effects of energy consumption on software systems [33].



**Figure 15: A comparison of Tower of Hanoi using iterative and recursive along with various optimisation flags.**

#### 2.5.4.4 Choice in Algorithm

The tower of Hanoi is a perfect example for demonstrating the energy consequence of algorithms. In the case of this benchmark, the author compares the iterative version against a recursive implementation.

Figure 15 is interesting as it shows how much more energy the iterative version consumes. In this case, the iterative version consumes approximately 400% more energy relative to the standard recursive implementation. Using the O2 flag, both implementations end up matching in terms of energy consumption. However, only the O3 flag achieves better efficiency with the recursive implementation.

## 2.6 Similar Tools

This section explores a variety of energy profiling tools used on other platforms. The idea in this case is to understand their strengths, weaknesses and capabilities.

### 2.6.1 AEON

The AEON plugin [34] for IntelliJ provides power profiling of Android Apps directly within the IDE. This tool works by utilising the ADB (Android Debug Bridge) to get power statistics directly from a connected android device. The power statistics are provided by a profiler from Qualcomm called Trepro [35]. Through this mechanism, the tool is able to attain accurate information relating to CPU/GPU and network activity. It also grants battery discharge rates for compatible Android devices.

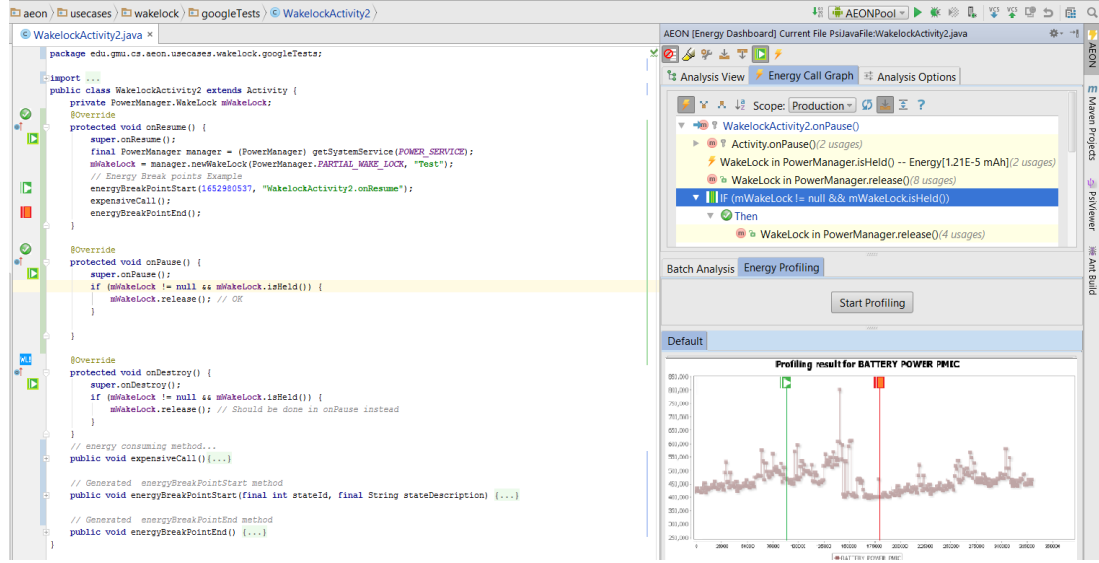


Figure 16: This Figure is from the AEON plugin page [34].

The way this tool profiles the code is by injecting network profiling metrics at energy points. Upon running the tool on the Android device, statistics are stored at specific points by the profiler. Following this, the tool is able to read the values at these specific profile points to plot a graph.

In terms of usability and utility for a developer the above program provides an ideal model as to what this project seeks to achieve. This is because the tool enables the capability to easily visualise the energy cost of a given function between certain portions of code. The tool is able to relate this data to the original source code with suggestions made for possible improvements to the code base. PowerKap aims to recreate these visualisations. In particular the tool makes use of common debugging features such as breakpoints and energy graphs for users to intuitively select particular points in the energy profile. These features can clearly be seen in Figure 16.

## 2.6.2 Visual Studio

On Windows based platforms, Microsoft has made recent strides to improve the energy profiling capability for their platforms. This was part of their move towards ensuring battery efficiency for windows store apps on their mobile platforms. For this reason, they introduced as part of Visual Studio 2013 [36], an application that is capable of profiling various CPU and network metrics. These can be seen in Figure 18. This project again aims to enable developers to be aware of how much energy their specific application is using. It is also able to achieve this across a variety of supported languages such as C#, C++, Visual Basic and JavaScript.

In terms of usability and utility, this program is a particularly good demonstration of how to integrate such tools in the development environment. The tool requires minimal installation beyond that which already comes with the SDK (Software Developer Kit) and

```

if (performance && performance.mark) {
    performance.mark(markDescription);
}

```

**Figure 17:** In order to use Visual Studio’s energy consumption tool, the developer needs to annotate their code with specific logging points such as the JavaScript code listed below.

IDE (Integrated Developer Environment). It also requires minimal annotation to the code whilst preserving functionality. The main disadvantage of the above application is obviously that it is limited purely to Windows and the fact that it does not contain statistics about other hardware such as other peripherals or the GPU.

As of Visual Studio 2017, the tool appears to have been deprecated due to lack of use and the fact that the tool gave inaccurate values for energy use [37].

## 2.7 Gathering Energy Measurements

The first stage with any power efficiency tool is to be able to capture how much energy is consumed. This is normally not a simple task as many of power detection approaches are not normally unified behind a single implementation or interface. Below are some of the available techniques and rationale behind various mechanisms of capturing energy information.

### 2.7.1 Module Specific Registers (MSR)

On Intel platforms, it is possible to access the energy counters within a CPU directly [38]. This approach works by utilising a module provided by the kernel called “MSR” (Module Specific Registers). Once loaded, a user can read RAPL energy statistics directly with root or sufficient read/write privileges. These can be accessed through the interface at “/dev/cpu/CPUNUM/msr” [39].

In terms of advantages, such an approach presents an advantage over other methods as it records directly results from the registers without any influence from the kernel. On Intel platforms, there is a wealth of information provided by this interface with respect to energy information.

#### 2.7.1.1 Running Average Power Limits (RAPL) Interfaces

The energy and power interface presented by RAPL is specific to certain areas of the computer. These areas are referred to as domains. These are listed below.

##### 1. Package

The first domain available to measure is package specific. This is for support on multi-CPU machines such as servers. The statistics generated in this layer are specific to the CPU package as a whole. Within this package, the energy counters are normally updated every millisecond. However, under high CPU loads, the package may only be updated every 60 seconds.



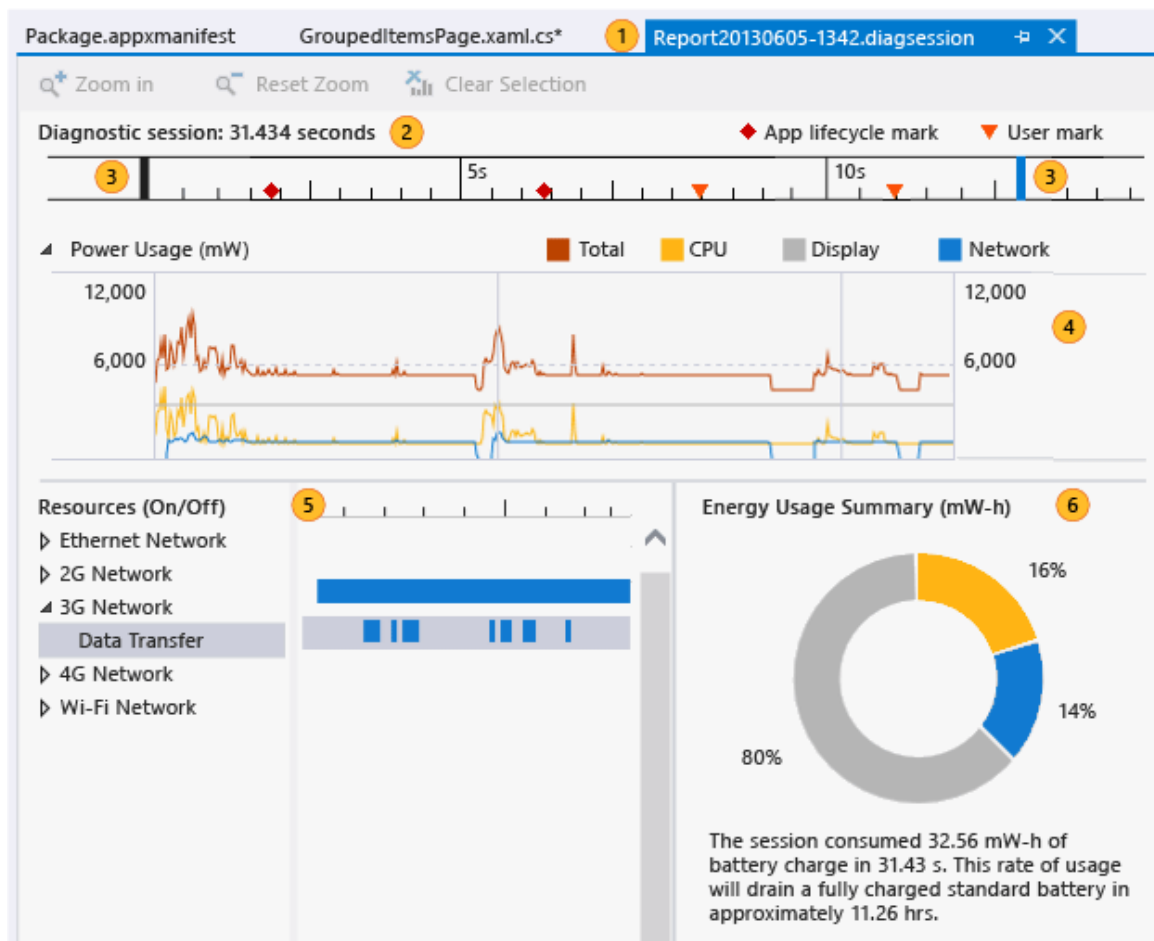


Figure 18: Upon profiling the respective portion of code, the developer is given useful visual information relating to CPU, network, data and display usage.

2. PP0 (Core)

This domain refers to specifically the stats of all the CPU cores within a package and L1 and L2 caches [40]. The energy counter information presented in this layer is updated every millisecond.

3. PP1 (Uncore)

The PP1 domain is only available on consumer devices. This interface is identical to the core domain above but refers specifically to the portions of the chip referred to as Uncore. This normally refers to other parts of the chip outside the core such as the integrated GPU, L3 cache rings and memory controller [40]. According to the documentation, this domain refers to the “power plane of a specific device in the Uncore”.

4. DRAM

On Server and certain architectures, Intel offers the ability to additionally capture energy consumption for the main memory controller.

Below is a list of capabilities that are available through the MSR interface. These relate to controlling the power consumption of a platform along with several different counters.

1. Power Limit

This interface allows the capability of adjusting various aspects of the system such as specifying specific power limits, adjusting time windows (seconds) for updating power counters. This is necessary because power is averaged over a period of time.

2. Energy Status (Joules)

This interface allows the ability to gather information from various energy counters.

3. Perf Status (Optional)

This optional interface is not available on all platforms and provides specific information with respect to performance effects that are caused by power limits.

4. Power Info (Optional) (Watts)

In addition to the above, this interface grants a range of parameters with respect to power for specific domains.

5. Policy

4-bit interface that allows hints to hardware on how to subdivide power to specific domains.

One of the main disadvantages of this approach to record such information is the fact that it requires root, along with configuration by installing the MSR module. It also requires intimate knowledge and interaction with the CPU which can be unsuitable for profiling unknown, buggy or insecure code. In addition, such a tool requires specific information with respect to the platform and is difficult to extend. For example, to read various MSR registers requires specific domain knowledge such as the access bits in specific registers. For this reason, PowerKap avoids such direct approaches as there are alternative user space mechanisms that provide equivalent information.

### 2.7.2 Perf

The Perf system is a performance analysing subsystem available for GNU/Linux that allows users to instrument specific CPU counters, as well as kernel objects and information [41]. It is developed in parallel with the GNU/Linux Kernel and can be used by simply installing additional packages such as “GNU/Linux-tools-common”. Once installed, the user can gather performance counter information by making use of the “Perf” command. In order to use the system for power profiling, it requires a kernel version greater than version 3.14 along with platform specific support [42].

There are many advantages of using such an approach for an energy profiler. Namely, supplementary information can be gathered from the system including memory cache misses, sleep times as well as granularity of how much time the CPU spends in each function. It is also largely compatible across a variety of platforms including products designed by IBM, AMD and Intel.

On the other hand, such an approach has certain disadvantages. For example, this interface requires intensive calibration by users relative to other approaches. For example, to use this interface requires additional packages as described earlier. In order to allow user programs to access energy statistics from user space, requires either root privileges or configuration that sets a kernel “paranoia” level value to less than 1. This paranoia level describes access writes to the various Perf subsystem. In addition to the above, support for various platforms often depends on kernel support which can be slow for most non-rolling distributions such as Ubuntu or Red Hat Enterprise Linux. Often such Operating Systems rely on much older kernel versions for stability reasons. Security and hardware support are then backported to such kernels as required.

### 2.7.3 PAPI (Performance Application Programming Interface)

The PAPI interface is similar to the Perf event system and provides an easy interface for users to profile their code [43]. It also provides supplementary information such as cache misses and counter specific information. To use the PAPI interface the package “papi-tools” is required. This tool is again limited by the fact that it requires additional installs and configuration that is specific to the platform and distro. For example, to make use of MSR information, the MSR module also needs to be loaded into the kernel.

### 2.7.4 HWMON

Another module interface that often gets compiled in the GNU/Linux kernel is the HWMON interface [44]. This stores various energy and thermal information for respective CPU and sensors. Once the module is loaded and compiled, it is possible to read various thermal and energy attributes at “/sys/class/hwmon”. This interface is one of the few methods of gathering energy information for AMD based platforms. See Figure 19 for details of hardware support.

### 2.7.5 Intel Powercap

As with HWMON, the GNU/Linux kernel often comes with a built in Intel Powercap module [45]. This module allows the MSR registers to be read from user space. This is

| Name                            | Family | Model   | TDP       | Energy | hwmon                 | perf_event        | PAPI  |
|---------------------------------|--------|---------|-----------|--------|-----------------------|-------------------|-------|
| Family 15h Bulldozer            | 21     | 1       | Yes       | No     | 3.0 (512d1027a)       | no                | hwmon |
| Family 15h Piledriver           | 21     | 2       | Yes       | No     | 3.0 ( 512d1027a)      | no                | hwmon |
| Family 15h Piledriver/Trinity   | 21     | 10, 13  | Yes       | No     | no                    | no                | hwmon |
| Family 15h Steamroller "Kaveri" | 21     | 30h     | CRIT only | No     | 3.17 (0a0039ad541)    | no                | hwmon |
| Family 15h Excavator "Carrizo"  | 21     | 60h-6fh | Yes       | Yes    | 4.3 (5dc087254acf)    | 4.6 (c7ab62bfbe0) | hwmon |
| Family 15h ????                 | 21     | 70h-7fh | Yes       | Yes    | 4.5 (eff2a94598)      | no                | hwmon |
| Family 16h Jaguar               | 22     | 0       | ?         | ?      | 3.10 (22e32f4f5)      | no                | hwmon |
| Family 16h Jaguar "Mullins"     | 22     | 30h     | Yes       | Maybe  | 3.18 (0bd52941586b3b) | no                | hwmon |

**Figure 19: Table of interfaces that the AMD supports for energy and power [47].**

achieved by reading the energy attributes within the “/sys/class/powercap”. Within this directory, there are an assortment of subdirectories that reflect the MSR and RAPL domain package structure listed previously with MSR. In addition, new versions of profilers such as PAPI 5.5 makes extensive use of this interface to gather energy information for a CPU package.

### 2.7.6 PowerAPI

The PowerAPI [46, 33] is a hardware profiling tool designed by the Lille University of Science and Technology. This API is designed in Scala and is capable of capturing various energy hardware information. At present, it is designed as a research project and as such is not available by default on most Linux platforms. It makes use of various process specific information such as energy and thermal information. This interface at present hasn’t been tested and aspects such as the Powercap interface are still in development.

This tool requires a calibration stage and applications must be linked against it to make use of it.

### 2.7.7 Summary

It is clear that there are a large assortment of methodologies for measuring various energy and power sensors. This can be difficult to design against as each of these interfaces require their own specific configuration or modules installed. For example, to use the Perf system, you need to install additional Perf specific packages and for the MSR interface, it is necessary to know specific registers. Figure 20 is a handy table with respect to CPU energy support and interface support for Intel platforms.

| Name                                    | Family | Model   | package | PP0 (usually cores) | PP1 (usually GPU) | DRAM | PSys | powercap            | perf_event          | PAPI                 |
|---|--------|---------|---------|---------------------|-------------------|------|------|---------------------|---------------------|----------------------|
| Sandybridge                             | 6      | 42      | Y       | Y                   | Y                 | N    | N    | 3.13 (2d281d8196)   | 3.14 (4788e5b4b23)  | yes                  |
| Sandy Bridge EP                         | 6      | 45      | Y       | Y                   | N                 | Y    | N    | 3.13 (2d281d8196)   | 3.14 (4788e5b4b23)  | yes                  |
| Ivy Bridge                              | 6      | 58      | Y       | Y                   | Y                 | N    | N    | 3.13 (2d281d8196)   | 3.14 (4788e5b4b23)  | yes                  |
| Ivy Bridge EP ("Ivy Town")              | 6      | 62      | Y       | Y                   | N                 | Y    | N    | no                  | 3.14 (4788e5b4b23)  | yes                  |
| Haswell                                 | 6      | 60      | Y       | Y                   | Y                 | Y    | N    | 3.16 (a97ac35b5d9)  | 3.14 (4788e5b4b23)  | yes                  |
| Haswell ULT                             | 6      | 69      | Y       | Y                   | Y                 | Y    | N    | 3.13 (2d281d8196)   | 3.14 (7fd565e27547) | yes                  |
| Haswell                                 | 6      | 70      | Y       | Y                   | Y                 | Y    | N    | 4.6 (462d8083f)     | 4.6 (e1089602a3bf)  | yes                  |
| Haswell EP                              | 6      | 63      | Y       | ?                   | N                 | Y    | N    | 3.17 (64c7569c065)  | 4.1 (64552396010)   | yes                  |
| Broadwell                               | 6      | 61      | Y       | Y                   | Y                 | Y    | N    | 3.16 (a97ac35b5d9)  | 4.1 (44b11fee517)   | ?                    |
| Broadwell-H                             | 6      | 71      | Y       | Y                   | Y                 | Y    | N    | 4.3 (4e0bec9e83)    | 4.6 (7b0fd569303)   | ?                    |
| Broadwell-DE                            | 6      | 86      | Y       | ?                   | ?                 | Y    | N    | 3.19 (d72be771c5d)  | 4.7 (31b84310c79)   | ?                    |
| Broadwell EP                            | 6      | 79      | Y       | ?                   | ?                 | Y    | N    | 4.1 (34dfa36c04c)   | 4.6 (7b0fd569303)   | ?                    |
| Skylake                                 | 6      | 78      | Y       | ?                   | ?                 | Y    | ?    | 4.1 (5fa0fa4b01)    | 4.7 (dcee75b3b7f02) | ?                    |
| Skylake H/S                             | 6      | 94      | Y       | ?                   | ?                 | Y    | ?    | 4.3 (2cac1f70)      | 4.7 (dcee75b3b7f02) | ?                    |
| Skylake Server                          | 6      | 85      | Y       | ?                   | ?                 | Y    | ?    | no                  | 4.8???              | 4.8 (348c5ac6c7dc11) |
| Kabylake                                | 6      | 142,158 | Y       | ?                   | ?                 | Y    | ?    | 4.7 (6c51cc0203)    | no                  | no                   |
| Knights Landing                         | 6      | 87      | Y       | ?                   | ?                 | Y    | N    | 4.2 (6f066d4d2)     | 4.6 (4d120c535d6)   | ?                    |
| Knights Mill                            | 6      | 133     | Y       | ?                   | ?                 | Y    | N    | ?                   | 4.9 (36c4b6c14d20)  | ?                    |
| Atom Goldmont (Apollo Lake?) "Broxtown" | 6      | 92      | Y       | Y                   | Y                 | Y    | N    | 4.4 (89e7b2553a)    | 4.9 (2668c6195685)  | no                   |
| Atom Braswell                           | 6      | 76      | ?       | ?                   | ?                 | ?    | N    | 3.19 (74af752e4895) | no                  | no                   |
| Atom Tangier                            | 6      | 74      | ?       | ?                   | ?                 | ?    | N    | 3.19 (74af752e4895) | no                  | no                   |
| Atom Annidale                           | 6      | 90      | ?       | ?                   | ?                 | ?    | N    | 3.19 (74af752e4895) | no                  | no                   |
| Atom Valleyview                         | 6      | 55      | ?       | ?                   | ?                 | ?    | N    | 3.13 (ed93b71492d)  | no                  | no                   |

Figure 20: Table of interfaces that the Intel platform supports for energy and power [47].

## 2.8 Interacting with these interfaces from user space

Each of the interfaces, have their own methodologies and approaches for capturing information. For example, to make use of information in the PAPI or Perf interface simply requires including specific headers within your program. Other approaches such as the MSR or Sysfs interface require reading files within various directories in the GNU/Linux subsystem. In addition, the kernel itself offers various methods of providing such information to user space. This section explores some of the capabilities of these default directories. In particular, the Sysfs subsystem, Netlink interface and Procfs subsystem are described.

### 2.8.1 Sysfs

The Sysfs subsystem is a feature introduced in GNU/Linux 2.6 that allows kernel code to export information into user space through an in-memory file system [48]. The directory contains a series of symbolic links, files and directories that allow access to specific device information as well as kernel structures. In the context of PowerKap, many directories are included within the structure that grants useful energy information. This includes the Powercap interface, HWMON along with generic thermal info and ACPI information. In addition, ACPI information on modern kernels such as battery discharge have been moved from other file-system structures into the Sysfs.

In terms of information transfer, this file-system is useful for developers to gather information. This is because it provides a file-like interface which is easily accessible by most utilities. It is also architecturally portable and allows for event based signalling. This as discussed earlier and is advantageous to avoid busy polling.

### 2.8.2 NetLink

Another subsystem that offers ACPI information such as battery and thermal information includes the Linux Netlink System [49, 50]. This system is more advanced than the Sysfs system and relies on networking for communication between the kernel and user space. As such, it offers much of the same advantages for Sysfs such as event based information and is architecturally portable. It also allows transfer of large data, which may be necessary for future device based power information. The Sysfs system at present, does not allow the capability of transferring large data and is limited in size to the equivalent of one page.

### 2.8.3 Procfs

The Procfs [48] is a in-memory file system present on GNU/Linux systems. This system exists only in memory and provides a useful mechanism to gain further information relating to specific attributes for a process. Within the structure, there are plenty of useful interfaces that are useful for program profiling. For example, within the file in “/proc/<pid>/io” it is possible to get various useful information with respect to disk usage. This includes statistics such as the number of read, write system calls as well as the raw amount of data read and written to storage. There is also information about how much data was lost to errors.

Other useful information present in the process folder includes network specific data information. For example, in “/proc/<pid>/net/dev” there is useful data relating to how many bytes have been received and sent from network interfaces. This information is from

the perspective of the process, but relates to the total bytes consumed for the system. Such information can be useful for capturing network usage but require further modifications to the execution of the program. Namely, the program would be required to use a virtualised network interface specific for that program. The directory also considers useful information with respect to program profiling such as the time spent in idle and the number of voluntary and involuntary context switches.

The main limitation of this subsystem is the volatility of the data present in this structure. This is because such information is in-memory directly from the kernel. As such, when a process is killed or terminated, the entire file structure can disappear leading to corruption. Worse would be if the Linux Kernel re-assigns that process id to another process which can lead to wrong results being gathered. Other issues concerning this subsystem includes the fact that this interface does not support event-based signalling nor large data structures.

## 3 Profiling a program

### 3.1 Initial Ideas

Initially, the project aimed to replicate the functionality of the Energy Consumption tool for Visual Studio. This approach would have been ideal for usability as a developer could profile their entire tool from within a single program without additional configuration. This approach works for Microsoft as they control the entire process of profiling from the compiler, Operating System, IDE and power capturing tools. For example, the tool could deliberately reduce the power consumption of the IDE by preventing various dynamic analysis processes from occurring. This uniformity in control, enables easy and energy efficient interactions across these various mechanisms.

Such an approach does not work for GNU/Linux mainly due to the lack of conformity across the various components. The approach used for PowerKap was to build a profiling mechanism external to the IDE. This approach has numerous advantages. Namely, this enabled the profiling mechanism to capture various energy statistics without conflicting with various security features present within the IDE. For example, with IntelliJ IDE, the plugin file access is limited in access to files within the current project directory. This is also advantageous, as it enables PowerKap to be extended to alternative IDEs such as KDevelop or Eclipse. It also allows the possibility of profiling code without worrying about the IDE necessarily influencing the results. This in turn would improve the accuracy of the results.

PowerKap is currently designed with two main components. The first being a profiler and second being a mechanism to integrate the results back to the developer.

### 3.2 The Profiler

#### 3.2.1 Design Ideas

There are currently a few methodologies and approaches to be considered from other profiler designs. The first approach considered was to follow a similar approach to TREPAN with AEON. In this case, TREPAN works as an app that continuously probes the various Qualcomm interfaces present on Android devices. This profiler works independent of any running program and runs continuously in the background. Developers are able to make use of this profiler by adding annotation within the application that facilitates interaction with TREPAN. AEON utilises this approach by communicating with TREPAN over adb (Android debug bridge) and the network. Such an approach can be replicated on Linux by using various forms of IPC (Inter-process communication), such as UNIX sockets, memory mapped files and named pipes.

For PowerKap, such an approach would have some advantages namely the fact that the application can control the profiler to enable more granular energy information. For example, the executing program can notify the profiler in order to gather more frequent or accurate measurements. The main limitations of this approach mainly concern program timing and overheads in IPC. This is because, using IPC would introduce various timing delays caused by reading, writing and handling the various IPC mechanisms. It would also require additional infrastructure such as buffers to handle situations such as too much data being produced. This approach is favourable for the AEON profiler as the host computer



can serve as a consumer for this data without affecting the measurements. As such, the Android device would not have to handle the overheads of consuming and processing the data. Another limitation of this program would be the fact that many current GNU/Linux IPC mechanisms are currently not supported across various high-level languages. This is for various security and portability reasons. As such, IPC mechanisms are not uniform across different Operating Systems. This can be difficult for development when the code that is being profiled is designed for multiple architectures and platforms.

Another potential approach is to design energy models which can further be used to estimate the energy consumption of a program. This approach is particularly useful for asynchronous computing. This strategy works by stress testing a computer under a benchmark like SPEC-2006 and creating a model which equates energy consumed with things such as bytes written to disk. For the CPU, it can also be used to identify the energy consumption of certain code blocks. This approach is generally accepted and is used in academic projects such as e-Prof. However, the main limitation of this approach for this design is the fact that this approach requires calibration and that additional equipment is necessary to verify the accuracy of the models. This approach also requires a lot more intensive calibration towards the specific language in order to be useful. For example, in the case of Java, it may be necessary to develop a model based on Java Byte Code in order to get relevant useful information about the original source code.

### **3.2.2 Chosen Design**

The approach chosen for PowerKap was to design a framework that explicitly controls the execution of the target program. In doing so, PowerKap gains additional benefits such as greater timing control for the execution of the program. In addition, PowerKap eliminates the limitations of IPC mainly by maintaining a separation between the executing program and the profiler. Instead, PowerKap makes use of standard files to communicate. This approach allows results to be stored for further analysis, and ensures any annotations remain compatible across other platforms such as Windows. This approach in turn loses granularity in control for profiling during execution.

PowerKap works by sampling synchronous counters throughout the execution of the user program at regular intervals. The user program in turn annotates the code by inserting specific profiling code to highlight positions of interest. These annotations mark points of interest by printing timestamps at execution. The profiler repeats multiple executions of the program in order to further reinforce the gathered measurements.

This simple approach to profiling facilitates multiple capabilities that are advantageous for profiling. For example, by explicitly controlling the execution of a program, it is possible to use more volatile interfaces such as the Procfs energy interface listed previously. This is because the parent process can be explicitly aware of the times in which the child process is active. The simple filesystem approach is also compatible across languages and Operating Systems. It also requires minimal calibration and testing for the user.

In Figure 21, one can see a simple overall architecture of PowerKap. The main idea is to abstract away the profiler aspects such as the timing, program execution, printer from the sensors and energy data.

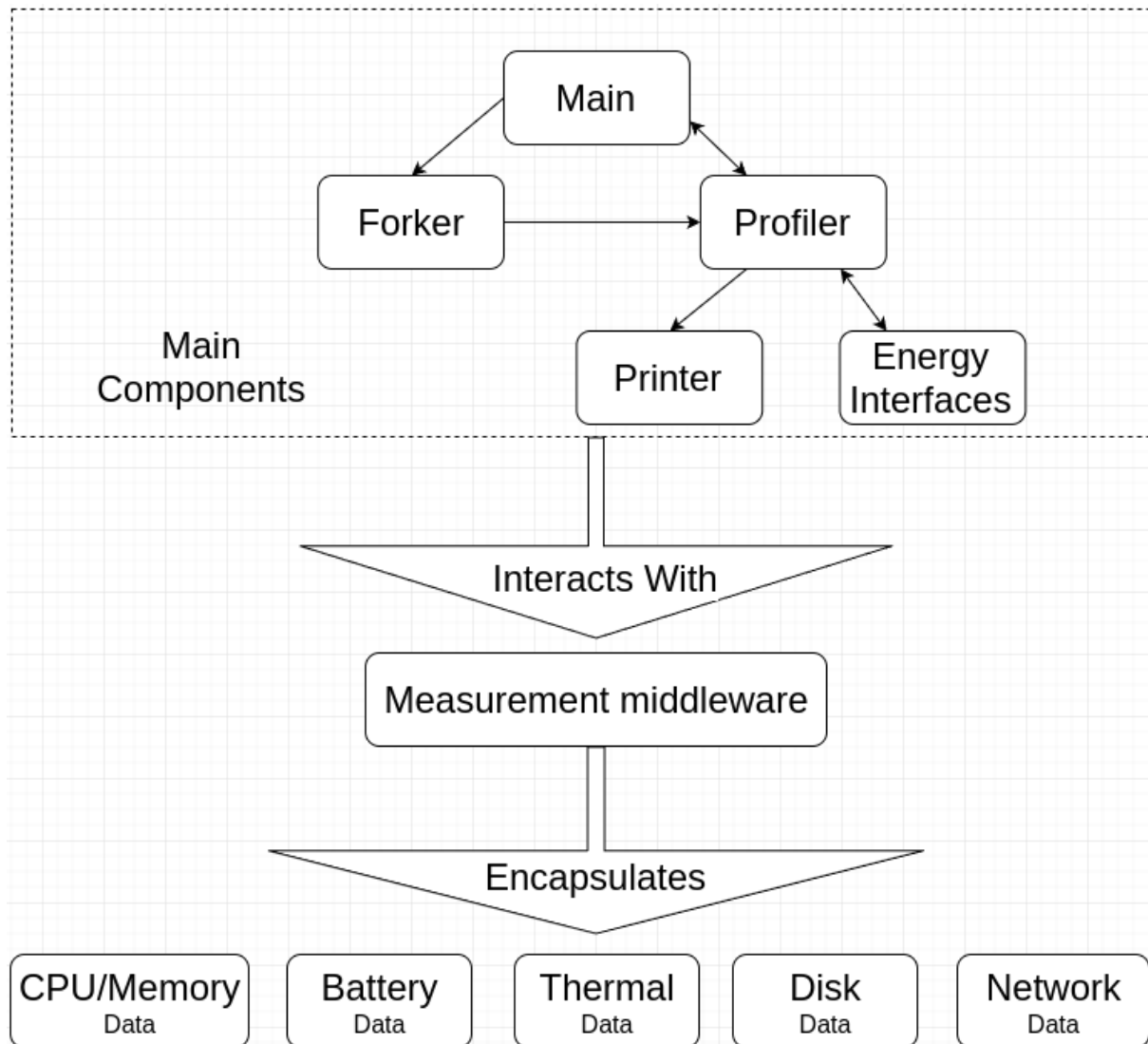


Figure 21: A simple overview of PowerKap.

### 3.2.3 Why not use current profilers?

As discussed previously, there are currently various tools and designs at present that capture various power and energy information across a variety of energy interfaces. There are a few reasons for not using Powertop, Turbostat or other middleware such as PowerAPI. Such APIs were not designed specifically for profiling a program. In this case, it is imperative to be aware of the mechanism and how data is being captured. This is because the API may not necessarily be the most energy efficient or uniform approach. The other reason why PowerKap does not extend from other tools is mainly due to a lack of documentation and easy build mechanisms. This is certainly the case for projects such as Powertop which cannot be built purely from the source repository without additional configuration.

### 3.2.4 Choice of energy interface

One of the main aims of this project is to explore the extent to which the current programs can gather energy estimations without particular user configuration or installation. The reason for this is because in most cases, it can be unrealistic to expect the user to download or configure specific modules just to enable profiling. The program is also designed to explore the limits of what is available from user space alone. This approach was designed mainly because developers are unlikely to want to profile untested and buggy code with root privileges. This is particularly relevant for GUI applications with configuration in the home directory. By running root, the user could potentially brick their home directory due to changing file permissions [51].

These restrictions can cause a lot of limitations with respect to the data that can be gathered from the profiler. For PowerKap, the aim is to capture as much information relating to energy consumption. For this reason, the project currently captures information on the following sub-categories.

1. Network Bytes
2. Disk Bytes
3. Temperature data
4. CPU energy statistics
5. Battery statistics

There were chosen for a few reasons. Namely, this project would replicate the functionality provided by the Microsoft Visual Studio Energy Tool. Another reason is that there is a particular limit with the number of interfaces and data that can be captured during each interval. This is to reduce the energy costs for gathering and processing this data. Similarly, too many samples could result in significant misalignment in sampling.

#### 3.2.4.1 Energy Consumption of CPU and Memory

The first and probably most useful information gathered by PowerKap is the energy information presented via the Powercap interface in the Sysfs directory. This information is useful mainly because in this case the user can capture accurately the various energy metrics relating to the Core, Uncore and DRAM controller present from RAPL.

Unfortunately, these statistics are specific to the package as whole rather than per thread. In order to gain more granular thread specific information, it may be useful to use a different metric like CPU utilisation.

### **3.2.5 Thermal Information**

In addition to the above, and in keeping with the issues of thermal energy listed previously, PowerKap aims to capture various thermal data. At present this is simply sensors registered in the Linux kernel and present in the thermal zone interface. This was chosen over the HWMON interface mainly because it is generic and standardised across kernels.

#### **3.2.5.1 Battery Information**

Most laptops in addition come with a built-in power meter. This is through the battery sensor which grant useful data with respect to current and voltage draw. This sensor is independent of the system and is located in the Sysfs system. With this data, it is possible to corroborate some of the energy trends so long as the rest of the device systems are either off or retain uniform energy consumption.

#### **3.2.5.2 DiskIO**

As explored in the background section, DiskIO can have a drastic effect on energy consumption. For profiling, PowerKap works by gathering IO stats present in the executed child process. This approach has some limitations mainly due to the fact that it is limited purely to the child process. The reason PowerKap restricts this behaviour is because the child process is the only PID that the profiler has control over. For this reason, if a process decides to spawn more children explicitly for the purpose of writing, the program will not be able to capture this IO data. Upon completing, the profiler does not have any notification of the process being stopped. As such, these children could theoretically be assigned to another process which could lead to inaccurate disk information.

An alternate approach to handling this would be to use a dynamic linked library to capture read and write information. This would work by utilising a technique LD\_PRELOAD to create a custom function to intercept calls. This approach can accurately capture at the point of the systemcall which can be useful for recording the point at which the disk is being accessed. The way this mechanism works is that the library chosen would be called before the corresponding read system call. The main limitation of this approach is that it doesn't account for the situation when data is cached and does not call from disk. This is in contrast to the IO file in the Procfs system which accurately accounts for the actual bytes written and read from disk. Another limitation of this approach is the fact that this approach would require additional inter-process communication techniques to capture the various information recorded from the system call. This is because the library would be loaded with the process and would therefore not have a mechanism to communicate with the profiler. Unfortunately, this technique would also result in an overhead due to the fact that the function would be slower than performing a native system call.

At present, the profiler only captures the point in which data is actually read and written from disk. Unfortunately, it is not currently possible to capture the actual energy

information from the disk. This is because, currently there are no energy or power counters exposed for these devices.

### 3.2.5.3 Network

Another important aspect with respect to energy consumption is networking. This can have significant effect on energy consumption due to the asynchronous nature of this system. As with the IO system, network adapters currently do not expose power or energy information. As such, the design of the module in PowerKap is similar to that used for disks. In this case, the aim was simply to capture synchronous data corresponding to the process such as the amount of bytes sent and received from the process.

This data is not particularly simple to capture as the kernel does not distinguish between the received and sent bytes for a specific process. Instead the kernel provides statistics of the bytes transmitted simply per interface. For example, it accounts for specific data from the Ethernet or wireless adapter. For this reason, it was particularly challenging to capture the data corresponding to a single process. At present, there are a few approaches attempted each with their own advantages and disadvantages.

The first approach considered was to try and capture data by using a technique called “strace”. This approach works by intercepting the process whenever it calls a relevant system call. This technique is similar to that used of LD\_PRELOAD. To capture specifically network information, this approach would need to be paired with “-e trace=network”. Unfortunately, this approach comes with disadvantages namely in terms of speed. In a worst-case scenario, this approach could lead to a 442x slower performance than without strace [52]. This kind of overhead is not acceptable when calibrating a function for estimating the average energy consumption.

Another approach that was considered was the use third party external module such as Nethogs [53]. This interface makes use of a various other libraries such as libpcap and ncurses to capture the network consumption of specific processes. It works by scanning the /proc/net/tcp file for established TCP connections. Upon reading the file, the local and remote socket addresses along with inode numbers are tracked. Nethogs subsequently uses the libpcap utility to sniff traffic and associate the data with /proc/net/tcp. In order to make use of this utility, there are many limitations at present. The first being user configuration; in order to make use of the program, users have to assign permission to the library using the “setcap” command and root privileges. In addition, the program does not offer a stable library to interface with. At present, the library needs to be compiled from source, is undocumented and is unlikely to work across distributions. The approach also requires significant processing relative to other methods. For this reason, this approach was also dismissed.

The final technique that was considered was to make use of the interface capturing capabilities present in Procfs. This would be possible by creating a virtual adapter which is only usable by the target process. This is currently an approach which is unique to PowerKap and was achieved without root privileges. To do so, it was possible by making use of user-namespaces. This is a sandboxing technique introduced by kernel 3.8 and completed in 3.9. At present it is used by few applications including Google Chrome, Firejail

(sandboxing software) and various container technology such as Docker. This approach has many advantages relative to the other approaches, namely a reduction in overheads. For example, to ping from the virtual interface to the Ethernet adapter only introduces a speed overhead of 0.05ms on Linux kernel 4.11. It also provides many other benefits with respect to security and isolation, which is particularly useful when profiling unknown or potentially dangerous code.

This feature requires the use of root privileges. The exception to this is if the namespace is created using clone or using the setns function. Both require the executing binary to have capabilities such as the CAP\_SYS\_ADMIN and CAP\_NET\_ADMIN. Fortunately, upon setting the capability, the rest of the application can remain in user space. In this case, the executing process is isolated with the same executing privileges as the calling process.

#### **3.2.5.4 Choice of language**

The energy profiler for PowerKap is designed in C++. This is due to various reasons, the first is to enable extensibility in PowerKap. This is because a lot of kernel and user interfaces such as PAPI and Perf all are written to be used in C. For this reason, PowerKap needs to be capable of reusing current code and interfaces to extend features and sensors. The language C++ was chosen simply to enable this compatibility with such external interfaces.

This project uses C++ over C is to take advantage of the Objected Oriented features of language. This feature is useful for encapsulating and processing and manipulation of data behind class structures. Therefore, the project is able to maintain a clear separation between the various mechanisms needed to read and process the various data elements. In the future, this property would be useful when incorporating other interfaces such as the msr module which requires register specific knowledge. The C++ STL also grants useful features such as string manipulation, fast dynamic data structures and file manipulation utilities. These are useful for manipulating the data gathered from the various energy interfaces which are not particularly clean or consistent.

#### **3.2.6 Profiler Design**

The profiler is the component that captures the energy information of a program. It is designed to minimise the amount of energy consumed by the profiler itself. It can be broken up into a series of sub-components that are listed as follows. These can be seen in Figure 22.

##### **3.2.6.1 Forker**

The forker module is a program that is dedicated towards controlling the profiler and execution of user-defined programs. This module is particularly complex as it makes use of a variety of low-level kernel techniques to ensure minimal overhead, explicit timing control and to reduce risks of zombie child processes. The main method of achieving these goals is through the use of the clone systemcall. This function call is more beneficial in this case than fork or vfork. This is because this call allows PowerKap to control various aspects of the fork such as being able to spawn the child fork within its own namespace. In this case, PowerKap makes use of the following flags:

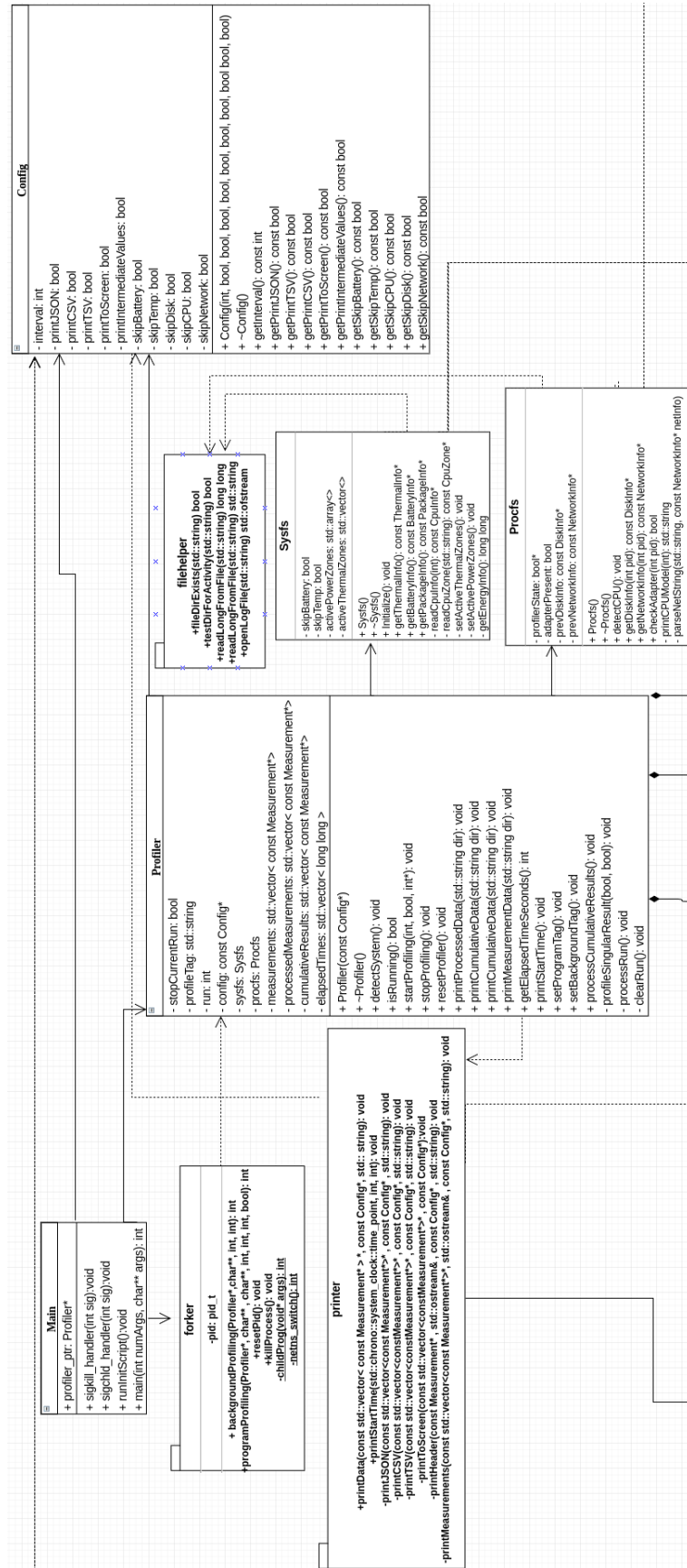


Figure 22: A UML diagram of the main components for PowerKap.

The first flag PowerKap makes use of is `CLONE_VFORK`. This flag is an optimisation that allows the pausing of the parent process until the child process either calls `exit` or `execve`. This approach is useful as it allows the profiler to ignore setup costs for creating the fork and setting up the namespaces. As a result, PowerKap will only profile the portions of code corresponding to the program.

PowerKap also makes use of the `SIGCHLD` flag which is particularly useful. The main advantage of this flag is that upon the child process being killed, the signal `SIGCHLD` is propagated to PowerKap. This is useful as PowerKap has specific signal handlers that can capture this signal and stop capturing data from volatile interfaces.

The final set of flags used by PowerKap mainly concern those in setting up a custom namespace. These are the flags `CLONE_NEWPID` and `CLONE_NEWNS`. The flags in this case are particularly useful for setting up an isolated program with its own network space.

### 3.2.6.2 Profiler

The actual profiler module is designed to act as a coordinator for accessing the different energy interfaces and printing the results. It is designed in an energy efficient manner. For example, during initialisation, the program begins by checking various thermal, energy and battery interfaces to ensure that the system minimises the number of IO calls. This is important for gaining accurate readings and to ensure a minimum overhead.

The actual profiling stage takes 3 steps which are again designed to minimise energy impact. The first stage while profiling is simply to read raw measurements and store the result. Once this stage is complete and the profiler has stopped running, the program subsequently calls the printer before processing the raw results into a useful form. This step is important as there are cases where it is necessary to correct various sensor measurements. An example includes AMD's thermal sensors included in modern Ryzen CPUs. In this case, there is a systematic error present that means that readings are consistently off [54]. This error is present to ensure the fans turn on. For PowerKap, this stage is designed to measure derivative measurements from the raw results. Upon completing this stage, the profiler prints the results and aggregates all the results from the run into a cumulative measurement. This stage is important as otherwise the density of gathered results could be too large especially for long running programs. The cumulative results stage is designed to gather important information such as running averages for the result along with maximum, minimum and standard deviation values.

### 3.2.6.3 Sysfs, Procfs and Energy Interfaces

One of the key points of the profiler is to be able to capture data from potentially multiple interfaces with as little user configuration as possible. For this reason, the program makes use of the Sysfs interface. This was chosen because most of the modules such as Powercap, thermal and battery and provided by default in the GNU/LINUX kernel. The information captured from these various interfaces and encapsulated within their own class.

PowerKap also makes use of volatile memory structures such as Procfs to gather process and system information. This interface is more complex than the Sysfs structure mainly due to



```

void killProcess(Profiler* profiler)
{
    if (::pid > 0)
    {
        //Let the process terminate cleanly first
        kill (::pid, SIGTERM);
        std::this_thread::sleep_for(std::chrono::milliseconds(20000));
        int status;
        pid_t pid;
        if ((pid = waitpid(-1, &status, WNOHANG)) <= 0)
        {
            if (pid == 0)
            {
                //Send SIGKILL after 20 seconds
                kill (::pid, SIGKILL);
                while(waitpid(-1, &status, WNOHANG) > 0)
                {
                    //Wait for the process to be killed
                    //(SIGKILL cannot be ignored)
                }
            } else if (pid == -1)
            {
                // "Error killing child process, clearing up and exiting"
                if (profiler)
                {
                    profiler->resetProfiler();
                }
                exit(EXIT_FAILURE);
            }
        }
        std::cout << "Process terminated cleanly" << std::endl;
        ::pid = -1;
    }
}

```

**Figure 23:** This figure shows the main mechanism for handling the killing of zombie processes. Particular care needs to be taken to ensure that child processes actually terminate.

```

char stack[4096];
struct ::cloneArgs childArgs;
childArgs.args = parmList;
childArgs.envp = envp;

int pid = clone(childProg, stack + 2048,
                CLONE_NEWPID | CLONE_NEWNS | CLONE_VFORK | SIGCHLD,
                &childArgs) ;

```

**Figure 24: An example of the clone function used in Forker. In this case, the stack needs to be setup. The arguments and environment variables are passed as the last argument to the function. These are subsequently used as the arguments to the execve call in childProg.**

the volatility in reading the contents of this file system. To help handle this volatility, this interface is designed in a different fashion to that of the Sysfs interface. In this case, the process works by gathering the data from the file line by line. It also keeps track of the last successful measurements in case of temporary file failures. The directory is also designed in a particularly error sensitive fashion to avoid creating new measurements unless it comes from the correct process id. This can be observed in Figure 25.

#### 3.2.6.4 Printer

The final profiling stage for PowerKap is to print the results generated into usable formats. So far, PowerKap supports printing various formats including tab separated values(TSV), comma separated values(CSV) and JavaScript Object Notation(JSON). This approach was chosen to enable users to display the results in alternative programs if required. In order to support JSON, PowerKap uses Nlohmann’s JSON serializer for C++ [55].

#### 3.2.6.5 Measurement and Energy Structure

In addition to the energy interfaces, the program is designed such that the internal functionality of each sensor and measurements are separated. The above main modules in Figure 22 are able to interact with the data using a series middleware of classes. This layer can be seen in Figure 26.

The classes listed in Figure 26 reflect the three stages of PowerKap. Each encapsulate all the behaviour and methods necessary for each stage and provide an easy interface to interact with the sensor data in Figure 27.

In Figure 27, one can see the various data sensors captured by PowerKap. The choice in capturing battery, CPU, thermal, network and disk attributes was chosen specifically because they match the functionality provided by Visual Studio. In the future, it would be possible to extend this functionality to additional interfaces such as graphics power consumption and peripheral energy consumption.

```

const NetworkInfo* Procfs::getNetworkInfo(int pid)
{
    const NetworkInfo* result = NULL;
    if (adapterPresent)
    {
        std::string dir = "/proc/" + std::to_string(pid);
        dir += "/net/dev";
        std::string line;
        std::ifstream infile(dir);
        if (!infile.fail())
        {
            while(std::getline(infile, line))
            {
                {
                    result = this->parseNetString(line);
                    if (result != NULL)
                    {
                        break;
                    }
                }
                infile.close();
            }
            if (result != NULL)
            {
                this->prevNetworkInfo = result;
            } else {
                if (profilerState && !(*profilerState))
                {
                    if (detectNetworkInterface(pid, "veth-a"))
                    {
                        if (prevNetworkInfo != NULL)
                        {
                            result = this->prevNetworkInfo->clone();
                        }
                    } else {
                        adapterPresent = false;
                        // Error no virtual namespace present.
                        // Skipping Analysis.
                    }
                }
            }
        }
    }
    return result;
}

```

**Figure 25: The function designed to read the Procfs file network interface.**

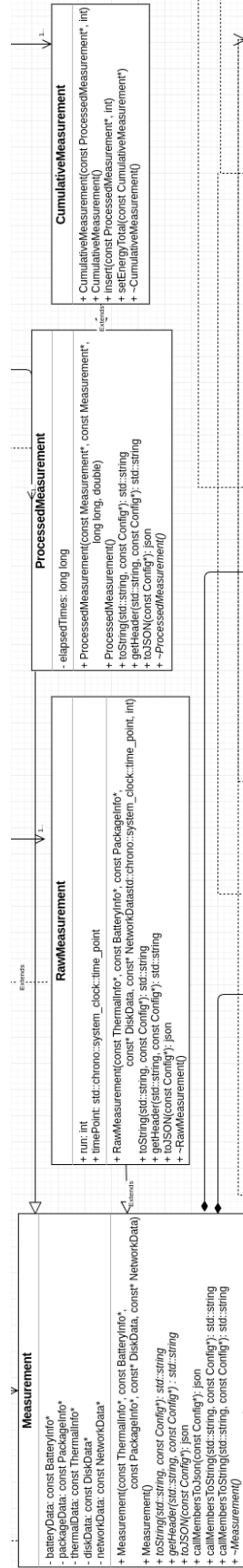


Figure 26: A UML diagram of the measurement layer for PowerKap.

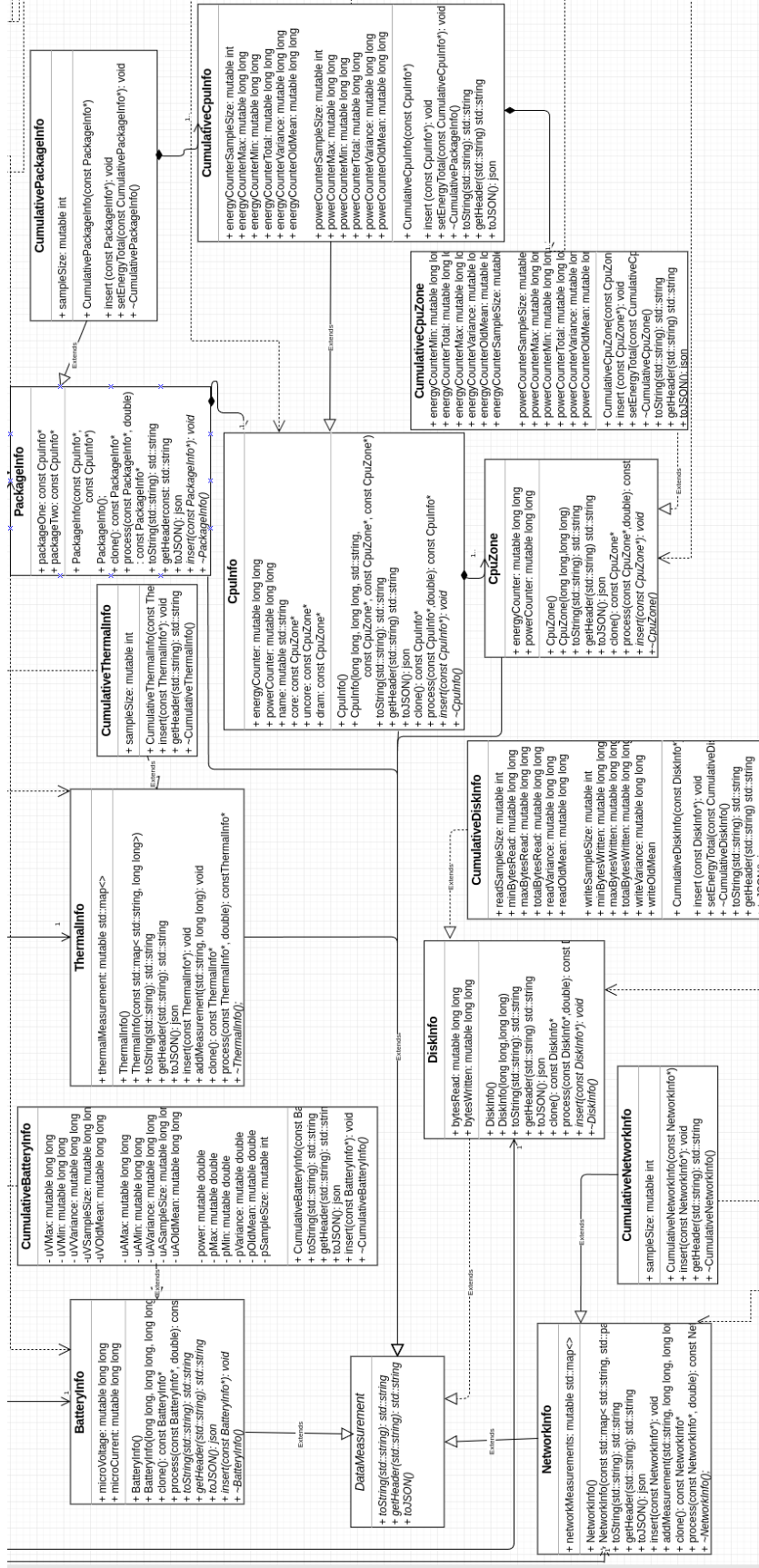


Figure 27: A UML diagram of the current sensor classes for PowerKap.

### 3.2.6.6 Networking Script

Many network applications require use of networking capabilities. Unfortunately, the default approach used with the `setns` function is to create a default namespace without any networking adapters. The normal approach used by Chrome is to assign a specific adapter to this network namespace. For PowerKap, a custom virtual adapter was needed so that it could be tracked in the `Procfs` system. To do so, a bash script was used to generate the necessary configuration. This script can be observed in Figure 28. This script is particularly useful as it is designed entirely to use the `ip` command rather than `ifconfig` and the like. As such, it is compatible with new Linux systems which do not come with the legacy deprecated network tools. It also forwards traffic between the interfaces using `iptables`. This is a particularly fast kernel firewall. The reason, PowerKap uses this shell script rather than within the program using system is mainly so that users can configure various aspects such as the `ip` to avoid network conflicts. All the commands executed in this script are designed to be temporary and will reset on reboot.

### 3.2.7 Implementation Details

This section is designed to serve as a documentation on some of the design decisions faced when constructing the profiler.

#### 3.2.7.1 Steps taken to minimise the overhead introduced by the profiler

As discussed earlier, with any form of in-band energy technique the accuracy of the measurement can be influenced by the profiler itself. PowerKap aims to minimise this impact as much as possible. This has influenced the design in many ways. The first being the separation of the integrated developer environment and the profiler. This is unlike other tools such as the visual studio profiler which is integrated into the IDE, these tools introduce IDE as an overhead. This would reduce the accuracy of the results generated.

The three stage pipeline used for PowerKap is also primarily designed to reduce the power consumption of PowerKap. In this case, processing the results only happens once gathering the raw data is complete therefore further analysis and processing does not influence the energy consumption of the results. In addition, the final stage of storing the results in averages enables minimal data to be stored for each run. This reduces the overall memory consumption used by PowerKap.

Other energy efficient techniques used are inspired by the green software techniques described in the background section. This mainly involves techniques such as only using interfaces that are usable and giving the option for users to pick and choose which metrics to gather. Both of these enable energy efficient approaches by avoiding IO wherever possible.

Another aspect that influenced the design of PowerKap is the type of data and the capturing mechanisms. The idea of the project was to avoid storing as much information unless relevant for energy and to ensure a consistent mechanism of capturing the data. For example, at present most of the techniques rely on file type directories present in the `Sysfs` and `Procfs` systems. All the data captured are related purely towards synchronous energy information and most of the data gathered is captured and processed within the kernel rather than user space. This approach minimises energy usage by avoiding the overheads of capturing

```

#!/bin/bash

echo "Welcome to my script to create a test interface:"
echo "First please pick a number corresponding to the interface you wish
to bind."
ip link show |grep UP

if [ "$EUID" -ne 0 ]
    then echo "Please run as root"
    exit
fi

read -n 1 -p "Input Selection:" interface
input="{interface}:"
stringInterface="$(ip link show |grep $input)"
cutVal=${stringInterface#*:}
chosenInterface=${cutVal%%:*}
sysctl -w net.ipv4.ip_forward=1
ip netns add test_ns
ip link add veth-a type veth peer name veth-b
ip link set veth-a netns test_ns
ip netns exec test_ns ip addr add 192.168.163.1/24 \
    broadcast 192.168.163.255 dev veth-a
ip netns exec test_ns ip link set dev veth-a up
ip netns exec test_ns ip link set dev lo up
ip addr add 192.168.163.254/24 broadcast 192.168.163.255 dev veth-b
ip link set dev veth-b up
ip netns exec test_ns ip route add default via 192.168.163.254 dev veth-a
localIP="$(hostname -I)"
getIP=${localIP%% *}
echo $chosenInterface
echo ${getIP}
iptables -t nat -A POSTROUTING -s 192.168.163.0/24 \
    -o $chosenInterface -j SNAT --to-source $getIP
iptables -A FORWARD -i $chosenInterface -o veth-b -j ACCEPT
iptables -A FORWARD -o $chosenInterface -i veth-b -j ACCEPT

```

**Figure 28: Network Virtual Adapter script**

$$\begin{aligned}
Mean_1 &= x_1 \\
S_1 &= 0 \\
Mean_k &= Mean_{k-1} + (x_k Mean_{k-1})/k \\
S_k &= S_{k-1} + (x_k Mean_{k-1}) * (x_k Mean_k) \\
Var &= S_K/k - 1 \\
SD &= \sqrt{Var}
\end{aligned}$$

**Figure 29: Welford’s algorithm for computing variance.**

through multiple techniques. For example, by using a technique such as the asynchronous Netlink interface, it could be difficult to retain a consistent interval in timing. Similarly, whilst other information present in the Procfs directory can be useful such as specific CPU utilisation and page swapping information, these statistics would greatly increase the size of each measurement which can increase the memory consumption overhead for PowerKap. Instead, current profiling tools such as the Perf system would be more useful for optimising the program in this regard.

### 3.2.7.2 Avoiding the impact of the user environment

One of the main issues with the problem of energy transparency is the influence of external factors such as the Operating System, background tasks, scheduler and even hardware influences. For example, for a hard drive that is extremely fragmented, extra energy could be spent waiting for the disk to gain results. Subsequent data could equally require less energy due to caching in memory. Alternatively, at extremely low power, the GNU/Linux system may itself prioritise other tasks or deliberately enable energy efficient modes with the hardware. These kinds of external influences can be difficult to control and can affect the accuracy of the results.

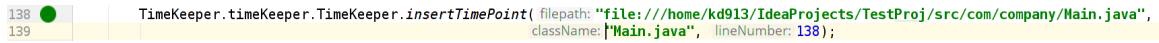
PowerKap tries to gain meaningful results that are useful for the developer. For this reason, the program calculates information such as standard deviation so that developers can judge the accuracy of the results presented. To calculate the Standard Deviation for PowerKap, Welford’s Algorithm [56] is used. This algorithm was developed by B.P Welford in 1962 and provides a mechanism of computing sample variance in a single pass. It achieves this without storing sample data points. This property is useful for reducing the overhead of PowerKap and to enable greater reproducibility across runs. The algorithm can be seen in Figure 29 and is useful for computing variance without risking numerical overflow. Such a situation would be a problem if using the standard sample variance formula listed as follows.

$$S_k = \frac{\sum_{k=1}^{\infty} (x - \bar{x})^2}{k - 1}$$

## 3.3 Linux Java Energy Assessment (LJEA) plugin

Along with the profiler, there is a second portion to the project which is designed to integrate the information gathered from PowerKap into the integrated development environment.



The image shows a snippet of Java code from an IDE. Line 138 contains the method call `TimeKeeper.timeKeeper.insertTimePoint`. A green dot is placed on the line number 138 in the left margin, indicating an energy point. The code continues on line 139 with `(filepath: "file:///home/kd913/IdeaProjects/TestProj/src/com/company/Main.java",`  
`className: "Main.java", lineNumber: 138);`

**Figure 30:** An example energy point that is automatically generated for the user.

This part of the module also has an important aspect of enabling the user to annotate the code for additional details whilst profiling elements. It also provides further granularity in the information gathered. This part of the project can also be split into various components.

### 3.3.1 Choice of IDE

IntelliJ was chosen the chosen platform for displaying the results. The predominant reason being that it is open source and has an open plugin API. The IDE is a commonly used development platform that is used extensively for multiple platforms. It has been extended by Google to form Android Studio. Due to this extensibility and wide support for languages, it is a good platform to develop LJEA. In addition, the platform was chosen due to the addition of a “power saving mode” [57]. With this mode, IntelliJ disables energy intensive activities such as error highlighting, on-the-fly inspections, autopopup code completion, and automatic incremental background compilation. This could be useful in the future if the tool could integrate directly within the IDE. Other benefits provided by IntelliJ includes extensions and custom language extensions. These features would be useful for implementing specific contracts for resource consumption.

At present, PowerKap only supports Java on IntelliJ. However, in the future, other IDEs and other languages can be extended to support LJEA. Java was chosen as an initial language to support profiling mainly because IntelliJ provides native support for Java. The project could be extended to CLion or to other IntelliJ languages

### 3.3.2 EnergyPoints

The first key aspect of this module with respect to power profiling is the capability of insertion specific points in the program to highlight energy use. This step is designed to highlight points of interest and to provide file location information. This can be seen in Figure 30. In this case, when the code is executed, a timestamp is recorded along with the file and line location. This allows the user to easily navigate to points of interest.

Before program termination, the user needs to specify points to print these generated data. This is so that the user can choose the best point for performing the IO necessary to profile the program. It is also because only the user the entry and exit point for a program.

#### 3.3.2.1 The profiling code

Upon annotating the code, the user can automatically generate a package and class which are specifically designed to handle these timepoints. This code is generated in a separate package at the root of the project to avoid conflicting with the current code base. This code can be seen in Figure 31.

```

public class TimeKeeper {
    private static List<EnergyAnnotation> annotations = Collections.synchronizedList(new ArrayList());
    private static int callIndex = 0;

    public static void insertTimePoint(String filepath, String className, int lineNumber) {
        callIndex++;
        annotations.add(new EnergyAnnotation(new Date(), filepath, className, lineNumber, callIndex));
    }

    public static void printTimePoints() {
        try {
            Writer w = new FileWriter("elapsetimes.txt", true);
            if (!annotations.isEmpty()) {
                for (EnergyAnnotation annotation : annotations) {
                    w.write(annotation.toString());
                }
                w.close();
                annotations.clear();
                Thread.sleep(1000);
            }
        } catch (IOException | InterruptedException e) {
            e.printStackTrace();
        }
    }

    static class EnergyAnnotation {
        Date date;
        String className;
        int lineNumber, callIndex;
        String filePath;

        EnergyAnnotation(Date date, String filePath, String className, int lineNumber, int callIndex) {
            this.date = date;
            this.className = className;
            this.lineNumber = lineNumber;
            this.callIndex = callIndex;
            this.filePath = filePath;
        }

        public String toString() {
            SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy:MM:dd HH:mm:ss.SSS");
            return callIndex + ", " + dateFormat.format(date) + ", " + filePath + ", " + className +
                ", " + lineNumber + "\n";
        }
    }
}

```

**Figure 31:** The automatically generated energy time code. It contains various information such as the file location, class name and line number.

### 3.3.2.2 StackTrace

Upon executing the program and interpreting the JSON results from PowerKap, the user is presented a stacktrace corresponding to the points in which the generated program has executed the timepoints.

Figure 32 shows an example stacktrace of a program. The mechanism works by comparing the executed time with a recorded start time marked by PowerKap. Based on averaging these elapsed time points, the program looks up the energy data produced from the profiler to create an average energy total. Within each entry point, the user is presented information relating to how much energy the CPU and memory hierarchy consumed from the Powercap interface. In addition, the user is provided various data such as how much battery was consumed between the various entries in the list.

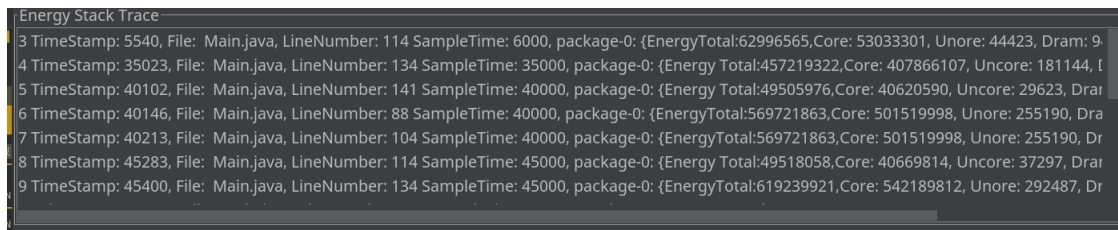


Figure 32: An example stacktrace.

The stacktrace box also provides features such as the capability of navigating the source code along with setting markers on energy graphs.

### 3.3.2.3 Energy Graphs

Along with the stacktrace, the user is presented a series of graphs that enable the user to visualise the energy consumed by the program over a period of time. This can be seen in Figure 33. Within each entry point, the user can also clearly see the error presented by the measurements. These are shown by the lighter graph surround the entry points which allows the user to determine the quality and accuracy of the results. In order to generate these graphs, the tool JFreeChart [58] is used. This tool offers many capabilities that were useful such as the capability of saving image plots. It also enabled auto-sizing of the plots and markers. The deviation capability is also built into the tool by using a deviation renderer.

### 3.3.3 Implementation Details

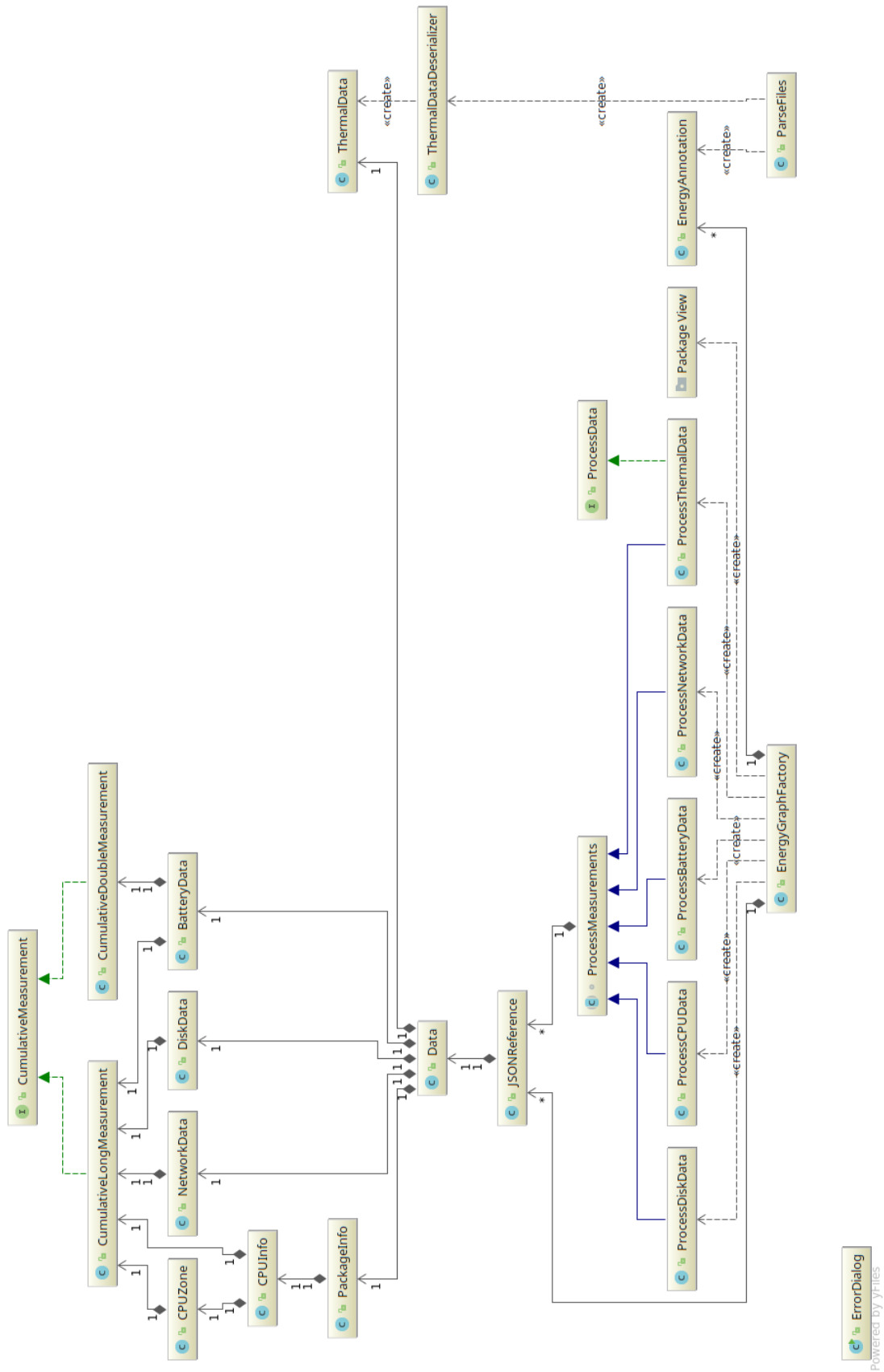
The process of building the graphical interface for LJEA was rather straightforward relative to building PowerKap. This is because there is a lot more documentation and tutorials which were useful for building the various graphical elements. The main design constraint for this part of the project was maintainability and usability.

#### 3.3.3.1 The UI design

Figure 34 shows the overall structure of the graphing module used for LJEA. The above mainly constitutes the model aspect of the plugin. In this case, we have the JSON structure generated from PowerKap, converted into Java class objects. These are all contained



Figure 33: The graphing module.

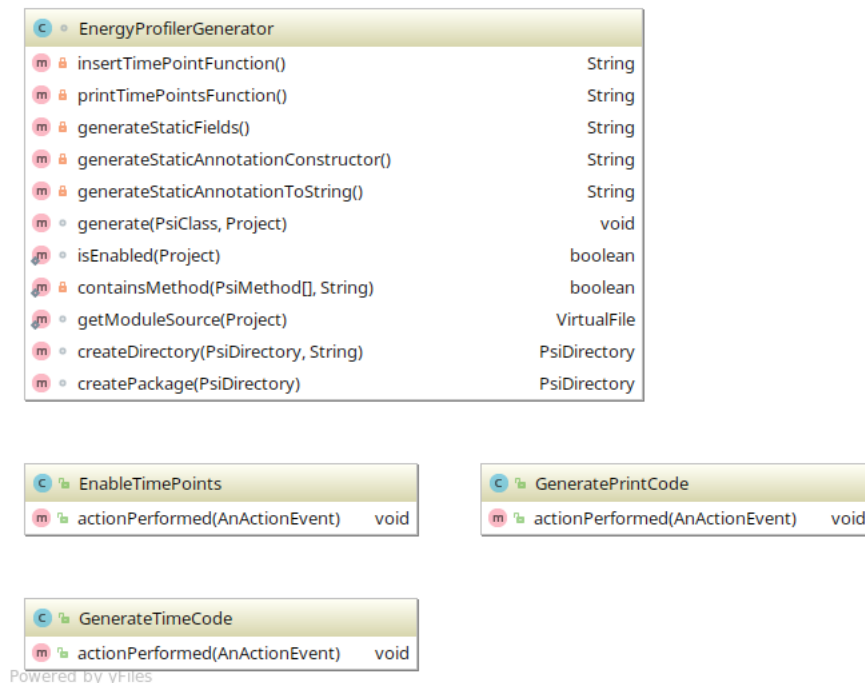


**Figure 34: A overview of the structure of the LJEA graphing module.**

behind JSONReference. This structure is useful for generating and processing data such as those used for the stacktrace. The structure is useful as calls can cascade down subclasses such as PackageInfo, Network Data, Disk Data and Battery Data. A custom deserializer was necessary to process the thermal data.

Along with this we have another layer which is designed to process this JSON generated data. In Figure 34, these are noted by classes with the “Process” prefix. These take in user options from the GUI and process the data into a usable format for JFreeChart. Finally, the package Energy Graph factory is designed in a standard Model View Controller structure. Within the package, there are two files, one being a form file and the other the controller class. These represent the view and controller respectively. The view was designed using Java Swing and is built with IntelliJ’s swing designer.

### 3.3.3.2 Action Classes



**Figure 35: A overview of the structure of the LJEa graphing module**

The action classes used in LJEa are designed simply to manipulate the GUI to enable the various annotations necessary for the project to work. This includes the energy annotation points and the various aspects necessary to generate the TimeKeeper class. This mainly involved manipulating the source and file system using virtual files and the PSIFile structure used within IntelliJ.

```

public void actionPerformed(final AnActionEvent event) {
    final Editor editor = event.getRequiredData(CommonDataKeys.EDITOR);
    final Project project = event.getRequiredData(CommonDataKeys.PROJECT);
    //Access document, caret, and selection
    final SelectionModel selectionModel = editor.getSelectionModel();

    //New instance of Runnable to make a replacement
    Runnable runnable = () -> {
        if (EnergyProfilerGenerator.isEnabled(project)) {
            final Document document = editor.getDocument();
            VirtualFile file = event.getData(PlatformDataKeys.VIRTUAL_FILE);
            String filePath = file != null ? file.getUrl() : "";
            String fileName = file != null ? file.getName() : "";
            int lineNumber = editor.getCaretModel().getLogicalPosition().line + 1;
            String replacementText = EnergyProfilerGenerator.className +
                "\n", " + "\n" + fileName + "\n", " + lineNumber + " );";
            int offsetStart = editor.getCaretModel().getOffset();

            document.insertString(offsetStart, replacementText );
        }
    };
    //Making the replacement
    WriteCommandAction.runWriteCommandAction(project, runnable);
    selectionModel.removeSelection();
}

```

**Figure 36:** An example of the method used to generate the timestamps.

## 4 Project Evaluation

In this section, the aim is to try and evaluate and research some of the capabilities of PowerKap.

From background research, there are currently many methods and approaches that can evaluate the accuracy and effectiveness of the profiler. In the ENTRA project for example, the chosen approach is to evaluate the effectiveness of the profiler against a set of known programs with known energy consumption. This is particularly useful for simple architectures and restricted environments. In the case of the original deliverable, the developers restricted and specialised the series of benchmarks for XMOS devices [59].

This approach does not scale well for larger devices such as laptops, desktops and the like. This is because of the large variability in hardware, software and background processes which can each contribute to variability in energy performance. The other limitation in this case is the ability to measure the accuracy of various measurements. This is because without laboratory equipment, it can be difficult to estimate the true accuracy in energy measurements for specific components.

### 4.1 The hardware and methodology

For the purpose of evaluating PowerKap, various hardware was used to measure the accuracy of the profiler across platforms. The first being a Dell XPS 13 9343 laptop running Kubuntu 17.04. This laptop was advantageous as it contains a Broadwell 5th generation processor. Specifically an Intel i7 5500U (a 2 physical, 2 virtual core processor). This processor is useful as it contains an integrated memory controller which enables the DRAM RAPL statistics. In addition, the laptop has 8gb of RAM, a 4k IPS touchscreen, Broadcom wireless chip and 512GB Samsung SSD. To avoid the influence of energy consumption of the various integrated devices within the laptop, various components such as the webcam, Bluetooth and USB controller were disabled. Similarly, for each test, various components were also disabled such as the WiFi, BlueTooth and screen brightness. The idea in this case was to isolate the environment to the component that is being stressed.

Along with this laptop, a custom desktop was also used as a comparison to see how the same tests perform unconstrained. This system is a high performance modern gaming desktop. This platform was useful as it can be used to demonstrate the energy consumption differences in a platform that is not energy or thermally constrained. In terms of hardware, this system runs with a 6gb Nvidia GTX 1060, Intel i5 7500 (a 4 physical core processor) and 16GB of 2133Mhz DDR4 memory. It also uses a 275GB Crucial MX300 SSD. For cooling, the system has 3 case fans and a Corsair H45 hydropump water cooler. Software side, this platform also provides an interesting test as it runs the same Kubuntu 17.04 instance as used in the laptop. The exception in this case is the use of a different energy governor. In this case it uses the Linux kernel CPU Frequency governor [60] as opposed to the default Intel Pstate governor. This governor is responsible for transitioning the CPU C and power states.

Another reference point used was a Dell XPS 15 9560 running with a Nvidia GTX 1050 and Intel i7 7700HQ (4 physical cores, 4 virtual cores). This platform as it shared many similarities with the Dell XPS 13, whilst providing a useful reference point against the





**Figure 37: The set of laptops used for testing. In this case, a Dell XPS 15 9560, a Dell XPS 13 9343 and two Toshiba Portege R830-13 laptops.**

gaming desktop. The platform in this case had 16GB of RAM and a 512GB SSD. When performing the tests on this machine, the test was run from a 128GB Sandisk Cruzer USB stick. The operating system in this case was running Ubuntu 16.04, which provides a useful comparison between the old 4.4 kernel and the new 4.10 kernels.

In addition to the laptop listed previously, an external power meter was used. This was a “Plugin electricity cost meter” from Maplin with a model number N67FU. This device has an accuracy of 1.5% for measuring power. To evaluate the power consumption for this, the results were recorded by video. This video was played back at a slower speed to capture the change in power every second. To ensure there is no background influence with the power meter, a kettle was plugged into adjacent plugs and it was confirmed that there were no power fluctuations in the meter.

Another test that was used includes specifically tests to measure the reproducibility of the measurements on legacy hardware. For this reason, PowerKap was also evaluated on a pair of Toshiba Portege R830 with Intel i5 2520M (a 2 physical, 2 virtual core processor) processors, 4GB of RAM and 13.3in screen. All the machines were booting and running tests purely from the USB stick listed previously. This was to avoid the difference in performance degradation caused by legacy hard disks.

Unless specified otherwise, each test was repeated 3 times.



**Figure 38:** The custom built gaming desktop and the power meter used for testing.

## 4.2 The Profiler

### 4.2.1 The Results

Before proceeding with the evaluation, it is important to be aware of what each diagram displays. The energy graphs represent the energy consumed by various CPU domains including Uncore, DRAM and Core. The package total in this case is the sum of all these components and represents the total energy for the CPU. For each graph, the y axis corresponds to the change in a normalised change in energy counter. The units in this case are  $\mu$ joules per second. This is equivalent to the instantaneous power consumed for that domain. The graph in this case, simply plots the power consumed for these various domains against time.

The battery graphs similarly represent the total power consumed by the system as a whole. On the y-axis, the graph displays the actual instantaneous power values for the system. This was calculated by taking the product of the current (mA) reported by the battery and the voltage (mV). The value was then converted to Watts. This obviously comes with some loss of precision in this conversion. The units in this case are joules per second.

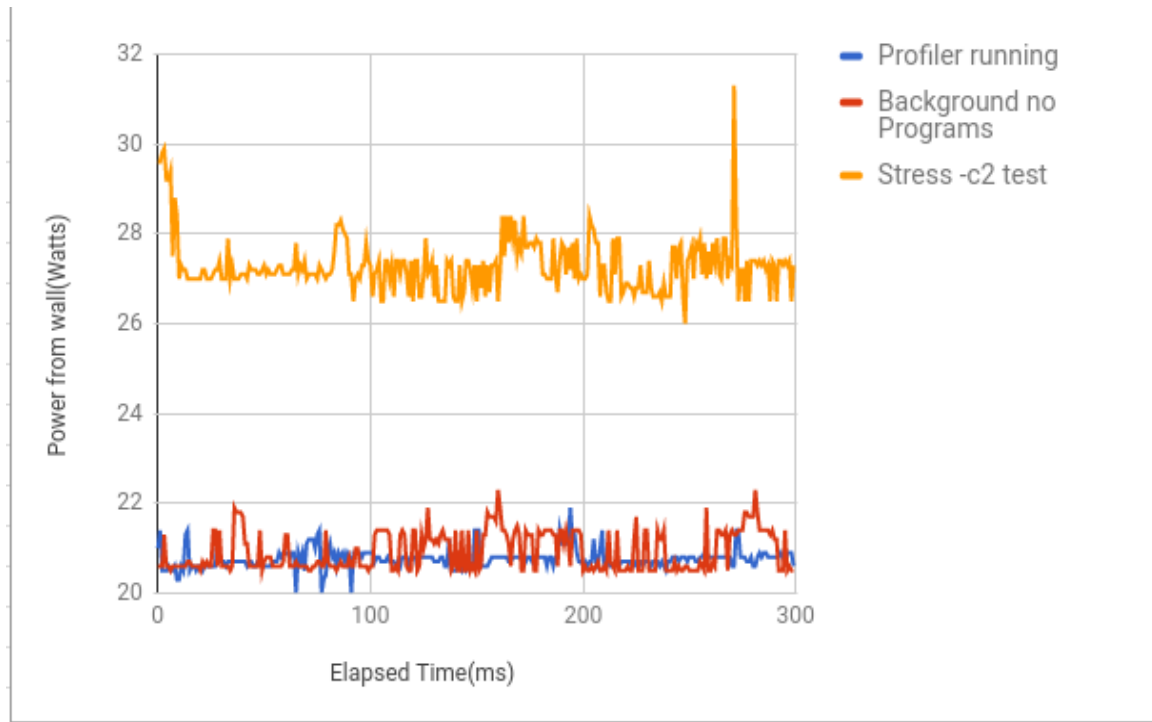
The IO graphs, are designed to show the number of bytes transmitted and sent. This is because according to our research, the wireless bytes sent and received directly correspond to the energy consumed by the device. A similar linear relationship is also known with hard disks.

The final graph represents a change in temperature against time for various components. The units are recorded in micro degrees Celcius.

## 4.3 The Battery Measurements

One of the first aspects that is important to evaluate is the battery measurements. This is particularly useful for evaluating the total power consumption for a given system. If the methodology used for calculating this is correct, it can prove to be a useful reference point to compare the rest of the energy measurements. This test was run using the Dell XPS 13 and Maplin power meter.

Figure 39 shows the results of three tests. These consist of power drawn from the wall under various background and profiling conditions. The stress test used was generated using the stress command. This is a package which is specifically designed as a workload-generator. In this case, the test is simply to calculate the square root of a random number. This utility causes the CPU to maximise utilisation at 100%. From the test shown in Figure 39, we used a maximum number of threads of 2 as that creates the highest effective workload for the system. The idea is to try and estimate whether the energy consumption across both mechanisms are consistent. It was also important for the purpose of estimating the actual power consumption of PowerKap. Within the graph, some interesting results can be observed. Namely, by averaging the stress results over 5 minutes, shows an average average power consumption of 27.3W. In contrast, the sensor reported an average reading of 20.9 and 20.8 with the systems running PowerKap and background alone. This is approximately 6.4-6.5W lower than the stress test. This test is also interesting as it highlights the minimal impact of PowerKap when run with a sampling rate of every second. There are few possible reasons for this such as the fact that the sampling rate for PowerKap could be too low to



**Figure 39: A graph of the various out-of-band power results.**

have an effect on the power consumption.

This minimal impact of PowerKap is corroborated by Figure 41 which shows the exact same series of tests performed generated from PowerKap. Averaging out the power measurements during the run results in an average power consumption of 15.9W for both the background and PowerKap. This value is significantly lower than the 21W reported by the power wall. Reasons for this could include power losses caused by the ac-dc power adapter used by the laptop. This was a limitation of the test as there isn't a methodology of capturing the energy past the adapter without special equipment. Another potential power loss includes hardware specific features that are designed to be more power efficient on battery. To mitigate such potential hardware issues, various hardware components such as the monitor were turned off to ensure the only difference in the test is the execution. Under stress, the average power for the test was reported as 21.3W. This is only 5.4W higher than the background test. An interesting aspect of this test is the comparison between the power graphs for the stress test. What is clear from repeated runs of the experiment is the particular power spike in the beginning following a plateau. This trend is demonstrated in both the out-of-band and in-band approaches.

In terms of quality of the results, it is difficult to determine the actual accuracy of the results reported from both sensors. This is because the battery sensor does not provide any accuracy or precision guarantees for the values gathered. All that is offered is the information presented in Figure 40. The values aren't particularly reliable. This can be seen from the first 25,000ms where there are periods with approximately 3W difference under

```
Handle 0x1600, DMI type 22, 26 bytes
Portable Battery
  Location: Sys. Battery Bay
  Manufacturer: SMP
  Manufacture Date: 03/31/2015
  Serial Number: 0208
  Name: DELL RWT1R43
  Design Capacity: 52940 mWh
  Design Voltage: 7640 mV
  SBDS Version: 1.0
  Maximum Error: 2%
  SBDS Chemistry: LiP
  OEM-specific Information: 0x00000801
```

**Figure 40:** Laptop battery information offered within the command “dmidecode”.

load. However, this test was useful for determining the power profiling trend, which can be useful for developers. In order to determine the accuracy, it would be better to perform the experiment in laboratory conditions in a standardised environment.

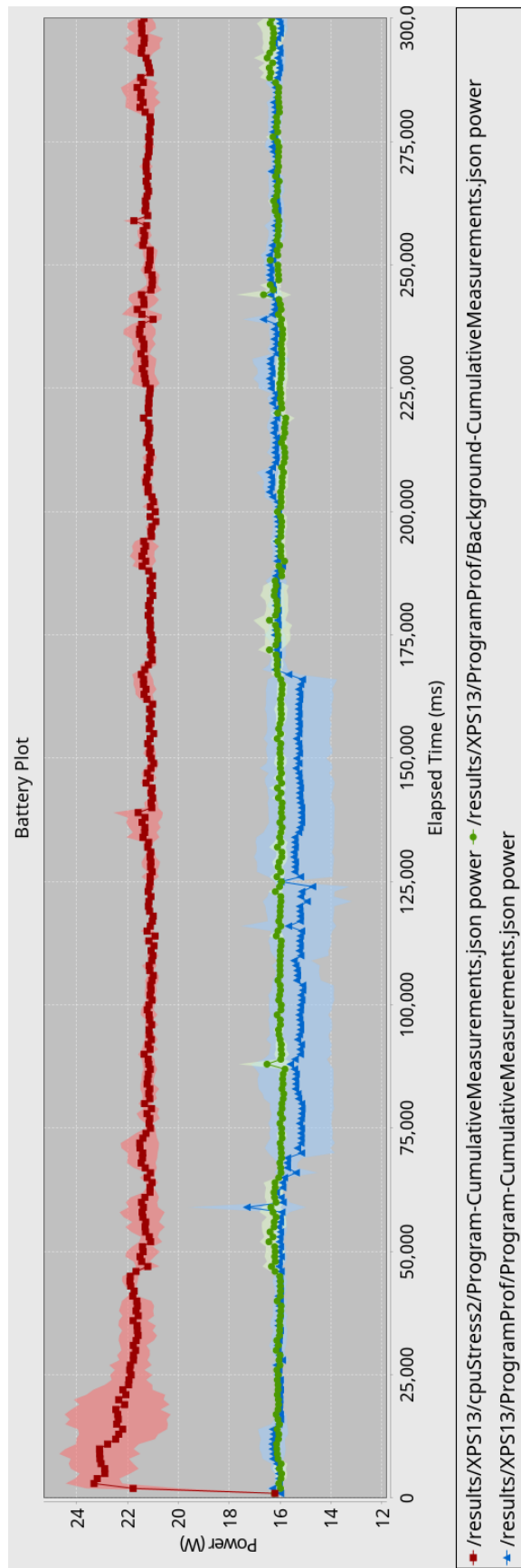


Figure 41: This graph shows the power consumption of three aspects. The first being a stress test which is shown the red line. The green line represents the measurements when testing PowerCap within itself. Finally, the blue line is the background.

## 4.4 CPU Measurements

### 4.4.1 Battery vs CPU measurements

This section evaluates the credibility of the CPU power metrics when evaluating the power consumption of programs. The approach taken to evaluate these metrics is to compare the results with corresponding battery power data. These tests are important to establish the accuracy and boundaries of the various CPU measurements.

### 4.4.2 CPU Stress Test

Figures 42-45 show the result of running the same stress package as listed previously in the battery section. This test was designed to explore the effects of multithreading on energy consumption and to test the limitations of PowerKap when starved of resources. Within the four graphs, various observations are shown that are particularly useful for profiling energy. The first being the difference between one and two thread energy consumption. This can be seen in Figure 42 where a distinct difference between the blue and red lines can be observed. This indicates that as expected, the difference in using multiple cores does lead to an increase in energy consumption. Another interesting point to note is the difference between the background, one core and two core stress test. In this case, a user may expect that the energy consumption for stressing two cores perfectly would be almost double the energy consumption of single core. This is clearly not the case as the graph shows stressing a single core, is almost as energy impactful as stressing two cores. This is probably because the CPU loses energy benefits that are attained when both cores are idle. Mainly the ability to achieve higher package C states.

Another interesting observation from the graph is the impact of hyperthreading. This is a hardware feature available on certain Intel processors that allow the simulation of virtual cores. In this case, we can observe that in this case the impact on the CPU energy consumption is roughly the same as that of two cores. This can be seen by the matching red and green lines seen in 42.

From Figure 43, we can see another consequence of this technique with respect to energy profiling. In this case, we can clearly see how under all the stress test conditions, the energy consumption of the memory system is reduced relative to the background DRAM energy consumption. One possible reason for this is likely to do with the consequence of the stress test. In this case, as the CPU Utilisation goes to 100% on both cores, background processes may be put to sleep. As a result, their resources are released. This observation is corroborated by the minimal difference between the single thread run and the dual thread run. In the single core case, some background processes can still continue which is not the case for the four thread case. This result in particular highlights the limitation in simply finding the difference in energy consumption in idle versus stressed conditions. From this example, we can see that in resource constrained situations, certain components can consume less energy relative to just the background.

Both Figures 44 and 45 are the results of the stress test from both the battery and the CPU. These can be used to verify the accuracy of the measurements. In this case, we compare the package energy consumption relative to the power consumption reported by the battery. In

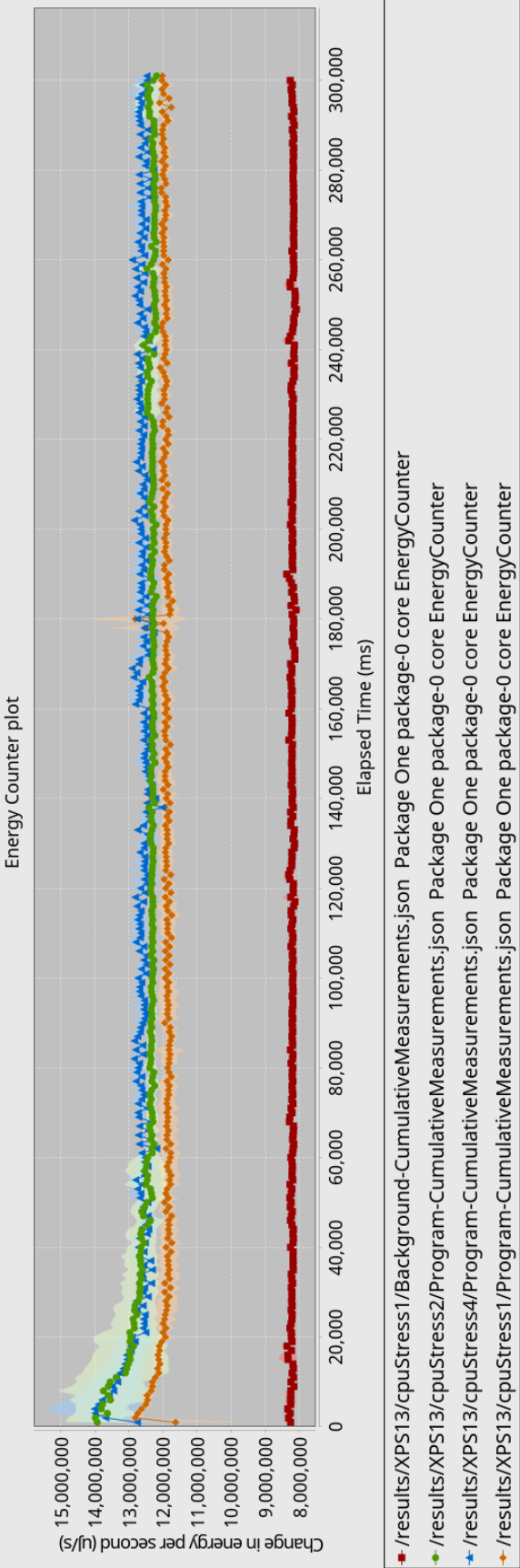


Figure 42: Dell XPS 13 Stress test, Core Energy graph

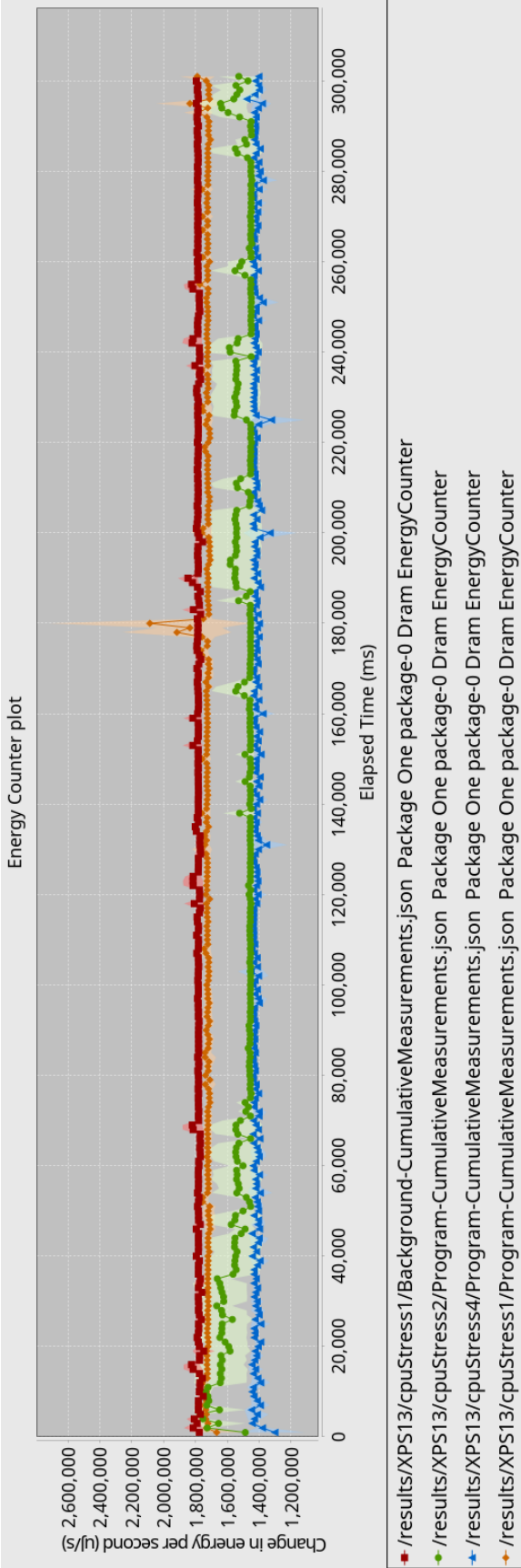


Figure 43: Dell XPS 13 Stress test, DRAM Energy graph



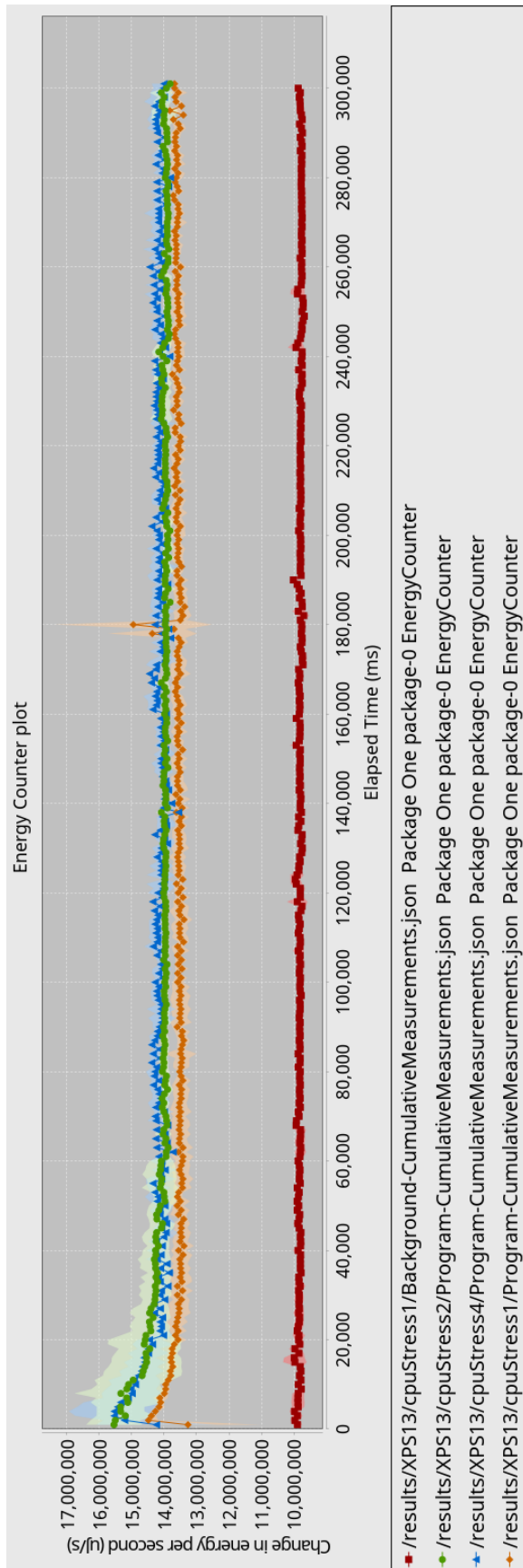


Figure 44: Dell XPS 13 Stress test, Package Total Energy graph

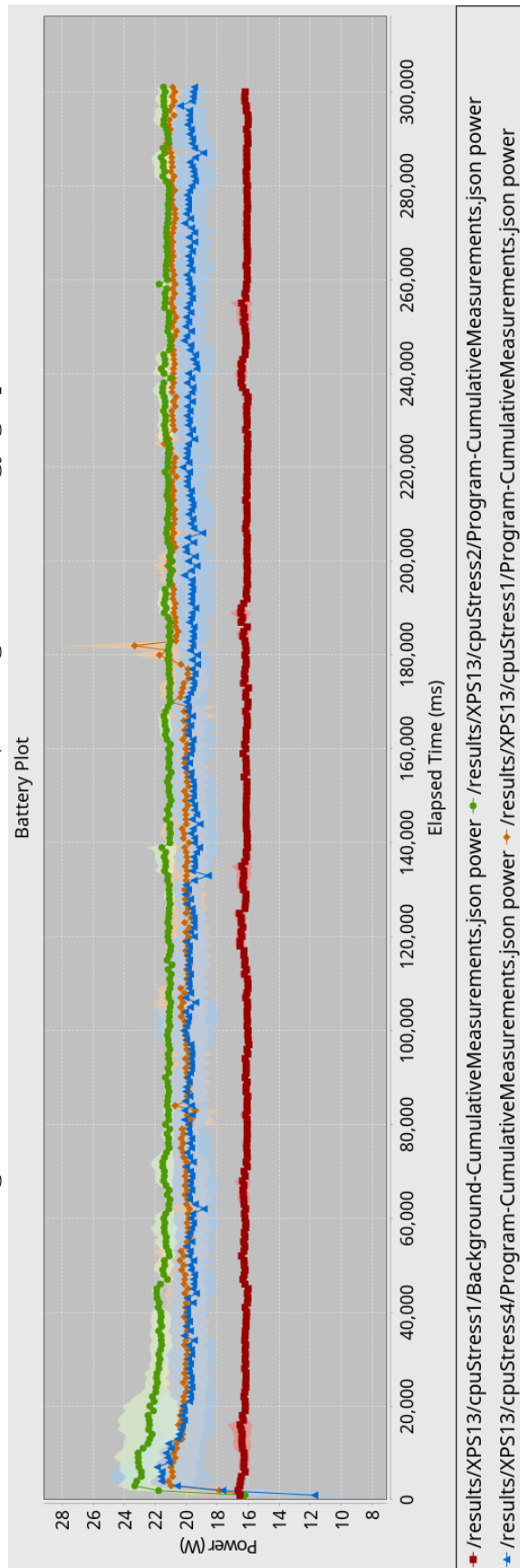


Figure 45: Dell XPS 13 Stress test, Battery Graph

both situations, we can see a similar general trend in the graph corresponding to a sharp initial spike following a plateau in the graphs. In both graphs, we can also clearly see a sharp distinction in energy consumption between the background and multithreaded examples. The distinction between two and four CPU stress test are less clear from the battery test compared with the package test. This is likely because the battery measurements are less accurate due to variability in temperatures and device performance.

### 4.4.3 Desktop vs Laptop

In addition to the previous experiment, the process was repeated on both a gaming desktop platform and a modern laptop. The main idea for this experiment was to see if there would be a noticeable difference in energy patterns for desktops and laptops. The other main goal of this test was to see if the same trend in energy consumption per core follows on other platforms. This test is important because it demonstrates the capability of the system for generalising a energy consumption trend across different platforms.

#### 4.4.3.1 Gaming Desktop

The first aspect to evaluate was the results gathered from the gaming desktop. These can be viewed in Figures 46 and 48. These graphs are also quite interesting from a energy perspective, especially when developing programs. From Figure 46, we can observe some major differences compared to the results gathered from the laptop CPU test used in Figure 42. The first and most striking difference being the clarity of the results gathered relative to the laptop results. In this case, the results are well-defined with minimal deviation with respect to energy results. The standard deviation in this case for all the results present for the energy consumption of the CPU package are less than 1 million microjoules per second. This is in comparison to the same stress test on the laptops which all have a maximum standard deviation of 1.4-2.5 million microjoules per second for the package energy. There are multiple possible explanations for this large difference. This includes potentially an impact caused by thermal throttling in the case of the laptop. Alternatively, this could also be explained by the difference in governor used. In this case, the laptop p-state governor may prioritise power-consumption over the more balanced governor used by the desktop. Another potential explanation could be the difference in architecture between the KabyLake and Broadwell processors although this aspect would be difficult to measure without a large sample size. In any case, all of these issues demonstrate that the same test varies across platforms, as you would expect for systems with different hardware.

Another interesting aspect of this test is the energy rate used per core. For each additional CPU thread stressed, the desktop has a consistently higher energy consumption per core used. For example, the difference between the background and 1 core is approximately 8.4W. Similarly, the difference between 2 and 3 cores is about 9.3W of additional power consumed. However, the difference between stressing cores 3 and 4 is negligible, with only an addition 0.3W on average being consumed. This can be observed from Figure 46, where we can observe a orange line almost matching the green line. Possible reasons for this behaviour includes the possibility that at least one core is always reserved for the Operating System. This isn't to say that stressing with an additional core does not have an effect. In Figure 48, we can observe a significant variability in the Uncore energy rate specifically for the 4 thread stress test. This is a surprising result, as the same trend is not observed

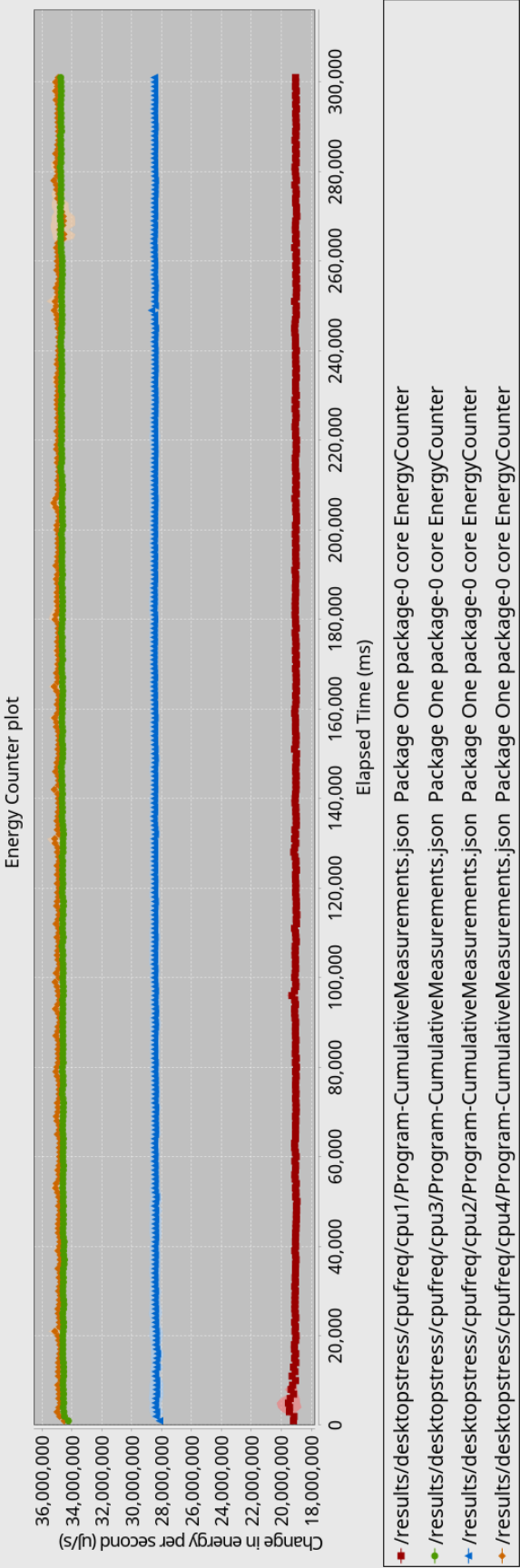


Figure 46: Gaming Desktop, Stress test, Core Energy graph

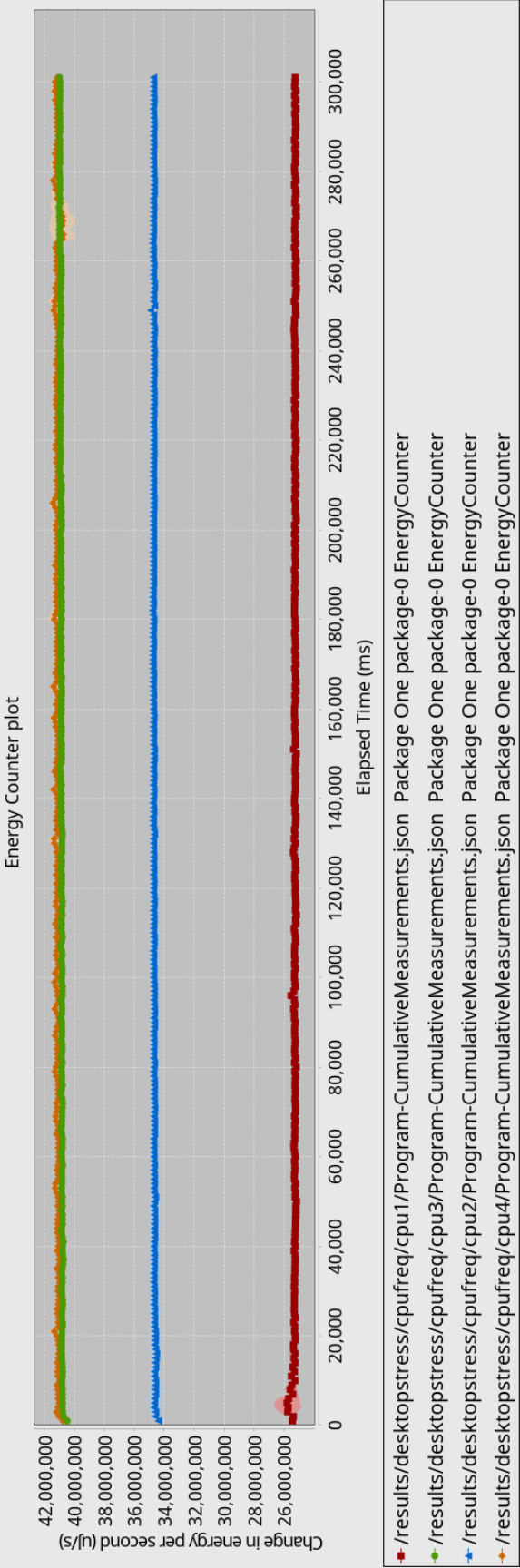


Figure 47: Gaming Desktop, Stress test, Package Total Energy graph

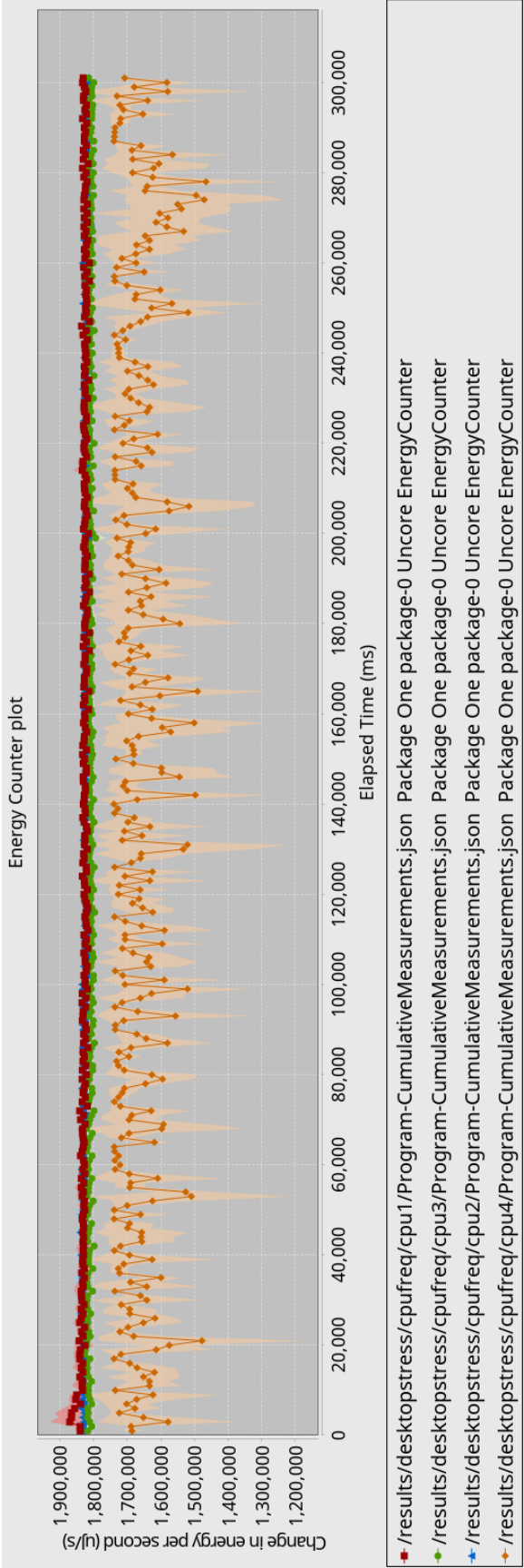


Figure 48: Gaming Desktop, Stress test, Uncore Energy graph

on laptop computers. Within Intel's documentation, the components referred to as Uncore are not explicitly defined as to what the energy domain refers to. In this case, it is likely to correspond with potentially cache or memory controllers as this behaviour matches the reduced energy consumption displayed in Figure 43.

Another important aspect of these graphs relative to the laptop results is the difference between the package total energy and the core energy. In the desktop case, the graphs show an average of about 7W consumed between the cores and the rest of the package. In comparison, the laptop measures a difference of only 1.5W. This large difference in energy consumption per core, and the energy consumed for the rest of the package can likely be explained by the fact that the laptop processor is likely designed to be more efficient. These differences are important to consider depending on the target application, because in some cases it appears to be more energy efficient to use more cores on the laptop relative to the desktop.

#### 4.4.3.2 Modern Laptop

In this section, we explore the results gathered from a modern laptop from the same architecture as the desktop. These can be viewed from Figures 49 and 50. The first striking information gathered from this test was the difference in energy measurements from a modern laptop in comparison to the Dell XPS 13. In this case, the results gathered in Figure 49 follow closer the graphs generated with the gaming desktop. The results gathered are distinct with little spread or variation in the data. This again is likely to be due to potential thermal throttling on the Dell XPS 13 relative to the newer laptop. From this graph, we can also observe how threads 5-8 do not have a noticeable energy impact. This observation follows the hyperthreading observations noticed within the 4 threaded CPU test previously. In Figure 50, we can see the memory impact of the various tests. In this case, nearly all of the results have a memory energy consumption of around 0.5J. The exception in this case being the single thread test which do not appear particularly reliable.

Another possible theory behind the difference is likely to be due to the difference between the Intel I7 5500U and the i7 7700HQ. The Dell XPS 13 laptop is using an ultraportable processor. For this reason, it may have different power peaks relative to the Dell XPS 15. This test further corroborates that particular energy behaviour on different Intel platforms are not necessarily consistent. For this reason, programs should be tailored specifically to a target platform.

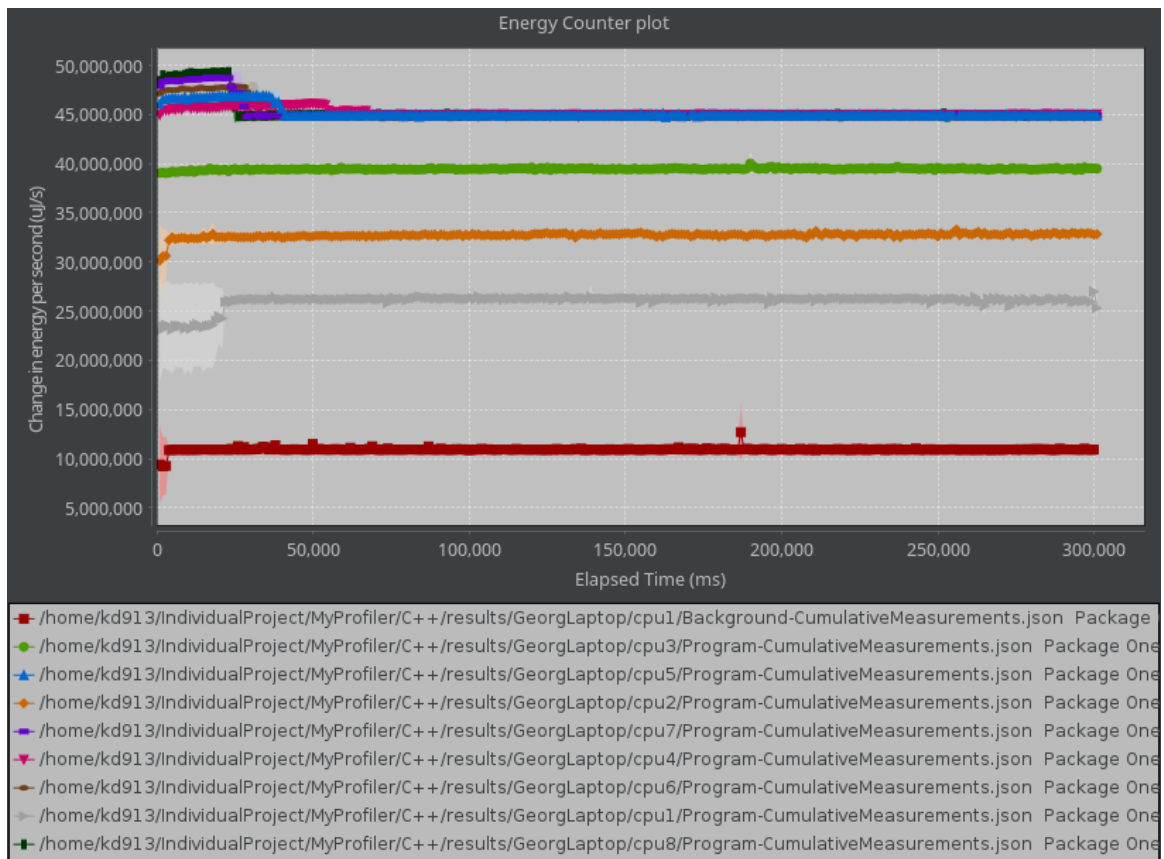


Figure 49: Dell XPS 15, Stress test, Package Total Energy graph

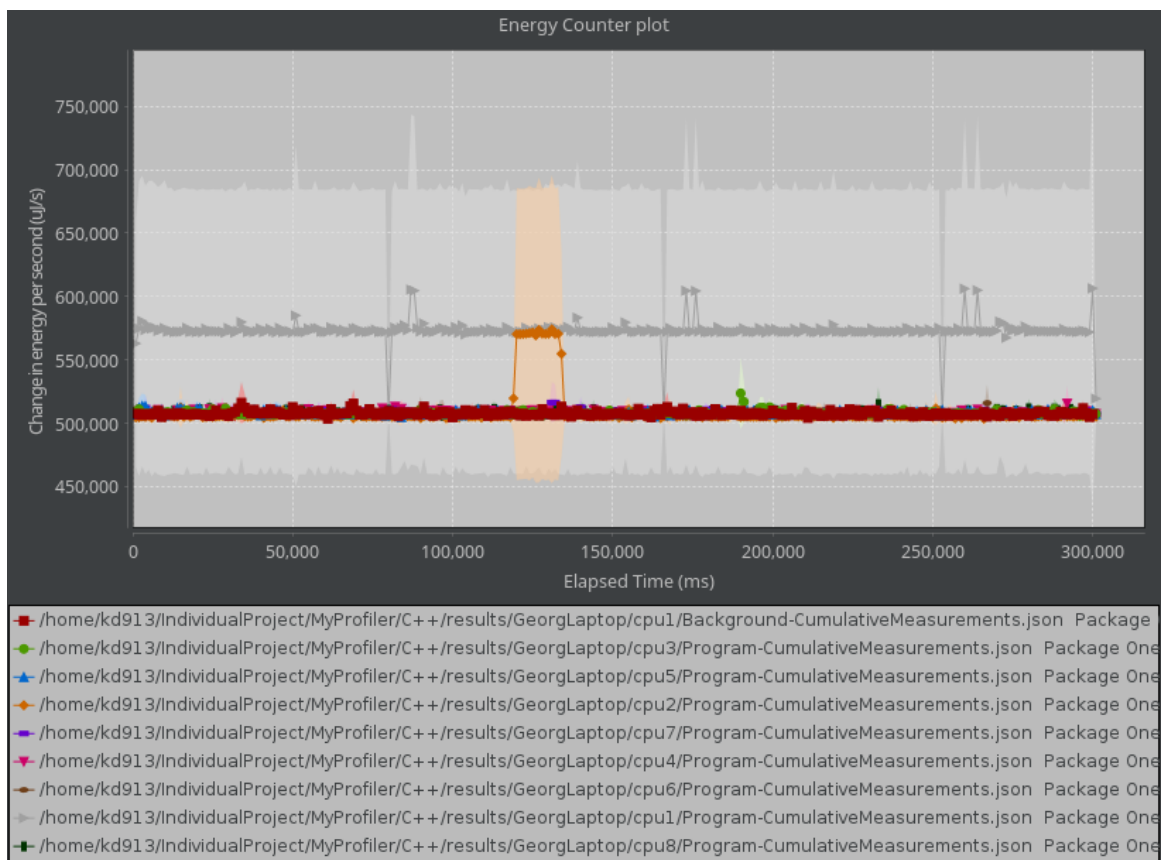
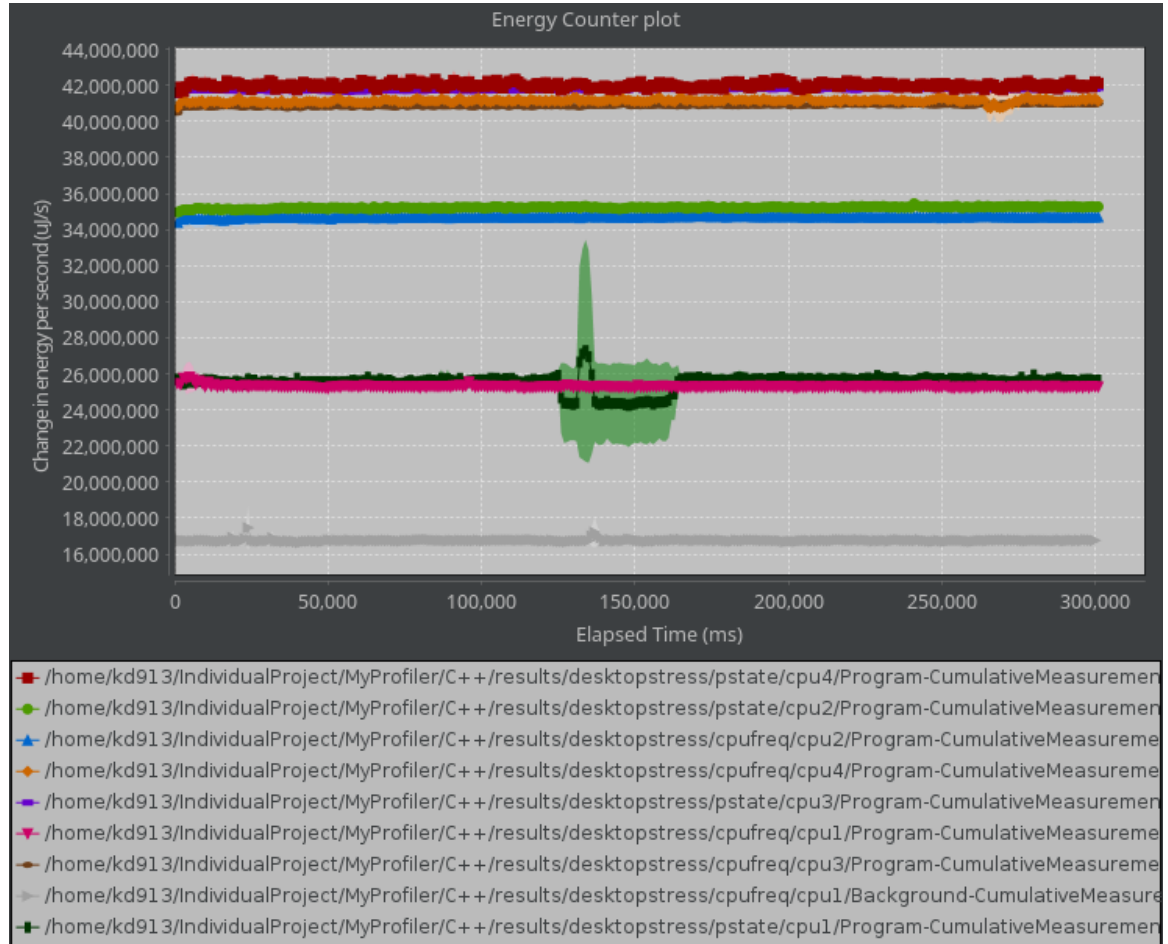


Figure 50: Dell XPS 15, Stress test, DRAM Energy graph

#### 4.4.4 Governor Choice

As discussed in the previous selection, one possible reason for the difference in energy consumption could likely be due to different governors. This governor is responsible for managing the power states of the CPU. In this experiment, the aim was to eliminate this as a variability in the results. To do so, the test repeated the same stress experiment used previously but with the Intel P-state governor. This governor has recently been patched in Kernel 4.10 to ensure more performant behaviour on desktop platforms [61].

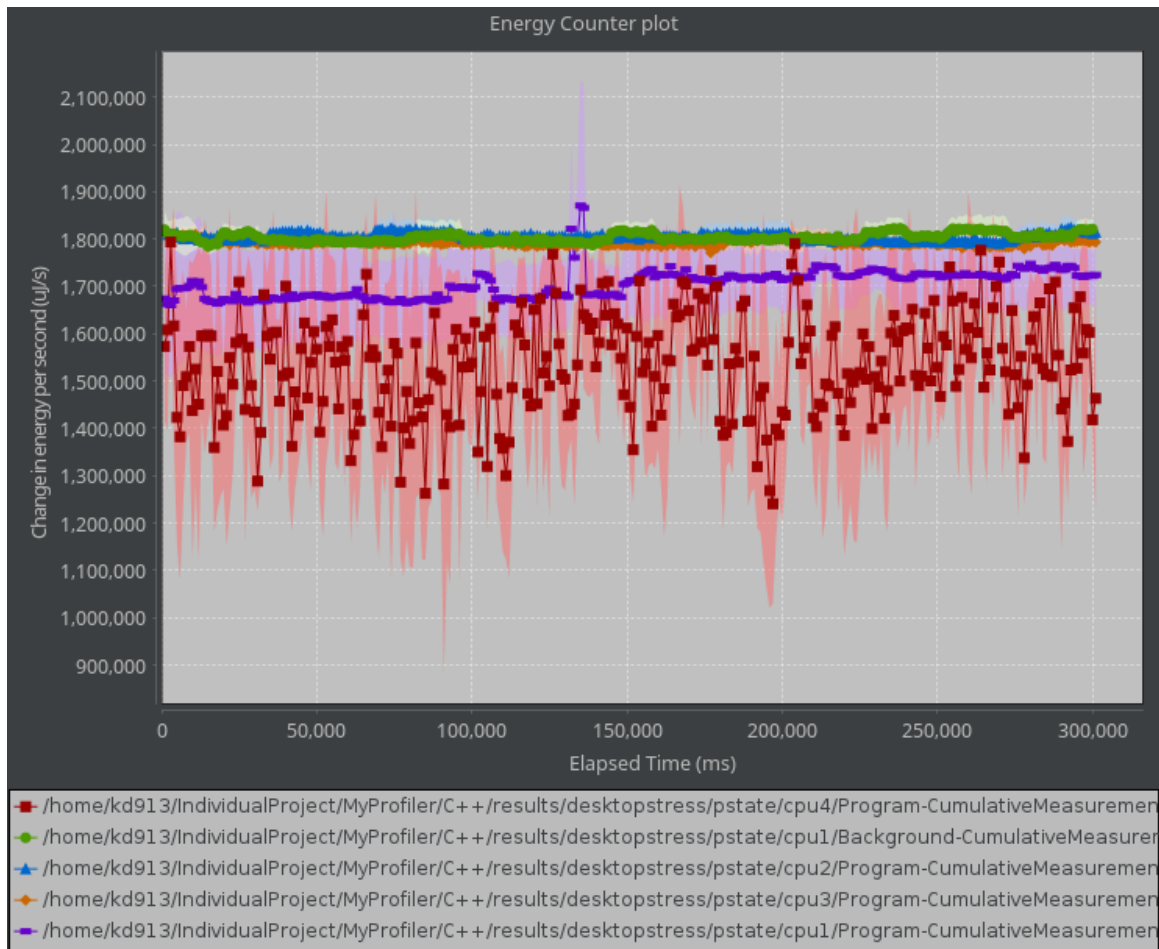


**Figure 51: Gaming Desktop, Pstate vs CpuFreq Governor test, Package Total Energy graph**

From Figure 51, we can see an overall picture of the energy rate for both governors. In general, both governors seem to have the same power behaviour. The exception to this is the single core performance at around 150000 milliseconds during which we can see a spike in energy.

The Uncore graph in this case, is a lot more interesting. In this case, comparing Figures 52 and 48, we can see that the general trend for multiple cores appears to be the same.





**Figure 52: Gaming Desktop, Pstate vs CpuFreq Governor test, Uncore Energy graph**

The exception is the single core energy consumption. In this case, there is a large variability in the single core performance test which was not observed with the CPUFrequency governor. This is observed from the purple line which is significantly lower than the rest of the graph. This likely adds to the theory that the governor may introduce variability in the energy consumption in exchange for power efficiency.



#### 4.4.5 BigBuckBunny Mplayer Test

The tests so far have been designed to test the behaviour of various CPU platforms under extreme workloads. These can be particularly unrealistic in real world situations. For this test, the project is exploring a more realistic benchmark of a typical workload. This test involved running Mplayer's benchmark [62] with various number of threads. This test works by testing the video playback of a 1080p video titled BigBuckBunny [63]. This test was performed on the desktop as this test was specifically designed to test the differences in physical cores. The results of this test can be observed from Figures 53-55.

The results of this test were particularly interesting as it mainly corroborates the guidance provided by Intel with respect to multithreading. The first interesting point to observe with this test is the reduction in execution times for the different physical core cases. In this case, we can observe a significant reduction in execution time by running the test with two threads relative to the one. The single thread example ran in around 364 seconds whilst the two thread test ran in 217 seconds respectively. This is a reduction of 40% execution time which is significant. In contrast, the four thread and three thread tests both ran with an execution time of 174 and 181 seconds respectively. A gain of 52% and 50.2% in execution time.

When comparing the average package energy consumption for this test, we found that the single thread consumed 8,175J. The two thread test ran with an energy consumption of 5,793J. Finally the three and four thread tests consumed on average 6,535J and 5,972J in total. This result is interesting as it shows that the two thread case is the most efficient solution compared to all the other solutions. It's also useful for a usability perspective as consuming all the four threads significantly impacts temperature and usability of the platform. This test demonstrates the utility of PowerKap for enabling developers to be aware of these decisions.

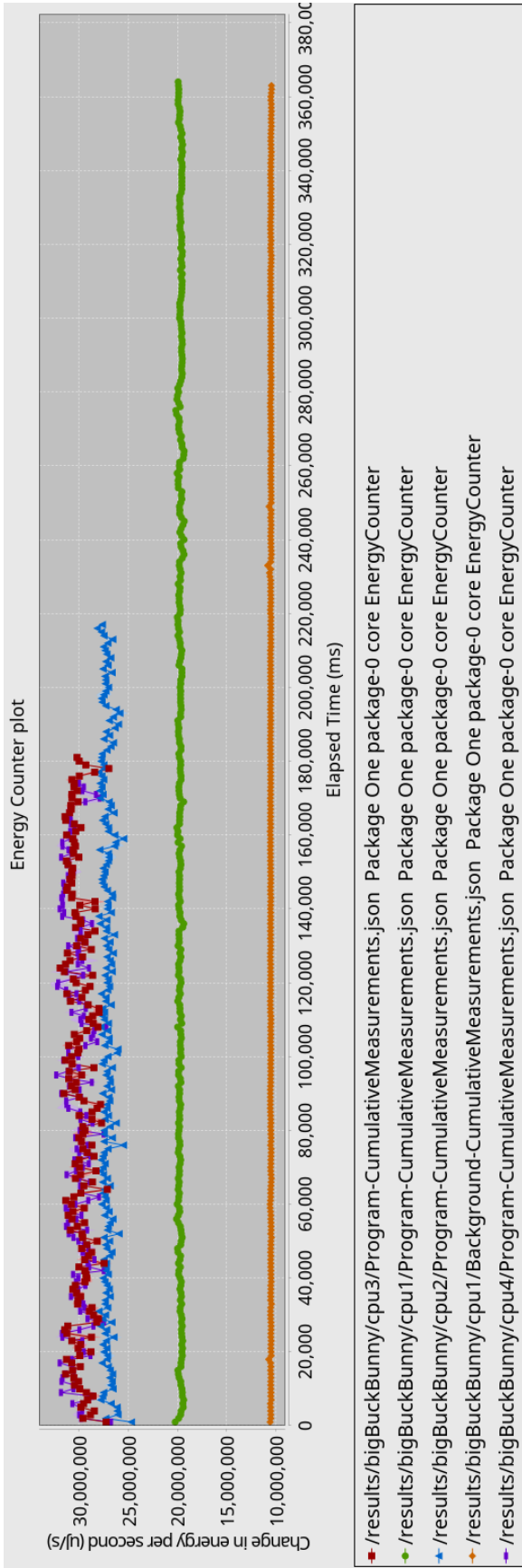


Figure 53: Gaming Desktop, Big Buck Bunny Test, Core Energy graph

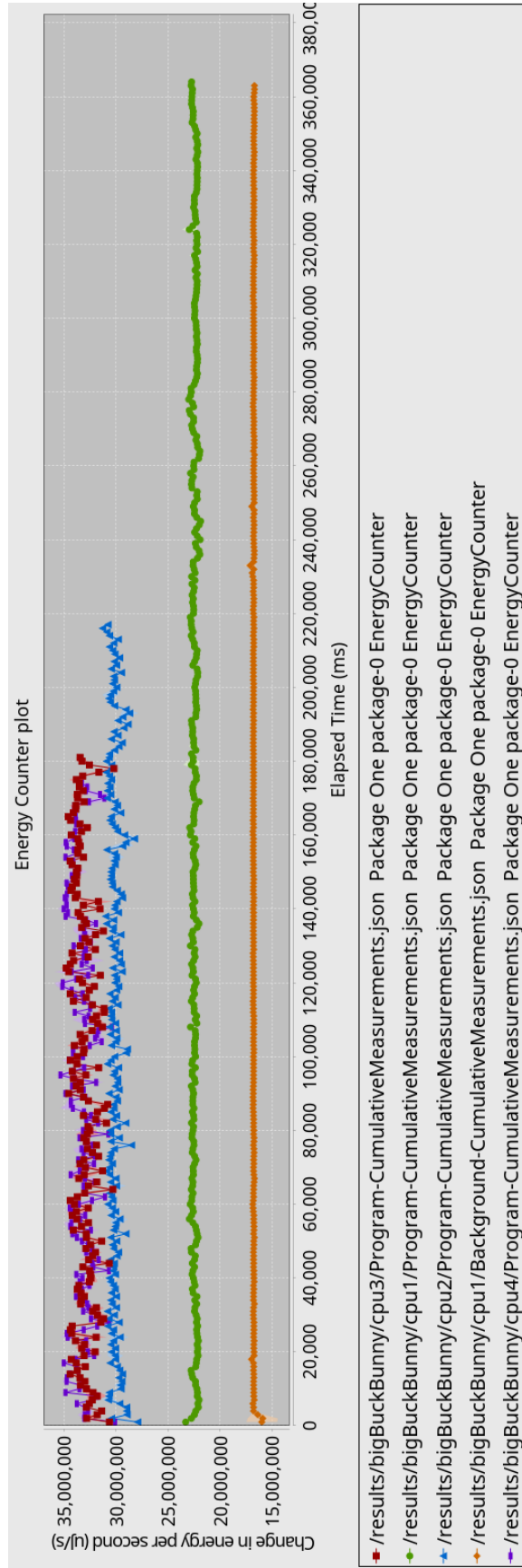


Figure 54: Gaming Desktop, Big Buck Bunny Test, Package Total Energy graph

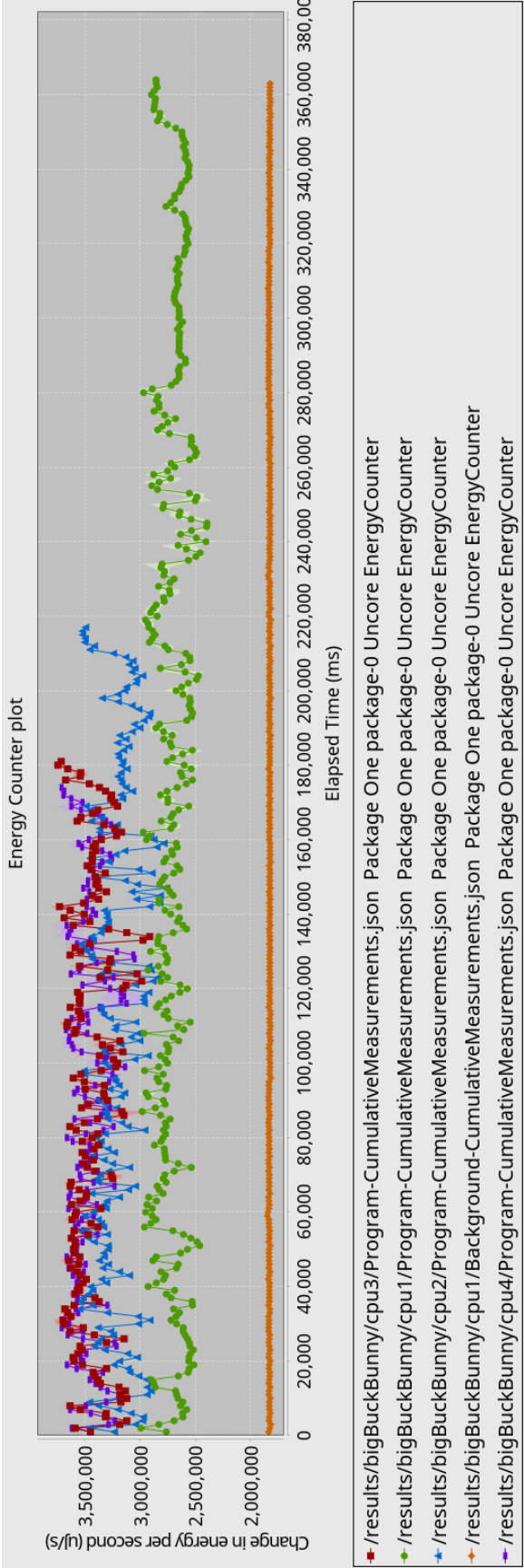


Figure 55: Gaming Desktop, Big Buck Bunny, Uncore Energy graph

#### 4.4.6 Choice of Algorithm

As discussed in the background section, the choice of algorithm can have a significant effect in the energy efficiency of a program. One of the goals of the profiler was to be able to distinguish the different energy results for the various algorithms. To test this, a benchmark was created that compared the effect of various sorting algorithms. This mainly comprised of 4 algorithms consisting of insertion sort, merge sort, bubble sort and randomised quick sort. For each test, the procedure was as follows.

1. Delete all previous results from fileSystem
2. Create a random generated array from a specific seed
3. Write array to disk
4. Sleep for 5 seconds
5. Read array back
6. Perform sorting algorithm
7. Sleep for 5 seconds
8. Write results to disk

This procedure was chosen to try and eliminate potential influences of caching by reading the results from disk. This approach didn't quite work which will be discussed later. However, it did seem to produce noteworthy results. The graphs below were generated from 3 separate runs of the sorting test.

Based on the results shown in Figure 56, we can see some useful observations from this execution of the program. When timing the various sorting algorithm, it was found that Insertion sort algorithm took about 30 seconds, Merge sort 0.12 seconds, Bubble sort 500 seconds and randomised quick sort 0.08 seconds. Based on this timing, and the corresponding energy graph, it is quite clear to identify the portions of code corresponding to the insertion sort and merge sort. It also highlights specifically the importance of using an appropriate algorithm. The merge sort and quick sort algorithms happened before and after the bubblesort test. In this case, the various sorting algorithms are indistinguishable from the thread sleep stage when the energy consumption of the system effectively matches the background. The reason for this is partly due to sample rate and the speed at which the algorithm completed.

The sleep times in this case can be seen at the points in which the green and red lines meet. These results are corroborated by the battery graph in Figure 59, where we can see the red line effectively match the green line within the graph.

Another interesting aspect of the graph are the spikes in the background energy at around the 100,000 millisecond and 200,000 millisecond points. This large variability was likely caused by background GPU processing. This can be seen in Figure 60 which compares the Uncore and the package total. When we compare the two graphs, we can see that the fluctuations appear to occur at points in which the Uncore values seem to change. These seem

to demonstrate the influence of these Uncore energy totals with respect to the package as a whole.

These results are useful as they demonstrate how algorithm choice can clearly impact energy choice. These will hopefully be useful for developers to at least be capable of distinguishing energy inefficient algorithms and behaviours. Unfortunately, this test also demonstrates some of the flaws and difficulties with this methodology. Namely that it can be difficult to estimate the energy of particularly optimised or in this case quick code.

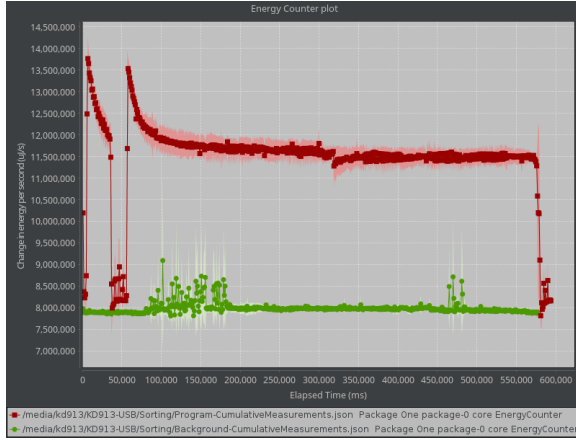


Figure 56: Dell XPS 13, Sorting Test, Core Energy graph

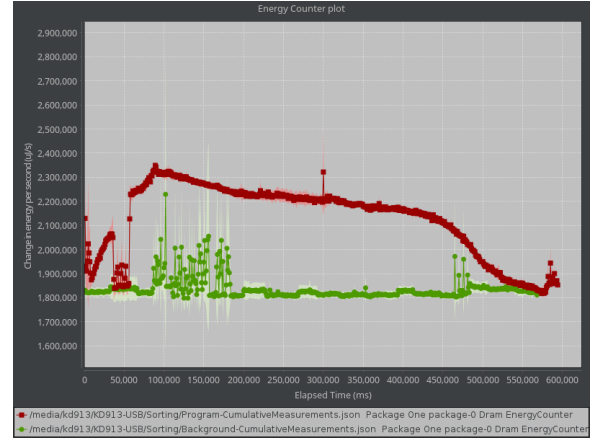


Figure 57: Dell XPS 13, Sorting Test, DRAM Energy graph

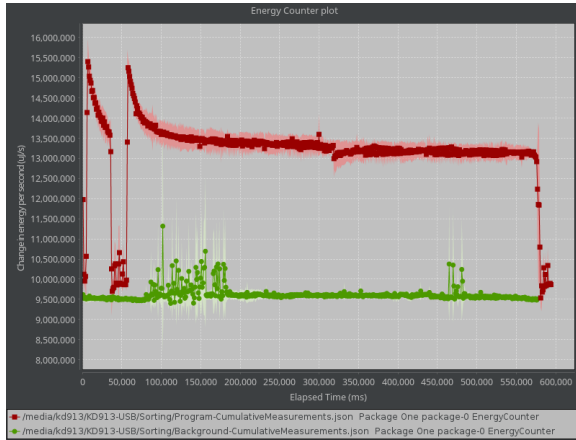


Figure 58: Dell XPS 13, Sorting Test, Package Total Energy graph

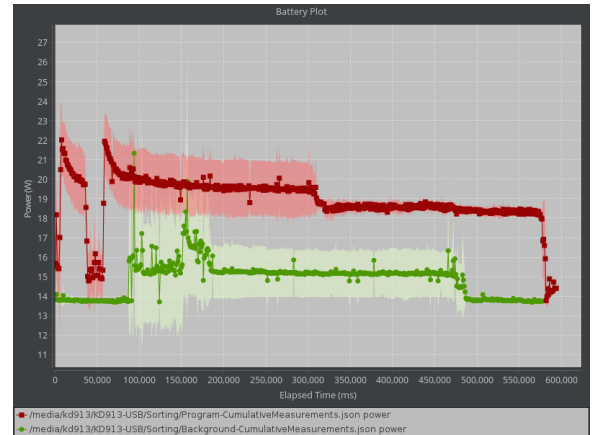


Figure 59: Dell XPS 13, Sorting Test, Battery Graph



**Figure 60: Dell XPS 13, Sorting Test, Uncore to Package Total Graph**

#### 4.4.7 Asynchronous vs Busywait

The main purpose of this test was to evaluate the capability of PowerKap for detecting energy inefficient behaviour with respect to asynchronous and busy-wait designs. In this test, we spawn a secondary thread that is responsible for notifying the parent thread. Each test runs for 5 minutes, with one test checking the effects of a while (true) loop without sleep. The other test performs the same test but with the parent thread waiting for a notification.

From Figures 61 and 64, we can clearly see a noticeable effect caused by the busy-wait test relative to the asynchronous notification test. In this case, the loop appears to consume approximately 5 additional watts during each second of the busy wait loop. This can be observed from the green line which is significantly higher for the first half of the execution. In contrast, the asynchronous loop consumes an energy consumption that matches the background as shown by the green and red lines effectively matching. This at least demonstrates that PowerKap is capable of identifying such problematic energy behaviour.

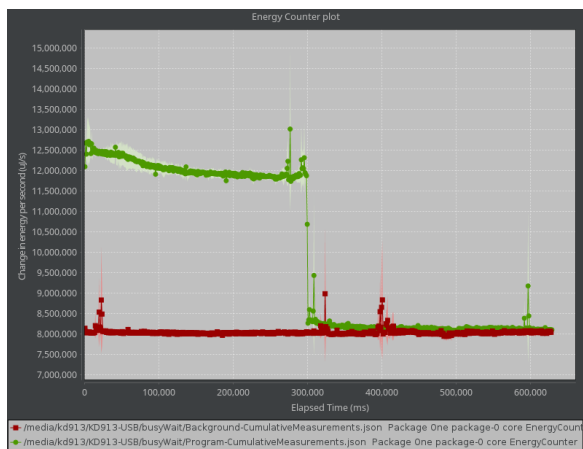


Figure 61: Dell XPS 13, BusyWait Test, Core Energy graph

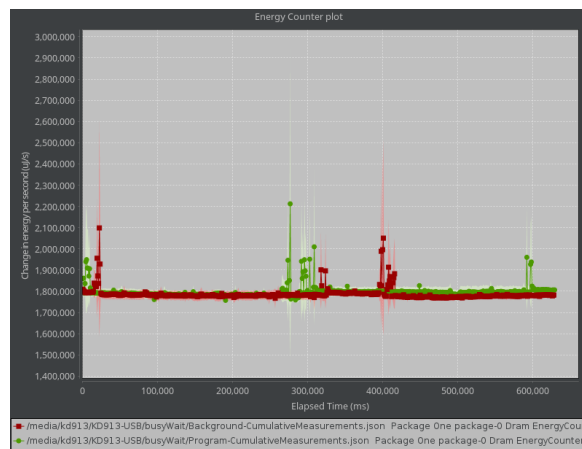


Figure 62: Dell XPS 13, BusyWait Test, DRAM Energy graph

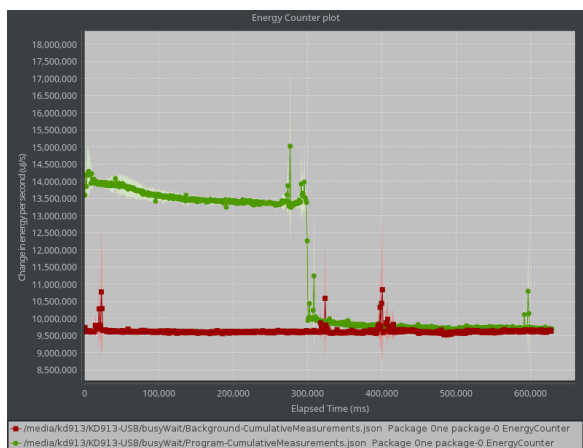


Figure 63: Dell XPS 13, BusyWait Test, Package Total Energy graph



Figure 64: Dell XPS 13, BusyWait Test, Battery Graph

#### 4.4.8 Effects of Timers

Another important concept introduced in the background section was the importance of timers. In this case, the test was to evaluate the energy consumption of various timer based functions on the platform. This capability was tested using PowerKap, to see if there is an effective impact of running the profiler with a smaller timing interval.

In this test, we can see a lot from the energy metrics gathered from the CPU. Namely we can compare the energy performance in the 10ms, 100ms and 1 second sample rate. From Figures 65 and 67, one can see some useful general trends with respect to energy consumption of PowerKap. In particular, the sampling rate for the 10ms example consumes significantly more energy compared to the 100ms and second sampling rate. This is demonstrated by the red line which is significantly higher than the green and blue lines. This trend is matched by the memory and core energy consumption. The result corroborates what was discovered during the background reading that timing loops can significantly impact energy usage.

In terms of actual energy consumption, in this case we compare the average energy consumed by the package during these runs. For the 10ms sampling test, the energy consumed by the package was approximately 6,385J. The 100ms test consumed on average 6006J for the CPU whilst the one second sampling rate consumed 5887J. For reference, the background sampling only consumed approximately 5822J. As such, the 10ms sampling rate consumed on average 10% more energy in comparison to the pure background run. The 100ms sampling test in comparison consumed on average 3% more energy relative to the background run. Finally, the one second sampling rate only consumed 1% more energy relative to the background.

Figure 68 also shows the battery statistics gathered from this test. Unfortunately, the results gathered are less clear cut than the Core energy consumption. In this case, the standard deviation between results are not quite enough to make definitive conclusions. Possible reasons for this is likely to do with temperature and battery condition fluctuations at the time of the test.

#### 4.4.9 Reproducibility of the results gathered

One of the important aspects when attributing the energy results for a program is the importance of the ability to reproduce the results. For this experiment, this test is designed to explore the reproducibility of the physical measurements gathered across the same hardware but with different ages and usage. This test was done on the two identical Toshiba laptops. The idea in this case is to see if the results generated from PowerKap can be attributed towards a platform as a whole.

In this test, the project uses numerous CPU and memory based tests that are also available through other benchmark suites. These tests cover a large set of common CPU bound tasks such as cryptography, ray tracing, video playback and a specific memory benchmarks. These are all common tests that are used in various benchmarks to evaluate CPU performance [64, 33, 65].



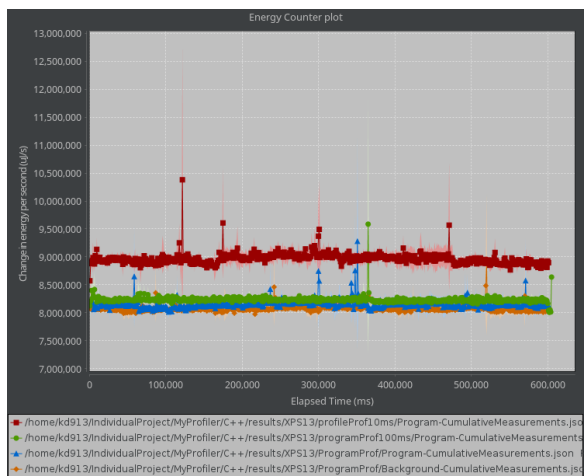


Figure 65: Dell XPS 13, Timer Comparison Test, Core Energy graph

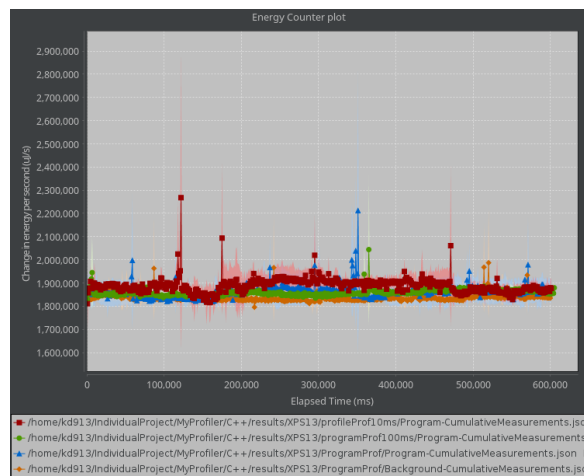


Figure 66: Dell XPS 13, Timer Comparison Test, DRAM Energy graph



Figure 67: Dell XPS 13, Timer Comparison Test, Package Total Energy graph

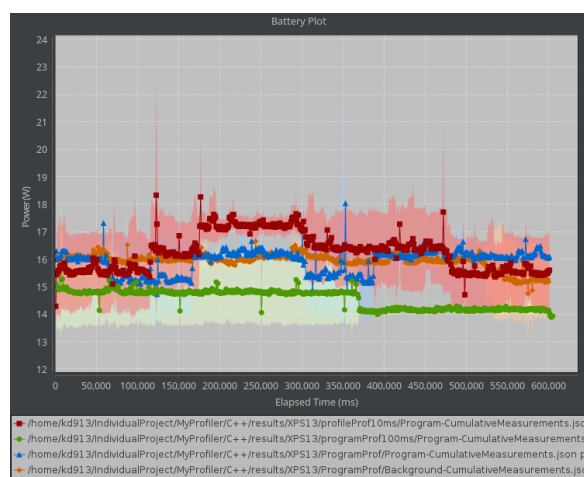


Figure 68: Dell XPS 13, Timer Comparison Test, Battery Graph

#### 4.4.9.1 John the Ripper

The first test was a benchmark provided within a open source program called John the ripper. This program is a password cracking utility designed to break passwords with various hashing algorithms. The results of the two tests are displayed in Figures 69-70.

There are some interesting points to note from the comparison graphs listed above. In Figure 69, we can see the total package energy consumption for both laptops. In both cases, the energy trend is mostly similar for both programs. The exception in this case is the Uncore graph in Figure70 in which we can see a distinct difference between the two executions. In one case, there are far more pronounced spikes in one laptop relative to the other graph. These seem to occur at the same peak points in the other laptop. This could be for various reasons including memory degradation.



**Figure 69: Toshiba Comparison, John The Ripper Test, Package Total Energy graph**



**Figure 70: Toshiba Comparison, John The Ripper Test, Uncore Energy graph**

#### 4.4.9.2 OpenSSL

This test runs an OpenSSL benchmark designed again to measure the energy consumption for a commonly used cryptographic function. In this case, the test measures the speed of the laptops when performing AES-256-cbc.

Observing the above graphs, we can see a similar trend to that noted previously. In this case, the general package energy remains roughly similar for both platforms. The main exception being the results generated for the uncore graph in Figure 72. From the graph, we again observe a difference in the spikes during the execution of both programs.



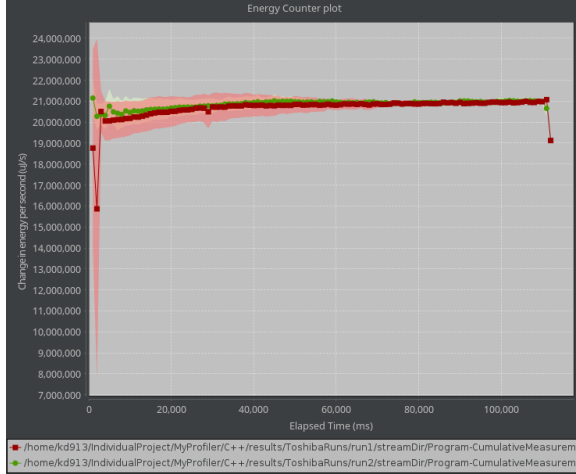
Figure 71: Toshiba Comparison, OpenSSL Test, Package Total Energy graph



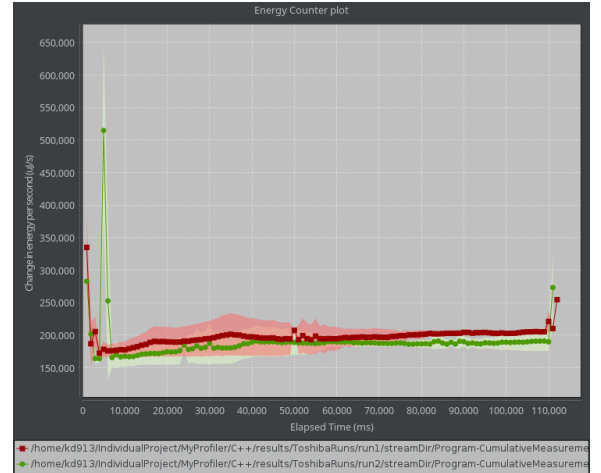
Figure 72: Toshiba Comparison, OpenSSL Test, Uncore Energy graph

#### 4.4.9.3 STREAM Benchmark

The STREAM benchmark [66] is an open source academic benchmark designed to evaluate the sustainable memory bandwidth. In this case, the aim of the project was to evaluate a cache only benchmark to see if the energy consumption is similar on both platforms. From Figures 73-74, we can see the result of this experiment. The result of this experiment is particularly interesting. In this case, both lines follow much closer relative to the previous graphs. Possible reasons for this is because this benchmark is designed to stress test purely the cache structure. As such, it may not be as massively affected by the difference in memory.



**Figure 73:** Toshiba Comparison, Stream Test, Package Total Energy graph



**Figure 74:** Toshiba Comparison, Stream Test, Uncore Energy graph

#### 4.4.9.4 Sunflow benchmark

The Sunflow benchmark, is a Java benchmark provided by the DaCapo Benchmarks [67]. This is a noteworthy benchmark used in famous publications including John Hennessey and David Patterson’s book “Computer Architecture: A quantitative approach” [65]. Within the book, the authors use it to accurately compare the performance of this test against various architectures. For this test, we used the Sunflow benchmark which performs ray tracing on images.

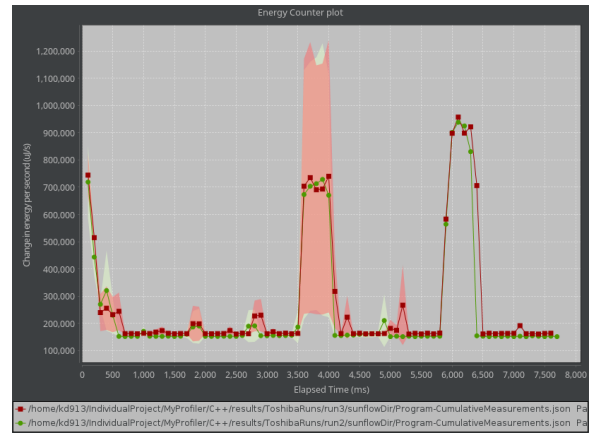
The graphs shown in Figures 75 and 76 are particularly interesting as both laptops appear to follow almost exactly the same execution and energy profile. However, there does seem to be a large variance in the execution especially between 0 and 3000 seconds and 3500-4000 seconds.

#### 4.4.9.5 MPlayer

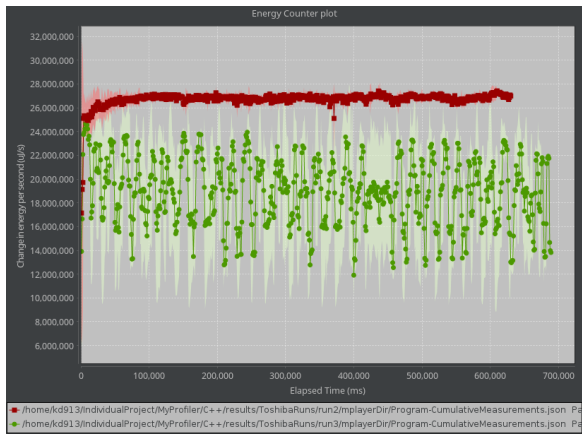
The final test used was a similar test used earlier. This was the Mplayer test which is designed to test the energy performance of a CPU when performing a playback of BigBuck-Bunny. In this test, we evaluate the accuracy of the energy measurements for both laptops. The graphs of these can be seen in Figures 77 and 78. This test was interesting as it demonstrates some of the issues with assuming the same energy profile for identical hardware. In this case, the two laptops performing the same test from the same storage device result in entirely different execution and energy consumption profiles. Possible reasons for this are likely to be due to age of hardware. We have already demonstrated that one laptop consumes different Uncore energy usage, it is not unlikely that the same applies to other aspects to the hardware. For example, in this case, one laptop may have a slower bandwidth when reading the file from the USB. Other possible reasons is likely again to be for thermal reasons. This is not unreasonable for laptops with large use which could accumulate dust. This explains the difference between this benchmark and the desktop result.



**Figure 75: Toshiba Comparison, Sun-flow Test, Package Total Energy graph**



**Figure 76: Toshiba Comparison, Sun-flow Test, Uncore Energy graph**



**Figure 77: Toshiba Comparison, MPlayer Test, Package Total Energy graph**



**Figure 78: Toshiba Comparison, MPlayer Test, Uncore Energy graph**

#### 4.4.9.6 Summary of Benchmark Findings

To conclude from these various benchmarks, this project has demonstrated that in certain aspects, the energy profile for various components act in very similar ways. However, this is not guaranteed and can be affected quite significantly by factors relating to the age of the hardware. In this case, we unfortunately only have a sample of size of two, so it is not enough to form conclusions. However, this trend does indicate that this approach may not be suitable for generalising energy performance across hardware.

#### 4.4.10 Temperature Sensor Data

Another important point that was highlighted with the stress tests was the importance of temperature information. These appear to greatly affect that accuracy of results due to thermal throttling. This chapter evaluates the capability of the system for detecting thermal information. The first test, comes from the thermal sensors gathered during the laptop stress test. The results of this can be observed from Figure 80. This image shows the results of the thermal data captured by PowerKap. The main limitation in this case, is the quality of the sensors used. For example, in the image, we can observe the lime green line representing a thermal package referred to as “pch\_wildcat\_point” which continuously reaches negative values. This is definitely an inaccurate sensor that represents the temperature of the chipset of the laptop. Other challenges include the multiple similarly named thermal sensors present on the laptop, with very little information as to what each sensor represents.

This lack of information can be even worse on custom builds such as desktops which often come with even fewer thermal sensors. This can be observed in the same stress test performed on the desktop as seen in Figure 81.

For each of these thermal sensors, many components do not register any particular information with respect to granularity. This can be seen in Figure 79. This image demonstrates part of the limitations of the system currently with respect to thermal interfaces. Namely, the current open interfaces are heavily underused and do not currently offer granular information with respect to components. This is particularly problematic as it may be useful to know aspects such as register temperatures and cooling energy consumption.

```
Handle 0x003A, DMI type 28, 22 bytes
Temperature Probe
  Description: LM78A
  Location: Power Unit
  Status: OK
  Maximum Value: Unknown
  Minimum Value: Unknown
  Resolution: Unknown
  Tolerance: Unknown
  Accuracy: Unknown
  OEM-specific Information: 0x00000000
  Nominal Value: Unknown
```

**Figure 79:** Information about a thermal sensor on the desktop. This was provided by the “dmidecode” command.

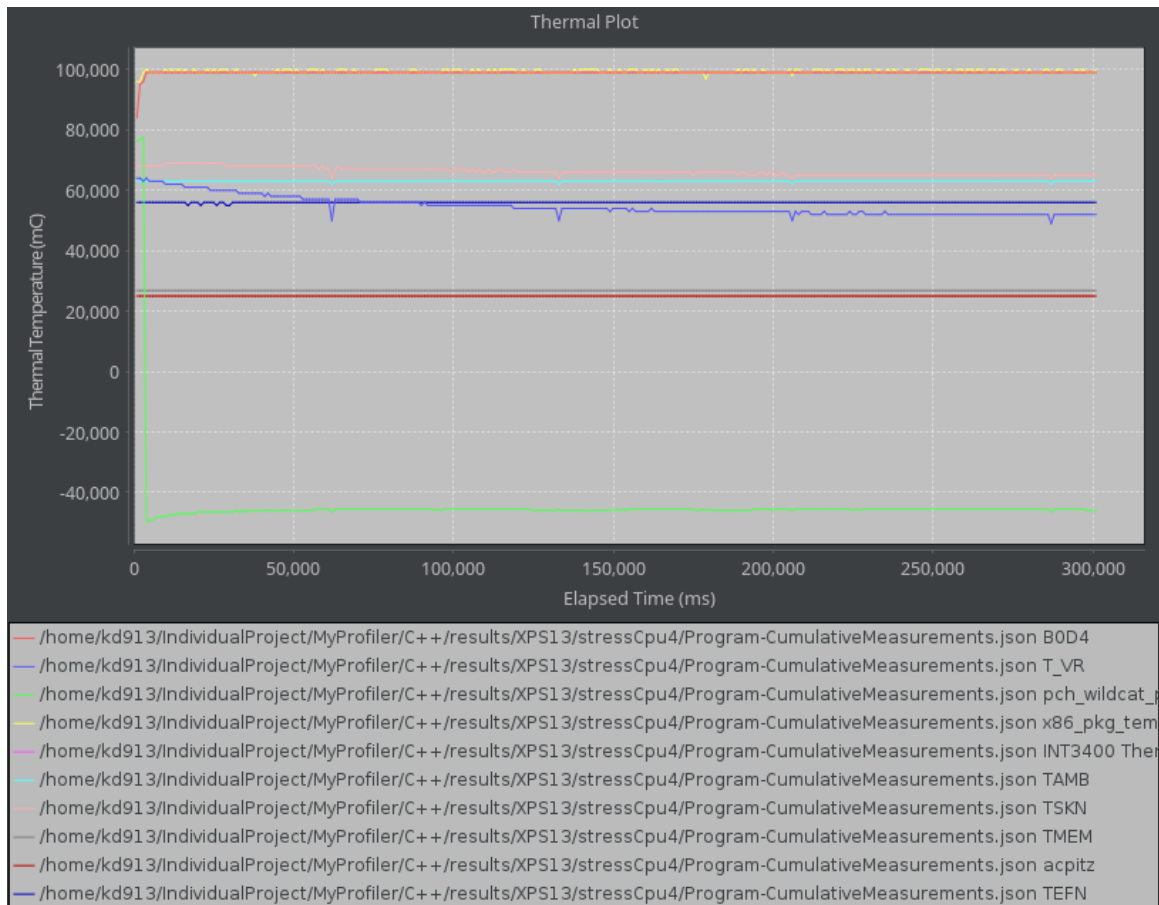
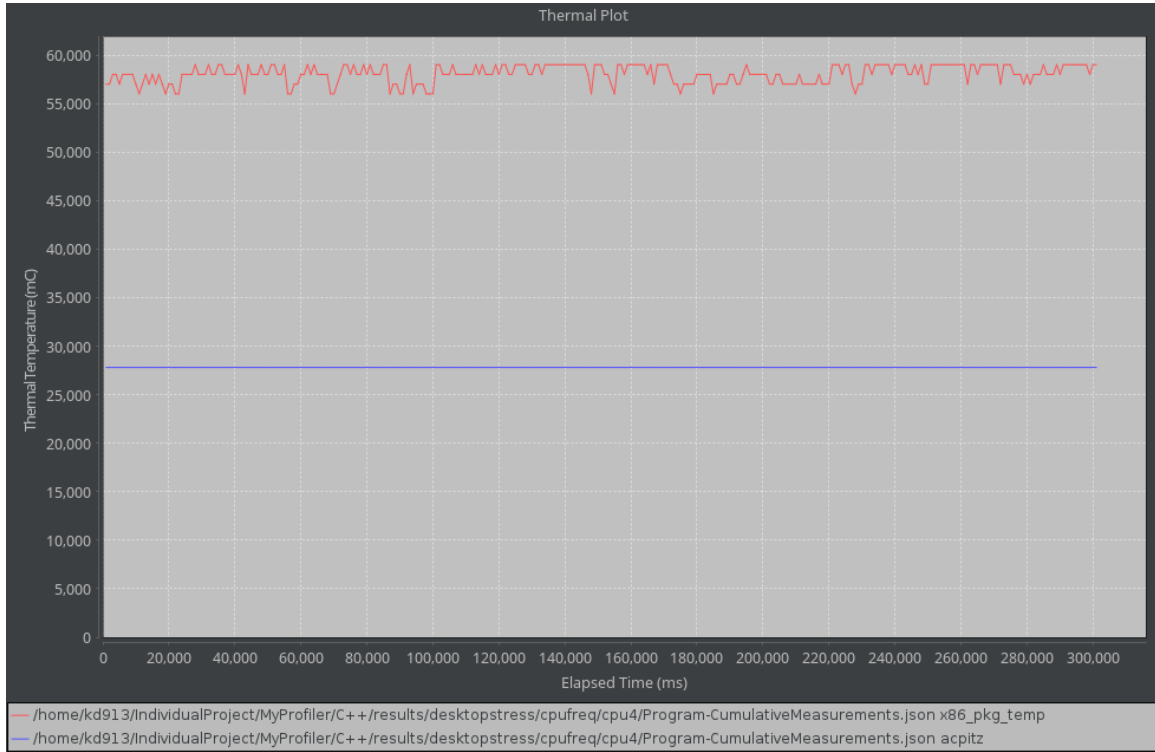


Figure 80: Thermal Data XPS 13 stress test, CPU 4



**Figure 81: Thermal Data Gaming Desktop stress test, CPU 4**

#### 4.4.11 IO Capturing Capability

This section explores the capability of PowerKap with some IO driven aspects. These as shown previously, can be costly from an energy perspective. The main idea of this test is to see the capability of the network and disk techniques that PowerKap uses.

##### 4.4.11.1 Ping Test

The first test performed, was a baseline test to prove the capability system and to demonstrate that it works. The main approach of doing this was to use the default Linux ping tool with the Google DNS address 8.8.8.8. Each ping was sent at a 30 second interval with a packet size of 1024 bytes. The results of this test can be seen in Figures 82.

The results from Figure 82 are useful as it demonstrates that PowerKap can accurately capture network data. In this case, it was able to accurately capture all data sent at a regulate interval with the correct packet size. Within the results there are interesting data points, namely the small spikes in network traffic that periodically occur. These points correspond to data points that periodically occur to maintain the virtualised network. This can be beneficial as it gives an accurate representation of how the deice would act in a real world test. However, such data is not directly relevant to the program.



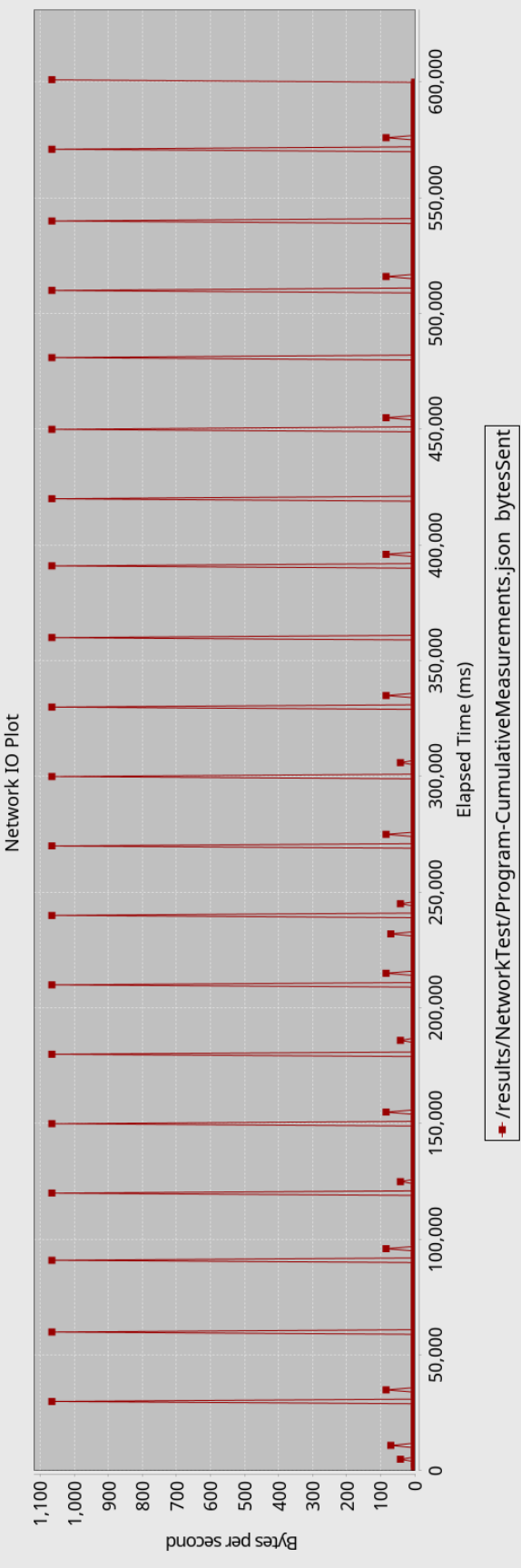


Figure 82: Dell XPS 13, Network Sent Bytes, ping test, 1024 packet size 30 second interval

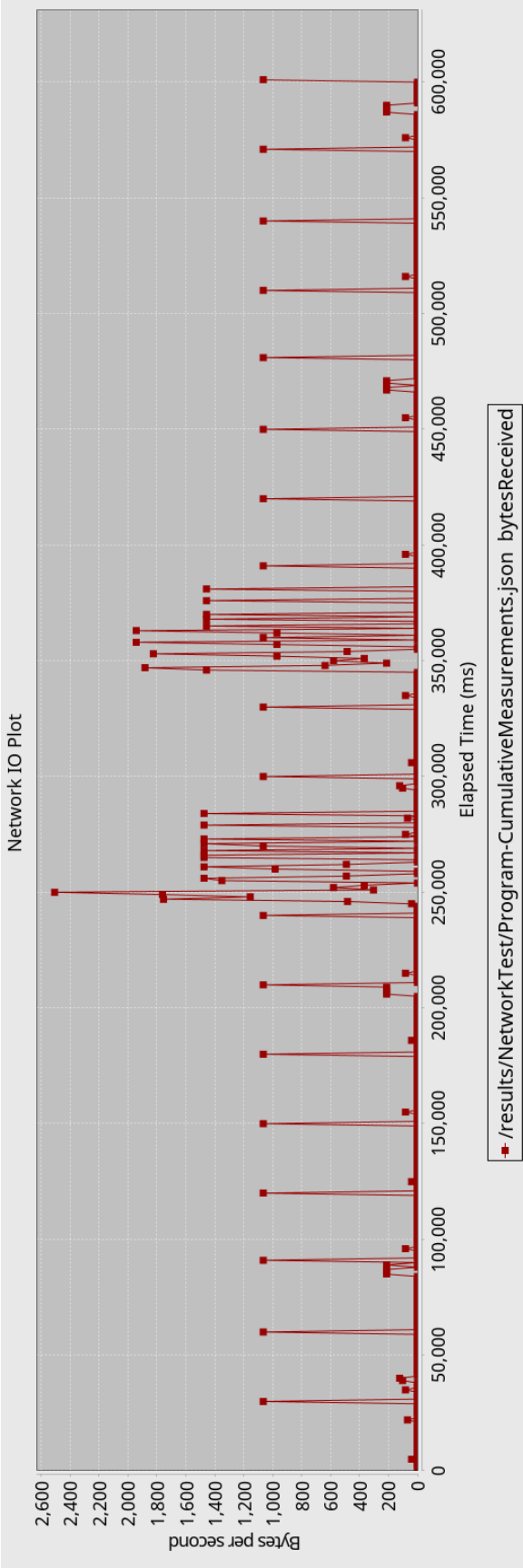


Figure 83: Dell XPS 13, Network Received Bytes, 1024 packet size 30 second interval

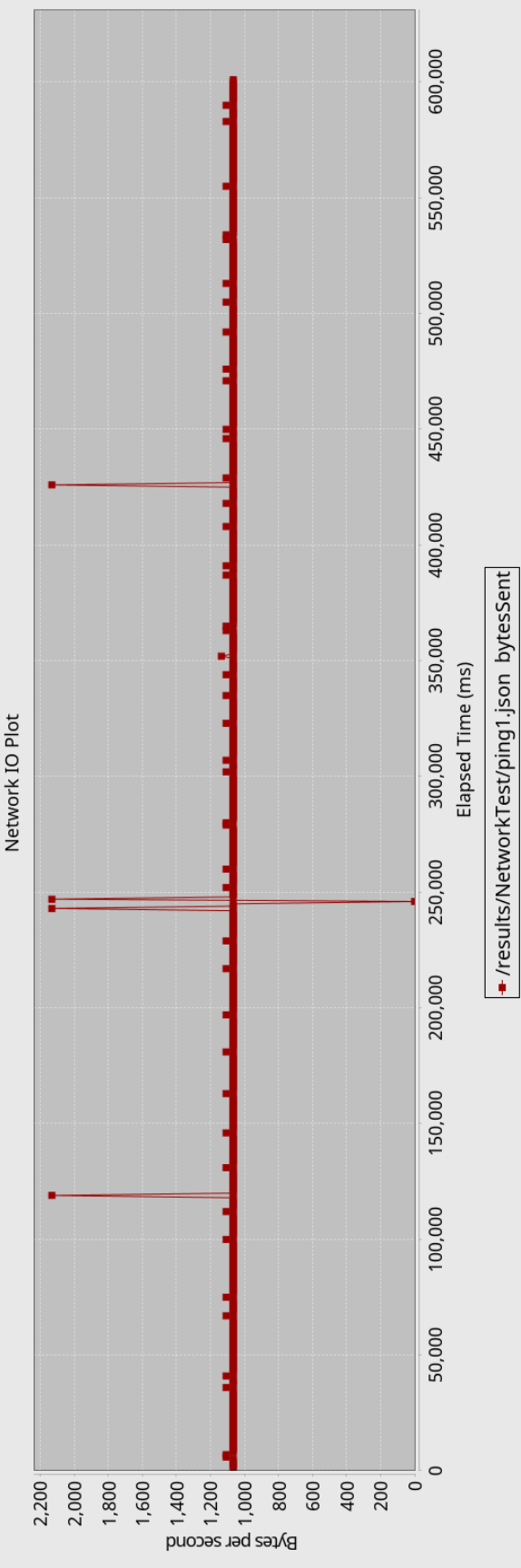


Figure 84: Dell XPS 13, Network Sent Bytes, ping test, 1024 packet size, 1 second interval

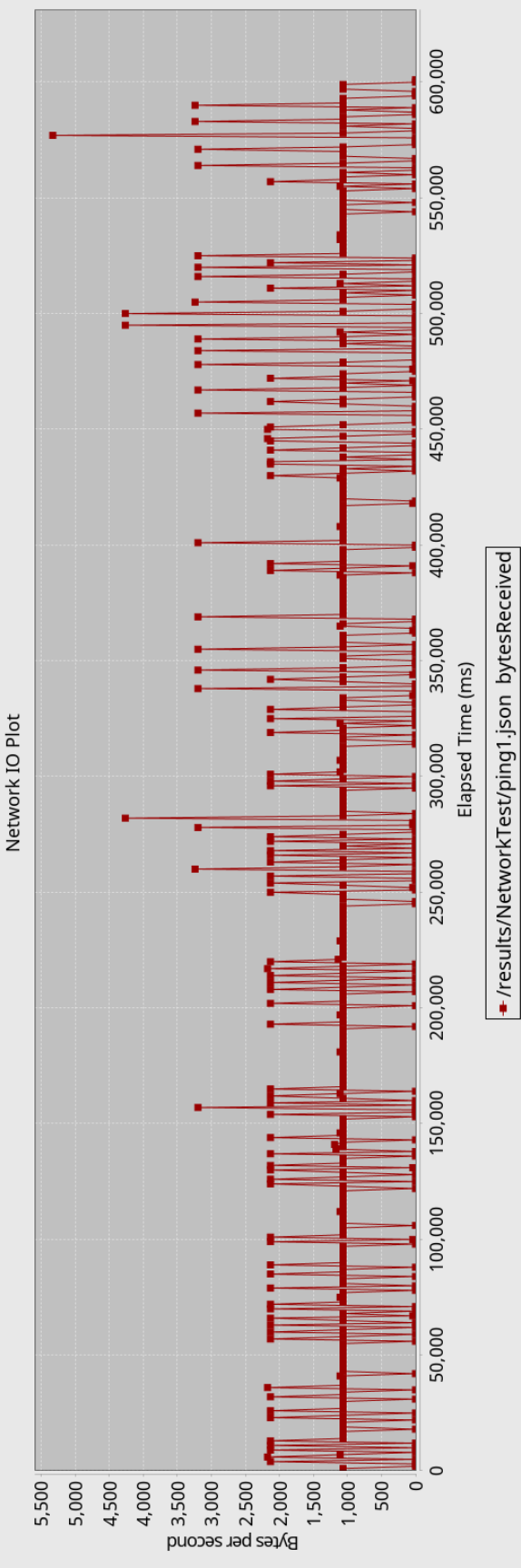


Figure 85: Dell XPS 13, Network Received Bytes, ping test, 1024 packet size, 1 second interval

The test in Figure 84, demonstrates the same ping test but at the default interval of one second. From this graph, we can see that the graph is mostly accurate with the exception of certain spikes. Even in this case, all the traffic is accounted for.

One of the main limitations of the technique used by this approach is the fact that the precision of the data depends heavily on the sampling rate. In this way, valuable information with respect to the times in which traffic is sent, is lost. To avoid this, it's recommended to repeat the test with a smaller sample size that just records network traffic.

#### 4.4.12 DiskIO Capturing Technique

One of the aims of the algorithm test presented previously was to eliminate potential caching issues. To do so, the project deliberately stored and read data from disk. Unfortunately, this test also demonstrated several flaws with the current approach of measuring disk data. In this case, we can observe this from the following figures.

From Figure 86, we can see several issues with respect to the data. In particular, each test case is designed to be about 5.5MB in size. However, from what we can observe from the graph, no data is actually being read beyond an initial 8800 bytes. The reason for this, is likely to do with space and time locality optimisation present in the OS and Java. In particular, as we write the data to disk, it is likely to still be present in various caches. For this reason, when we subsequently request data from the disk, it fetches the data from the cache anyway. This kind of approach highlights the main flaw in capturing data just purely by the bytes send and received to the storage controller. This approach also doesn't really work as the nature of caching can greatly affect subsequent executions. To alleviate this, the current approach is to simply account for the first execution. In the future, the program can be improved by dropping the cache before profiling.

```
echo 3 > /proc/sys/vm/drop_caches
```

This command sends a signal to the kernel to clear the page cache and free slab objects. Unfortunately, such a command at present requires root privileges which makes it unsuitable for PowerKap.

Another limitation gathered from Figure 87 demonstrates a limitation of the methodology of the program. At present, the method of capturing the information works by sampling precisely the disk data stats present in Procfs. Unfortunately, there is a limitation in capturing data at the end of the process. This is especially the case for data that is written just before program termination. In such cases, the data may be lost due to the Procfs directory being destroyed before PowerKap could capture this additional data. This can be seen from the results which show a file write of 38.6MB in comparison to the 42MB actually written to disk. For this reason, this approach is not particularly suitable for capturing disk data information.

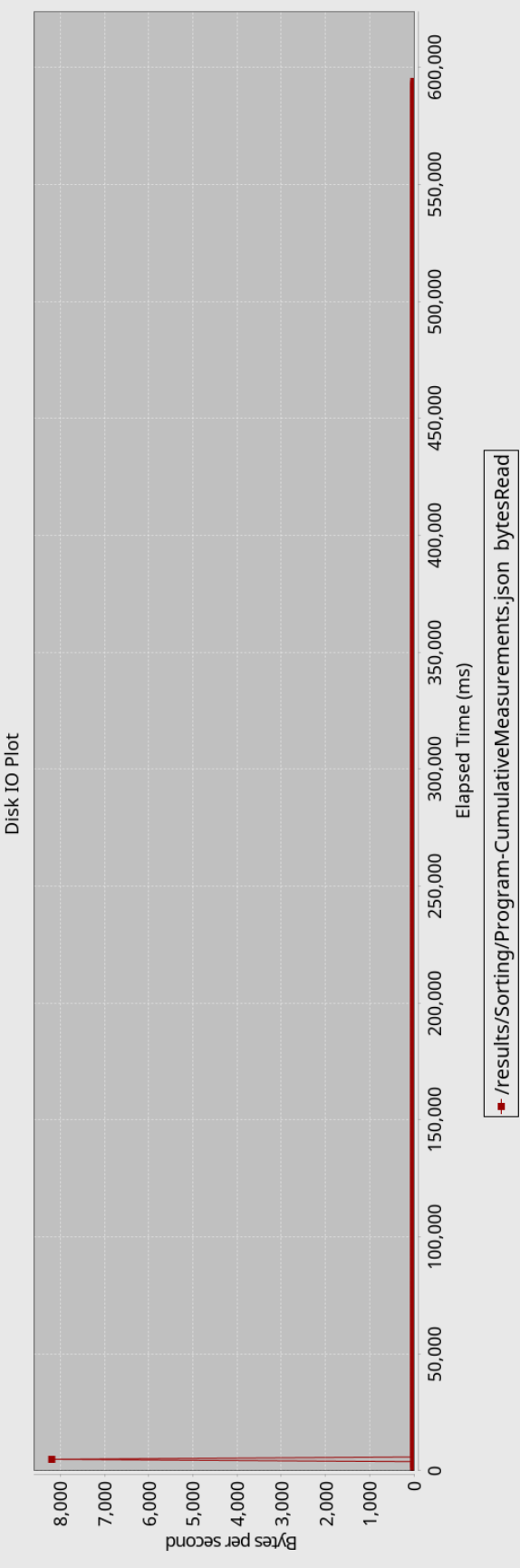


Figure 86: Dell XPS 13, Sorting Test, IO Read Graph

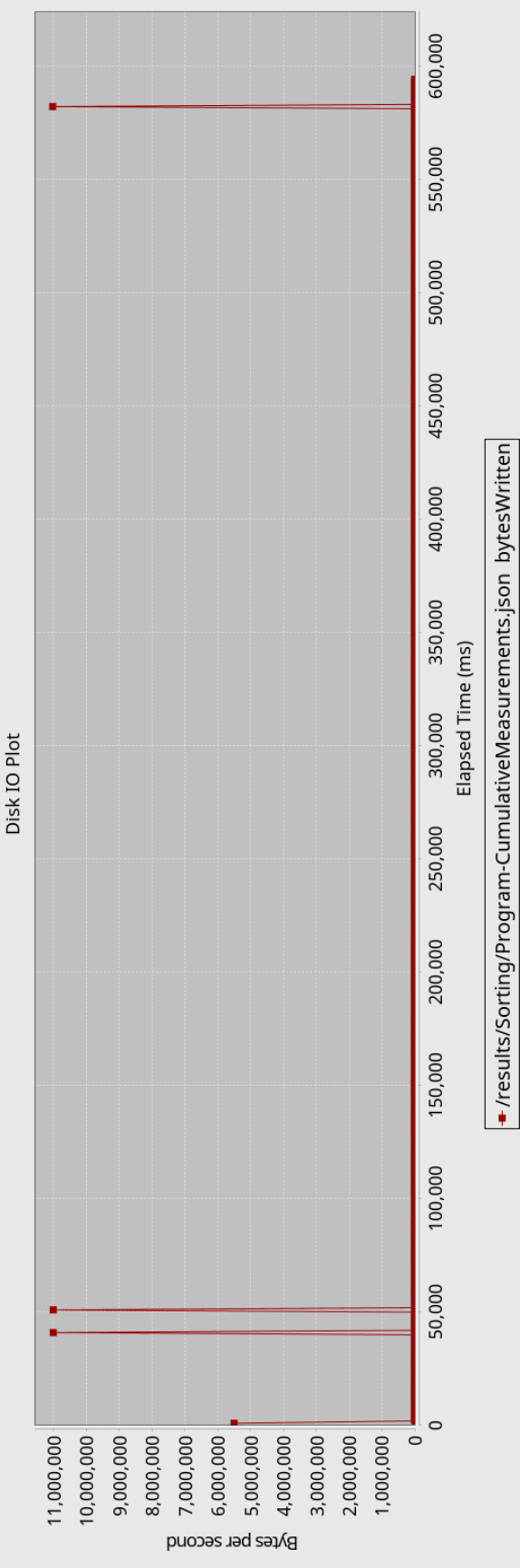


Figure 87: Dell XPS 13, Sorting Test, IO Write Graph

#### 4.4.13 Case Study: Browser Comparison

The last test used to evaluate the profiler was a test designed to compare the relative energy consumption when displaying a YouTube video. This was the test that originally inspired the creation of this program. For this test, the procedure was simply to open a default instance of Mozilla Firefox 53 and Google Chrome 59 that both open a 1080p YouTube video. In both cases, the video is using a VP9 encoding. This is important as this encoding is not hardware accelerated on both browsers and the hardware. The test each ran for 5 minutes, with the video active whilst the browser cut off mid way through the video. This test was particularly useful for demonstrating the robustness of PowerKap as Google Chrome makes use of advanced features such as namespaces and multi-process execution. The first aspect to evaluate is the total energy consumption according to the battery and CPU.

In Figures 88-91, we can see the energy results of executing the test. What is clear across all of the graphs is that Mozilla Firefox appears to be significantly more power efficient than Chrome for browsing YouTube. In this case, Google Chrome appears to consume more total energy, Uncore energy and DRAM energy. This is likely for a few reasons such as the multiprocess and sandboxing used in Chrome. These results are corroborated by the battery statistics.

In addition to this, PowerKap also captures various IO traffic. The results of this can be observed in Figures 92- 95.



Figure 88: Browser Comparison, Package Total Energy Graphs



Figure 89: Dell XPS 13, Browser Comparison Test, Uncore Energy Graphs



Figure 90: Dell XPS 13, Browser Comparison Test, DRAM Energy Graphs

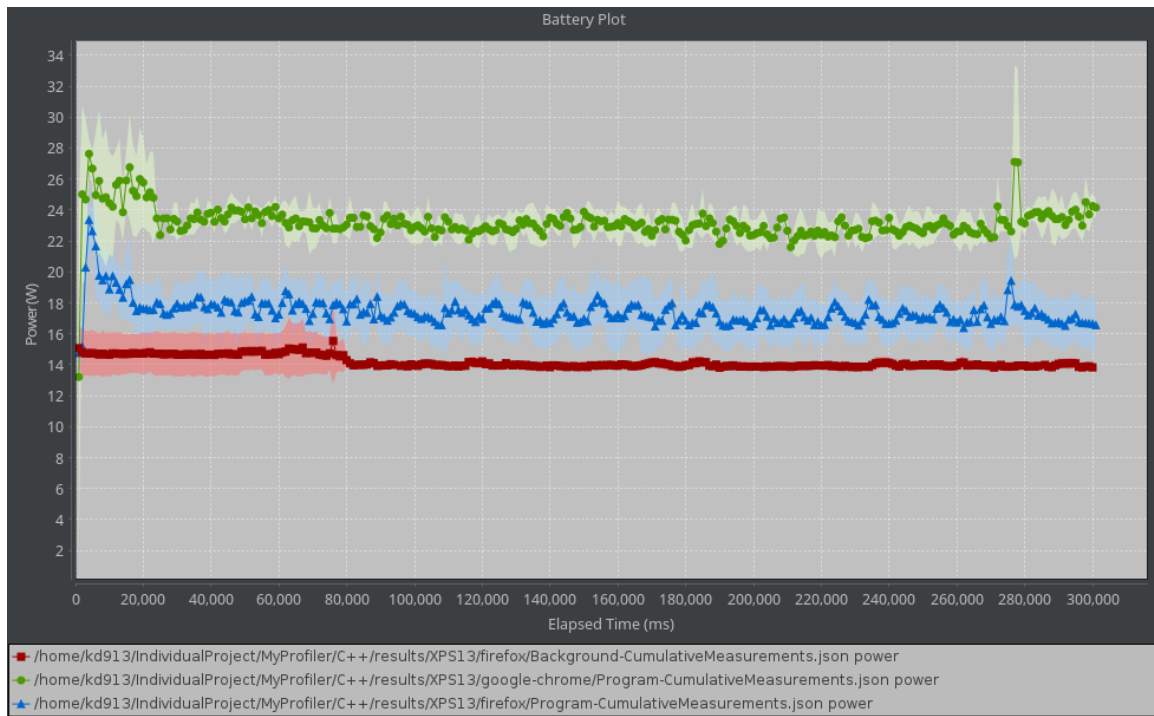


Figure 91: Dell XPS 13, Browser Comparison Test, Battery Graph

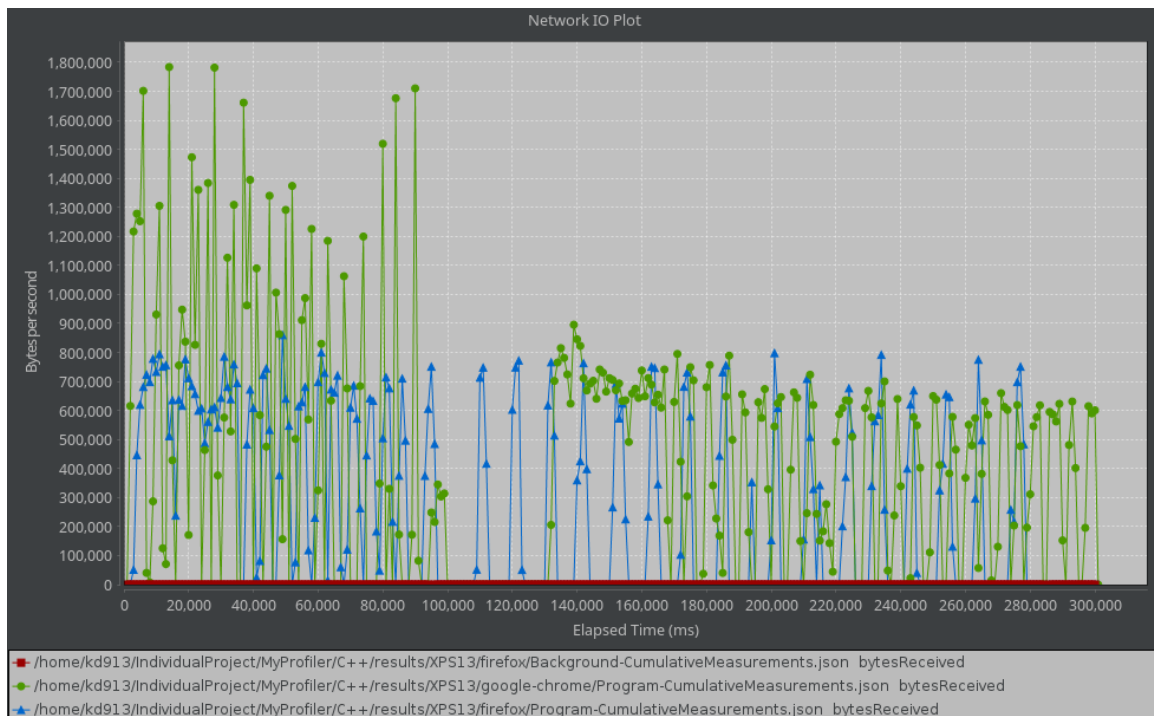
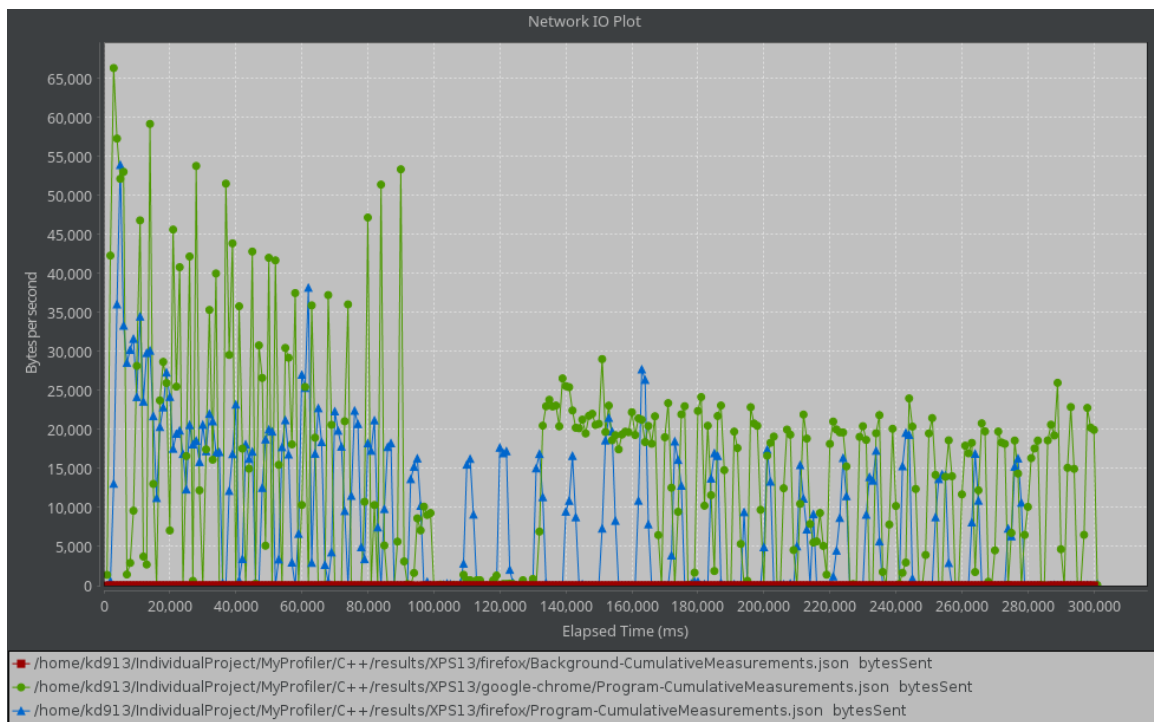
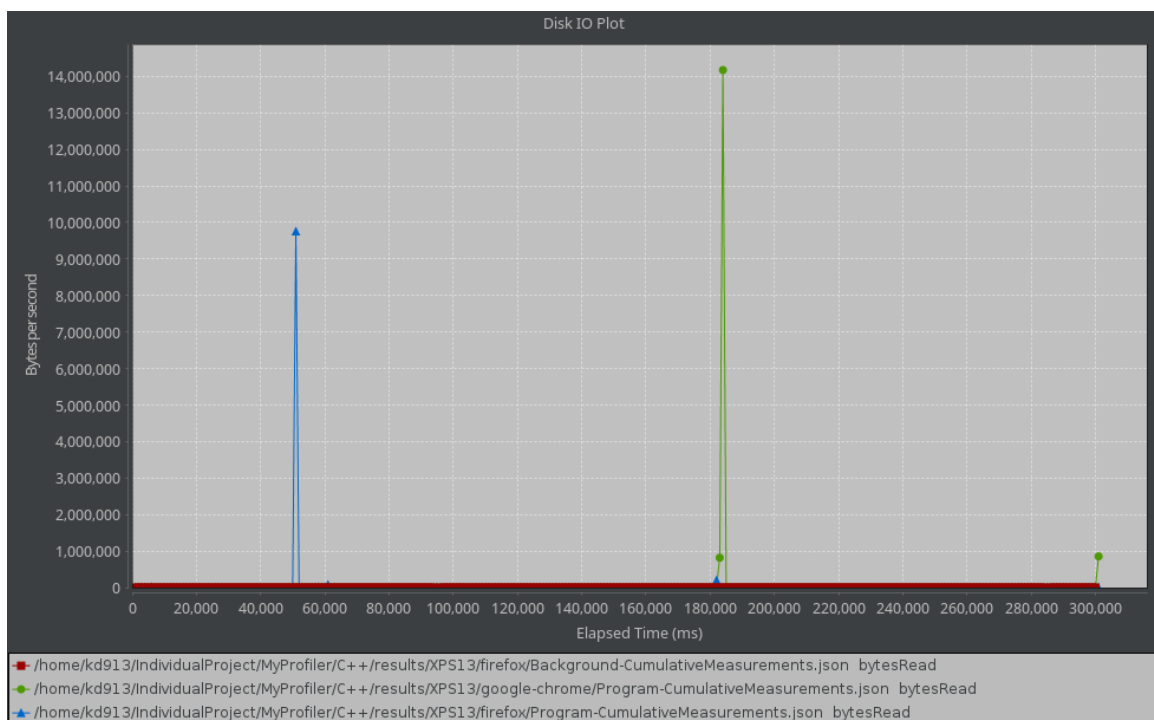


Figure 92: Dell XPS 13, Browser Comparison Test, Network Received Bytes

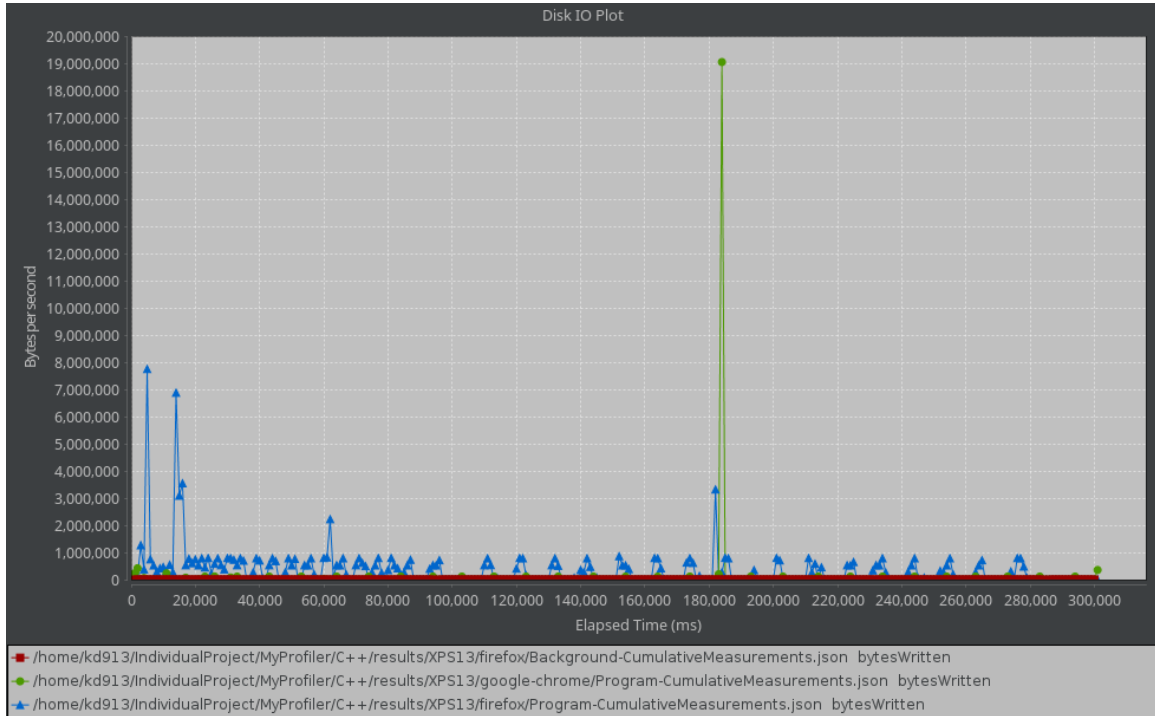


**Figure 93: Dell XPS 13, Browser Comparison Test, Network Sent Bytes**



**Figure 94: Dell XPS 13, Browser Comparison Test, IO Read Bytes**





**Figure 95: Dell XPS 13, Browser Comparison Test, IO Written Bytes**

#### 4.4.14 LJE A

The other important feature is the ability to attribute the measurements gathered back to the program. This is an important step as the ultimate goal for this tool is to enable developers to be aware of the energy cost of their programs and functions.

##### 4.4.14.1 Graphing Module

The first point to note is that all the measurements gathered and profiled previously were generated and saved by LJE A. In each case, the module managed to handle a large volume of data reliably.

##### 4.4.14.2 Energy Trace

Another important feature of the project was the stacktrace, to see if the energy results gathered could be attributed back to the source code. This approach was tested with the algorithm benchmark created earlier.

In Figure 96, we can observe the result of the stacktrace along with various execution points. Within this image, we can observe some limitations with respect to the program. The first can be the fact that elements one and two are missing from the stack trace. The reason for this, is because certain events can occur much faster than the sampling rate of PowerKap. This can be problematic when attempting to attribute energy as simply the information is lost. One such solution to this problem is to choose a smaller sampling interval with PowerKap. This as shown previously can affect the energy measurements gathered. For this reason, this project is unable to gather fine detailed information that occur in a

short interval.

That is not to say that the approach does not necessarily work for all cases. In this case, one possible use is to identify specific code or functions of interest. This can be seen from the graph in Figure 97. In this case, the stack trace technique was accurate in pinpointing the points in which the function sorting algorithms occur.

This test, is only able to work on perfectly synchronous code. In this case, it doesn't rely on any external factors such as an external database, networking, or even hard disks. This technique is unlikely to work reliably for such codebases as this technique requires averaging the time stamps across multiple runs to find an accurate marker point. Alternatively, this approach may only work when evaluating a program over a single run.

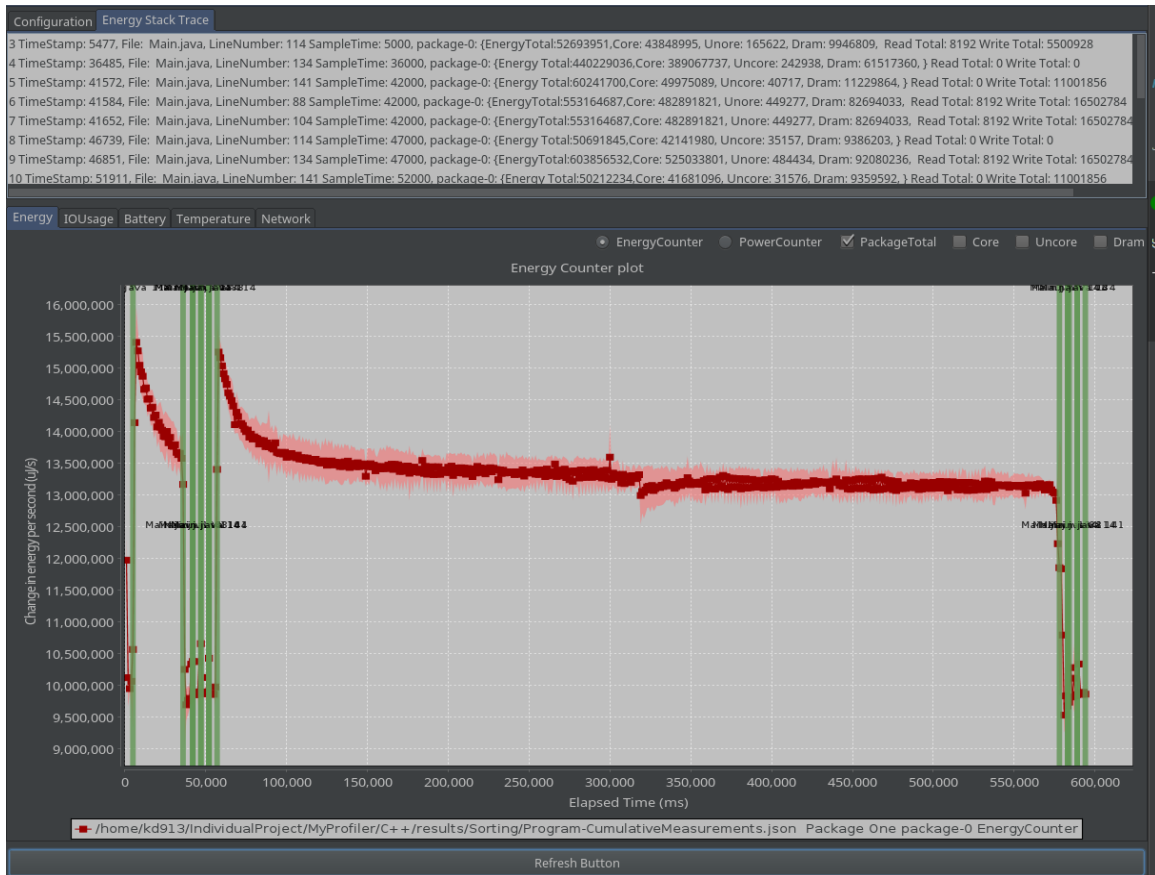
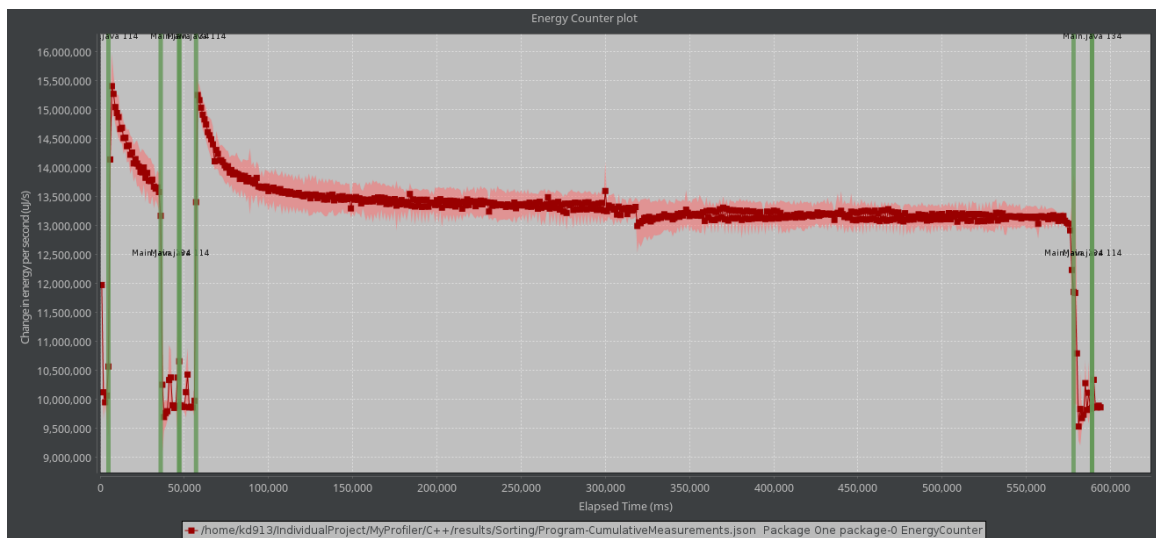


Figure 96: Algorithms StackTrace with all stackpoint markers enabled



**Figure 97: Algorithms StackTrace with all stackpoint markers enabled**

## 5 Conclusion

To conclude, we have demonstrated the capability of measuring the energy consumption of a given program using physical measurements on Linux. All of this is possible within user space. This was achieved by gathering various physical data counters along with making use of advances in the Linux Kernel. In addition, this project integrates the results of these energy metrics into a common integrated development environment (IDE). Through this tool, it is now possible to identify potential problematic power behaviour such as busy loops. It is also now possible for developers to see the actual energy consumption of their program and make further analysis. We have shown this in our case study of BigBuckBunny video player, where we have demonstrated that the two threaded program was more efficient than the four threaded test. This kind of result allows developers to better adjust their code to maximise energy efficiency for a target platform.

In addition, this project has explored the difference in energy consumption across various hardware platforms. In particular, we evaluated using common benchmarks to see if such techniques can be used to create formal contracts on energy consumption. Our benchmarks have shown that even across identical hardware and operating systems, there can be differences in the energy profile. We also found that different platforms have different energy requirements especially with respect to desktop versus laptop energy consumption. As such, it may be inaccurate to assume that the same energy preserving behaviour applies across hardware. This is even the case on identical hardware where uncontrolled factors can result in different energy profiles. In particular, it may be a challenge to guarantee a generalised energy consumption for a program across multiple platforms and hardware.

In this project, we have shown the capability of the program in distinguishing real world programs energy profiles. In this case, we have tested two commercial browsers and came to the conclusion that Firefox is significantly more energy efficient on this particular Dell XPS 13 9343 laptop when watching YouTube. This conclusion may not apply universally as we have shown that in some of our hardware analysis that not all platforms demonstrate the same power curves.

From PowerKap, we can also conclude the effectiveness of the new virtual adapter technique. In our tests, we have demonstrated that it is capable of accurately measuring the network usage of a given program.

## 6 Recommendations for Future Work

In our evaluation of PowerKap, we have demonstrated many flaws in the current issues of using physical counters especially in capturing disk and temperature information. This section expands upon this, by exploring potential approaches to improve PowerKap to mitigate against these issues.

### 6.1 PowerKap

#### 6.1.1 Expanding the interfaces

It would be useful to expand PowerKap to become a more general profiler. Within the background section, we have described multiple forms of interfaces that are each useful for a specific hardware configuration and purpose. It would be particularly useful in this respect to expand to gain data from say the Perf or PAPI interface. These structures already come with additional information which itself can be useful for program profiling. This can include various counters and data such as page cache miss information and the specific times in which a thread is active or sleeping. This approach would come with a trade-off in that they require root and additional configuration. It would however, be useful to gather these metrics if they are available.

In addition to this, there are scopes to expand the energy interfaces gathered such as the GPU power and energy consumption. This can be quite complex as currently the GPU market on Linux is quite fragmented between proprietary and open source drivers. Currently the only the methodology of interacting with Nvidia graphics power data is through PAPI. Each mechanism often comes with its additional configuration and processing.

#### 6.1.2 Sysfs/Procfs/Linux Interfaces

Another issue with respect to power profilers on Linux is the inconsistency across Kernel versions. This can be particularly challenging as in certain versions, entire directories may shift. This can be problematic for estimating energy and would likely break the profiler. Historically, this was the case as the entire Battery and ACPI information were present in the Procfs directory. In this regard, it would be useful to have a unified systemcall or interface that gathered most of the energy metrics. This would save energy when profiling as currently various techniques are necessary to interact with each attribute separately. At present, the GNU utils do provide some interfaces for resources, this is mainly through the Resources.h header [68]. This interface gives a particularly useful set of metrics relating to the resource consumption of a process. It would be particularly useful to have a similar interface available for energy counter metrics.

#### 6.1.3 Asynchronous computing

Another major challenge which wasn't dealt with particularly effectively is the problem with asynchronous devices in computers. This is particularly problematic in scenarios which rely on an external factor such as a network response from a server. In these contexts, PowerKap is not particularly effective past the first run. In order to improve this situation, it is recommended to attempt the approach used by eProf. In this case, it would be useful to create an energy model for these asynchronous network and disk attributes. The idea being

that whenever a program requests a certain amount of data from a network card, or hard disk, this model is used to estimate the energy consumption of this request.

The main reason this approach wasn't currently adopted is mainly due to the fact that the energy estimation for generating this model cannot currently be done from software alone. This is because additional external power meters are necessary to gain the physical energy values required for generating the energy model.

#### **6.1.4 Machine Learning and Model Generating**

At present, the main technique of estimating the energy consumption of a program is quite simplistic. In this case, the code currently just maps the execution with the physical energy counter metrics. In the future, it would be particularly useful to perhaps use an intermediate form of the code such as JavaByteCode. By analysing patterns and blocks, PowerKap may be able to generate a more generalised database with respect to energy consumption. This kind of approach would be useful as at present, measuring the energy consumption by execution time can take a significant amount of time. For example, each benchmark used in the evaluation section took significant amount of time to generate, in particular in the order of minutes and hours. This is unacceptable for most developers who would like to know the energy consumption within seconds to minutes. This can only be done from analysing and attributing energy properties with source code.

This model based approach may also be useful for generalising the energy consumption for any input. At present, PowerKap only works on test cases and scenarios where the input and program is already constructed. In the future, it would be useful to be able to make a general assumption or estimate of the energy consumption based on any input.

#### **6.1.5 Handling Thermal Spikes**

One major issue at the moment when evaluating programs is the issue of understanding what aspect of the device is responsible for potential variations during the execution. This can be observed in our evaluation graphs by certain areas with large standard deviation. With such small sample sizes used during repeated execution, it is currently not possible to eliminate energy points as anomalous. For this reason, the current method of gathering energy information is susceptible to large spikes. These can be caused for various reasons including temperature or hardware degradation. In the future, it would be useful to be able to trace the origin of these spikes and perhaps filter them out. It would be useful to design PowerKap to detect the general trends across a platform and to filter these results out.

### **6.2 LJEA**

#### **6.2.1 Introducing code suggestions**

In the context of AEON, the plugin is capable of discovering poor energy behaviour such as the use of improper wake locks. This capability can be extended into LJEA now that we can measure the energy profile of a given piece of code. In this case, the tool could be used to test various input and conditions, such as different data structures or threads. Using the information gathered, we can tune the program to optimise for a particular platform. For example, in the Mplayer evaluation benchmark, we were able to deduce that the optimal

energy consumption for MPlayer was to use two threads rather than four. If this capability was available automatically, this product could greatly improve energy efficiency of software.

### **6.2.2 Expanding to other IDEs and Languages**

Part of the reason this project has split the power profiler and the plugin was so that it could be used for different languages. As such, this project is capable of being extended to other IDEs such as kDevelop or Eclipse.

## References

- [1] Erol Gelenbe and Yves Caseau. The Impact of Information Technology on Energy Consumption and Carbon Emissions. *Ubiquity*, 2015(June):1:1–1:15, June 2015. ISSN 1530-2180. doi: 10.1145/2755977. URL <http://doi.acm.org/10.1145/2755977>.
- [2] Alison Coleman. How much does it cost to keep your computer online? (Lots, it turns out), April 2017.  
[Article], Available: <http://www.telegraph.co.uk/business/energy-efficiency/cost-keeping-computer-online/>, Accessed on 31/05/2017.
- [3] Anders S. G. Andrae and Tomas Edler. On global electricity usage of communication technology: Trends to 2030. *Challenges*, 6(1):117–157, 2015. ISSN 2078-1547. doi: 10.3390/challe6010117. URL <http://www.mdpi.com/2078-1547/6/1/117>.
- [4] Chris Edwards. Lack of Software support marks the low power scorecard at DAC. July 2011.  
[Article] Electronics Weekly 15-21 July 2011—No. 2072, Accessed on 05/12/2016.
- [5] POWERTOP. Powertop User’s Guide. Technical report, 2015.  
[Online], Available: <https://01.org/powertop/overview>, Accessed on 27/12/2016.
- [6] TURBOSTAT. Turbostat Man Page. Technical report, 2010.  
[Online], Available: <http://manpages.ubuntu.com/manpages/precise/man8/turbostat.8.html>, Accessed on 27/12/2016.
- [7] POWERDEF. Oxford Power Definition.  
[Online], Available: <https://en.oxforddictionaries.com/definition/power>, Accessed on 09/01/2017.
- [8] chrisdavidmills wcosta, nnethercote. Power Profiling Overview, 2015.  
[Online] Available: [https://developer.mozilla.org/en-US/docs/Mozilla/Performance/Power\\_profiling\\_overview](https://developer.mozilla.org/en-US/docs/Mozilla/Performance/Power_profiling_overview), Accessed on 08/01/2017.
- [9] ACPI. Advanced Configuration and Power Interface Specification, Vol. 6.0a, Chapters 3 and 8. Technical report, ACPI, 2016.  
[Online], Available: <http://www.acpi.info/>, Accessed on 30/12/2016.
- [10] Dr.Bob Steigerwald and Abhishek Agrawal. Developing Green Software.  
[Report]. Available: [https://software.intel.com/sites/default/files/developing\\_green\\_software.pdf](https://software.intel.com/sites/default/files/developing_green_software.pdf), Accessed on 06/01/2017, 2010.
- [11] ASPM. ECN\_ASPM\_OPTIONALITY, Chapter 5.4.1. Technical report, 2013.  
[Online], Available: [https://pcisig.com/sites/default/files/specification\\_documents/ECN\\_ASPM\\_Optionality\\_2009-08-20.pdf](https://pcisig.com/sites/default/files/specification_documents/ECN_ASPM_Optionality_2009-08-20.pdf), Accessed on 30/12/2016.
- [12] IAORM. Intel®64 and ia-32 Architectures Optimization Reference Manual, Chapter 13.5. Technical report, 2016.  
[Online], Available: <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf>, Accessed on 27/12/2016.
- [13] Dr John Bell. Thread Definition.  
[Online], Available: [https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4\\_Threads.html](https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html), Accessed on 09/01/2017.
- [14] Cinebench. Maxon Cinebench benchmark tool, 2017.  
[Online], Available: <https://www.maxon.net/en/products/cinebench/>, Accessed on 27/12/2016.
- [15] GCC. GCC optimization flags, 2017.  
[Online], Available: <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options>.



- html, Accessed on 07/01/2017.
- [16] SIMD. Kernel Org SIMD Examples, 2017.  
[Online], Available: <https://www.kernel.org/pub/linux/kernel/people/geoff/cell/ps3-linux-docs/CellProgrammingTutorial/BasicsOfSIMDProgramming.html>, Accessed on 07/01/2017.
  - [17] WINDOWSTIMER. Timers, Timer Resolution, and Development of Efficient Code. Technical report, 2010.  
[Online], Available: <http://download.microsoft.com/download/3/0/2/3027D574-C433-412A-A8B6-5E0A75D5B237/Timer-Resolution.docx>, Accessed on 08/01/2017.
  - [18] Robert Love. *Linux Kernel Development (2Nd Edition)* (Novell Press). Novell Press, 2005. ISBN 0672327201. Accessed on 11/01/2017.
  - [19] JIFFY. Linux Jiffy and Tick Rate, 2013.  
[Online], Available: [http://elinux.org/Kernel\\_Timer\\_Systems](http://elinux.org/Kernel_Timer_Systems), Accessed on 07/01/2017.
  - [20] Jonathan Corbet. Timer Slack. 2010.  
[Online], Available: <https://lwn.net/Articles/369549/>, Accessed on 07/01/2017.
  - [21] Belinda Liviero. Measuring application power consumption on the Linux\* Operating Systems, 2013.  
[Online], Available: <https://software.intel.com/en-us/blogs/2013/06/18measuring-application-power-consumption-on-linux-operating-system>, Accessed on 27/12/2016.
  - [22] Srinivas Pandravadu. Intel Running Average Power Limit, 2014.  
[Online], Available: <https://01.org/blogs/2014/running-average-power-limit-%E2%80%93-rapl>, Accessed on 16/12/2016.
  - [23] Andreas Herrmann. AMD linux hardware power driver (fam15h\_power).  
[Online], Available: [https://www.kernel.org/doc/Documentation/hwmon/fam15h\\_power](https://www.kernel.org/doc/Documentation/hwmon/fam15h_power), Accessed on 14/01/2017.
  - [24] ENTRA. Whole-Systems Energy Transparency Project.  
[Report], Available: <http://entraproject.eu/>, Accessed on 05/01/2017, 2012-2015.
  - [25] ENTRA. Common Assertion Language, Deliverable Number D2.1. Technical report, University of Bristol, 2013.  
[Report], Available: [http://entraproject.eu/wp-content/uploads/2014/03/deliv\\_2.1\\_final.pdf](http://entraproject.eu/wp-content/uploads/2014/03/deliv_2.1_final.pdf), Accessed on 30/12/2016.
  - [26] ENTRA. Energy Optimization: Basic Static Techniques, Deliverable Number D4.3. Technical report, Roskilde University, 2014.  
[Report], Available: [http://entraproject.eu/wp-content/uploads/2014/09/deliv\\_4.1.pdf](http://entraproject.eu/wp-content/uploads/2014/09/deliv_4.1.pdf), Accessed on 30/12/2016.
  - [27] Intel. How to Use Loop Blocking to Optimize Memory Use on 32-Bit Intel® Architecture. December 2013.  
[Online], Available: <https://software.intel.com/en-us/articles/how-to-use-loop-blocking-to-optimize-memory-use-on-32-bit-intel-architecture>, Accessed on 09/01/2017.
  - [28] Henry Massalin. Superoptimizer: A Look at the Smallest Program. *SIGPLAN Not.*, 22(10):122–126, October 1987. ISSN 0362-1340. doi: 10.1145/36205.36194. URL <http://doi.acm.org/10.1145/36205.36194>.
  - [29] Zorana Banković. Study of Possible Static Power Reduction due to Temperature Hot Spot Reduction provided by Uniform Register Utilization, Attachment D4.1.1.

- [Online], Available: [http://entraproject.eu/wp-content/uploads/2014/09/deliv\\_4.1.pdf](http://entraproject.eu/wp-content/uploads/2014/09/deliv_4.1.pdf), Accessed on 30/12/2016, 2014.
- [30] S. Schubert, D. Kostic, W. Zwaenepoel, and K. G. Shin. Profiling software for energy consumption. In *2012 IEEE International Conference on Green Computing and Communications*, pages 515–522, Nov 2012. doi: 10.1109/GreenCom.2012.86.
- [31] Chiyong Seo, Sam Malek, and Nenad Medvidovic. An energy consumption framework for distributed java-based systems. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE '07*, pages 421–424, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi: 10.1145/1321631.1321699. URL <http://doi.acm.org/10.1145/1321631.1321699>.
- [32] Aman Kansal and Feng Zhao. Fine-grained energy profiling for power-aware application design. *SIGMETRICS Perform. Eval. Rev.*, 36(2):26–31, August 2008. ISSN 0163-5999. doi: 10.1145/1453175.1453180. URL <http://doi.acm.org/10.1145/1453175.1453180>.
- [33] Adel Nouredine. *Towards a Better Understanding of the Energy Consumption of Software Systems*. Theses, Université des Sciences et Technologie de Lille - Lille I, March 2014. URL <https://tel.archives-ouvertes.fr/tel-00961346>.
- [34] David Gonzalez. AEON (Automated Android Energy Optimization), 2016.  
[Online], Available: <https://plugins.jetbrains.com/idea/plugin/7444-aeon-automated-android-energy-optimization->, Accessed on 12/01/2017.
- [35] Qualcomm. Qualcomm’s Trepn profiler, 2016.  
[Online], Available: <https://developer.qualcomm.com/software/trepn-power-profiler>, Accessed on 12/01/2017.
- [36] Microsoft. Analyse Energy in Store Apps, 2015.  
[Online], Available: <https://msdn.microsoft.com/en-us/library/dn263062.aspx>, Accessed on 04/05/2017.
- [37] VSDEPRECATED. Energy Consumption Counter option in Visual Studio 2017, 2017.  
[Online], Available: <https://social.msdn.microsoft.com/Forums/vstudio/en-US/3260a475-eb9e-4c5f-9bf5-14219cdbbc81/energy-consumption-counter-option-in-visual-studio-profiler-2017?forum=vsdebug>, Accessed on 19/05/2017.
- [38] INTELMSR. Intel 64 and IA-32 Architectures software developer manual Vol. 3, Chapter 14.9. Technical report, 2016.  
[Online], Available: <http://download.intel.com/products/processor/manual/325384.pdf>, Accessed on 04/05/2017.
- [39] LINUXMSR. Linux MSR man page.  
[Online], Available: <http://man7.org/linux/man-pages/man4/msr.4.html>, Accessed on 19/05/2017.
- [40] Anand Lil Shilpi. Nehalem: the unwritten chapters.  
[Article], Available: <http://www.anandtech.com/show/2663>, Accessed on 13/06/2017, 2008.
- [41] PERF. Perf Linux System.  
[Online], Available: [https://perf.wiki.kernel.org/index.php/Main\\_Page](https://perf.wiki.kernel.org/index.php/Main_Page), Accessed on 19/05/2017.
- [42] Vince Weaver. Reading RAPL energy measurements from Linux.  
[Online], Available: <http://web.eece.maine.edu/~vweaver/projects/rapl/>, Accessed on 19/05/2017.

- [43] PAPI. PAPI homepage.  
[Online], Available: <http://icl.cs.utk.edu/papi/index.html>, Accessed on 19/05/2017.
- [44] HWMON. Linux HWMON interface.  
[Online], Available: <https://www.kernel.org/doc/Documentation/hwmon/sysfs-interface>, Accessed on 19/05/2017.
- [45] POWERCAP. Intel Powercap Interface.  
[Online], Available: <https://www.kernel.org/doc/Documentation/power/powercap/powercap.txt>, Accessed on 19/05/2017.
- [46] Spiral Team. PowerAPI git page.  
[Online], Available: <https://github.com/pwrapi/pwrapi-ref>, Accessed on 31/05/2017.
- [47] CPUCHART. Various Energy Interface support for Intel and AMD platforms.  
[Online], Available: [http://web.eece.maine.edu/~vweaver/projects/rapl/rapl\\_support.html](http://web.eece.maine.edu/~vweaver/projects/rapl/rapl_support.html), Accessed on 19/05/2017.
- [48] Patrick Mochel. The Sysfs FileSystem.  
[Online], Available: <http://milbret.anydns.info/pub/linux/kernel/people/mochel/doc/papers/ols-2005/mochel.pdf>, Accessed on 19/05/2017, 2005.
- [49] Pablo Neira-Ayuso, Rafael M. Gasca, and Laurent Lefevre. Communicating between the kernel and user-space in Linux using Netlink sockets. *Software: Practice and Experience*, 40(9):797–810, 2010. ISSN 1097-024X. doi: 10.1002/spe.981. URL <http://dx.doi.org/10.1002/spe.981>.
- [50] NetlinkACPI. Kernel Driver Power Meter.  
[Online], Available: [https://www.kernel.org/doc/Documentation/hwmon/acpi\\_power\\_meter](https://www.kernel.org/doc/Documentation/hwmon/acpi_power_meter), Accessed on 19/05/2017.
- [51] aj504. RootSudo.  
[Online], Available: [https://help.ubuntu.com/community/RootSudo#Graphical\\_sudo](https://help.ubuntu.com/community/RootSudo#Graphical_sudo), Accessed on 05/06/2017.
- [52] Brendan Gregg. strace wow much syscall (a comparison of the impacts of strace).  
[Online], Available: <http://www.brendangregg.com/blog/2014-05-11/strace-wow-much-syscall.html>, Accessed on 05/06/2017.
- [53] raboof. NetHogs.  
[Online], Available: <https://github.com/raboof/nethogs>, Accessed on 01/06/2017, 2017.
- [54] rhallok. AMD Ryzen Community Update, 2017.  
[Online], Available: <https://community.amd.com/community/gaming/blog/2017/03/13/amd-ryzen-community-update>, Accessed on 24/05/2017.
- [55] nlohmann. Nlohmann’s JSON for C++, 2016.  
[Online], Available: <https://github.com/nlohmann/json#serialization--deserialization>, Accessed on 15/06/2017.
- [56] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*, volume 2. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. ISBN 0-201-89684-2.
- [57] JetBrains. JetBrains IntelliJ Idea PowerSaveMode, 2016. [Online], Available: <https://www.jetbrains.com/help/idea/2016.3/status-bar.html>, Accessed on 10/01/2017.
- [58] David Gilbert. JFreeChart, 2016.  
[Online], Available: <http://www.jfree.org/index.html>, Accessed on 15/06/2017.
- [59] IMDEA Software Institute XMOS Limited Roskilde University, University of Bristol.

- ENTRA Benchmark Suites, Deliverable Number 5.1, 2013.  
 [Report], Available: [http://entraproject.eu/wp-content/uploads/2014/03/deliv\\_5.1\\_final.pdf](http://entraproject.eu/wp-content/uploads/2014/03/deliv_5.1_final.pdf), Accessed on 05/06/2017.
- [60] Dominik Browski. Linux Kernel CPUFreq governor.  
 [Online], Available: <https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt>, Accessed on 13/06/2017.
- [61] Alex Campbell. Kaby Lake is unleashed with Linux kernel 4.10, 2017.  
 [Article], Available: <http://www.pcworld.com/article/3173618/linux/kaby-lake-is-unleashed-with-kernel-410.html>, Accessed on 13/06/2017.
- [62] The MPlayer Project. Mplayer, 2000.  
 [Online], Available: <http://www.mplayerhq.hu/design7/news.html>, Accessed on 15/06/2017.
- [63] Blender Foundation. Big buck bunny movie, 2008.  
 [Online], Available: <https://peach.blender.org/>, Accessed on 15/06/2017.
- [64] OPENBENCHMARK. Phoronix CPU OpenBenchmark, 2010.
- [65] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011. ISBN 012383872X, 9780123838728.
- [66] John D McCalpin. STREAM benchmark. *Link: www.cs.virginia.edu/stream/ref.html# what*, 22, 1995.
- [67] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press. doi: <http://doi.acm.org/10.1145/1167473.1167488>.
- [68] GNU Free Software Foundation. GNU/Resource.h syscall API.  
 [Online], Available: [http://www.gnu.org/software/libc/manual/html\\_node/Resource-Usage.html](http://www.gnu.org/software/libc/manual/html_node/Resource-Usage.html), Accessed on 15/06/2017.

## 7 Appendix

### 7.1 User Guide

This guide forms a setup and manual in using PowerKap and LJEA.

#### 7.1.1 Setup

First, setup and install the libraries. To install PowerKap, this is a relatively simple procedure. All that is needed is to run the commands “make optimised” followed by “sudo make install”. This procedure compiles PowerKap with the -O2 flag and installs the files in /usr/local directory. It also allocates the correct permissions and capabilities for the file and shell script.

To install LJEA, all that is necessary is to install the zip from disk. This can be done by performing the following action on the menus.

*File > Settings > Plugins > Install plugin from disk*

#### 7.1.2 How to use PowerKap

This program can work from command line without root. Before profiling a binary, it is necessary to setup the virtual adapter to gather network traffic. This can be achieved by running “sudo profKap -a”. From that point, it is only necessary to pick an internet facing network adapter. Upon installation the user makes use of a series of flags to set the parameters of the profiling. These are listed as follows.

1. -a This enables the virtual adapter necessary for PowerKap to run.
2. -t This sets the time in which the program should run. It is useful for programs that do not terminate. The value is recorded in seconds.
3. -r This is used to set the number of times PowerKap repeats the test.
4. -i This sets the interval between sample points. The value passed is recorded in milliseconds.
5. -e The flag necessary to tell the program to profile an external program. To use this flag requires program arguments. An example of such a command is as follows:

`profKap -e — /usr/bin/program <program-args>`

6. -s This tells the profiler to skip profiling the background.
7. -z Print intermediate values. This flag is useful for gathering raw and intermediate results for further verification.
8. -x Prints the results gathered to std::cout.
9. -c Prints the results in JSON format. This is needed to be used in LJEA. These are stored in the calling directory for the program.

10. -v Prints the results gathered in CSV format. These are stored in the calling directory for the program.
11. -b Prints the results gathered in TSV format. These are stored in the calling directory for the program.
12. -g Skips gathering network measurements.
13. -h Skips gathering CPU RAPL metrics.
14. -j Skips gathering data relating to the disk.
15. -k Skips gathering battery information.
16. -l Skips gathering temperature information.

### 7.1.3 How to use LJEA

In order to use the plugin to profile code requires the following steps.

1. The first step is to enable the time profiling capabilities of the program. This can be done by performing the following action in the main toolbar.

*Energy module > Enable TimePoint Logging*

2. For positions of interest, we want to automatically generate energy points. This can be done by right clicking the position on a line of code and selecting generate. Then simply just select the insert energy point option.
3. At the program exit, it is necessary to insert a print point. This procedure is the same as the one used to insert the point.

The procedure for analysing the results gathered is also simple. All that is necessary is to upload the JSON and the elapsed times file to the plugin via the provided buttons. To use the stack trace, simply highlight a specific JSON file within the configuration table and click refresh. This would load the data into the elapsed stack trace tab. To annotate the graph, simply select the points of interest and refresh the graph to add markings. Finally, to navigate back to the source code, simply right click the energy point to navigate to the relevant piece of code.