Exploring the Energy Consumption of Highly Parallel Software on Windows

Mads Hjuler Kusk*, Jeppe Jon Holt*
and Jamie Baldwin Pedersen*
Department of Computer Science, Aalborg University, Denmark
*{mkusk18, jholt18, jjbp18}@student.aau.dk

May 1, 2023

Abstract

With the evolution of CPUs over the last few years, an increase in the number of cores has become the norm. This work investigates the performance gains obtained from the additional processing power and the impact of the P- and E-cores. Performance is analyzed using benchmarks, considering both energy consumption and execution time on a per-core basis and with an increasing number of cores. As this work is based on Windows, where Intel's Running Average Power Limit is unavailable, we compare alternative measurement methods for Windows. The benchmarks in this work include both microbenchmarks and macrobenchmarks illustrating a more realistic test case.

1 Introduction

In recent years there has been rapid growth in Information and Communications Technology (ICT) which has led to an increase in energy consumption. Furthermore, it is expected that the rapid growth of ICT will continue in the future. [1, 2] As the use of ICT rises the demand for computational power rises as well, therefore energy efficiency has or perhaps should become more of a concern for companies and software developers alike.

In this paper, we investigate energy consumption of various benchmarks on Windows 11, comparing the efficiency and tradeoffs between sequential and parallel execution. Our experiments involve two Device Under Tests (DUTs): an Intel Coffee Lake CPU with a traditional performance core setup and an Intel Raptor Lake CPU with performance and efficiency cores (P- and E-cores). We analyze the impact of Asymmetric Multicore Processors (AMPs) on parallel execution compared to traditional Symmetric MultiCore Processors. In the first two experiments conducted, the focus will be on C++ where different C++ compilers and measuring instruments for Windows are compared and explored using microbenchmarks. C++ was chosen to avoid noises from e.g. garbage collectors or just-in-time compilation. The third experiment will use the best performing measuring instrument to go beyond C++ programs, to larger macrobenchmarks. The macrobenchmarks will be run on an increasing number of cores in order to explore the benifits in terms of runtime and energy consumption. The following research questions are formulated to assist with the process:

- RQ**1**: How does the C++ compiler used to compile the benchmarks impact the energy consumption?
- RQ**2**: What are the advantages and drawbacks of the different types of measuring instruments in terms of accuracy, ease of use, and availability?
- RQ**3**: What effect does parallelism have on the energy consumption of the benchmarks?
- RQ**4**: What effect do P- and E-cores have on the parallel execution of a process?

To answer these research questions a command line framework is created to assist with running a series of different experiments.

In Section **2** the related work which lay the foundation for our work is covered, including our previous work. This is followed by Section **3** which includes the necessary background information about e.g. CPUs and schedulers. Thereafter in Section **4** our experimental setup is presented. In Section **5** the results are presented whereafter they are discussed in Section **6** and finally a conclusion is made in **??**.

2 Related Work

This section provides an overview of related work in energy consumption, parallel software, compilers, and asymmetric multicore processors. It also builds upon our previous work in comparing measuring instruments.

2.1 Previous Work

In our previous work *"A Comparison Study of Measuring Instruments"*[3], different measuring instruments were compared to explore whether a viable software-based measuring instrument was available for Windows. It was found that Intel Power Gadget (IPG) and Libre Hardware Monitor (LHM) on Windows have similar correlation to hardware-based measuring instruments as Intel's Running Average Power Limit (RAPL) has on Linux. The remainder of this chapter builds upon the related work chapter in [3].

2.2 Parallel Software

Amdahl's law describes the potential speedup achieved by running an algorithm in parallel based on the proportion of the algorithm that can be parallelized and the number of cores used.[4] In [5] Amdahl's law, was extended to also estimate the energy consumption, after which three different many-core designs were compared with different amounts of cores using the extended Amdahl's law. The comparison showed that a CPU can lose its energy efficiency as the number of cores increases and it was argued that knowing how parallelizable a program is before execution allows for calculating the optimal number of active cores for maximizing performance and energy consumption. The comparison was however based on an analytical model and not real measurements.[5]

[6] compares the observed speedup of computing Laplace equations with one, two, and four cores, with estimates given by Amdahl's law and Gustafson's law. Gustafson's law evaluates the speedup of a parallel program based on the size of the problem and the number of cores. Unlike Amdahl's law which assumes a fixed problem size and a fixed proportion of the program that can be parallelized, Gustafson's law takes into account that larger problems can be solved when more cores are available and that the parallelization of a program can scale with the problem size. Comparing the observed and estimated speedup it was clear that Gustafson's law is more optimistic than Amdahl's law, however, both estimate smaller speedups than the observed speedup on two and four cores. [6]

In [7], three different thread management constructs from Java were explored and analyzed. It was found that the energy consumption increased with the number of threads used. However, after a point energy consumption would start to decrease as the number of threads approached the number of cores in the CPU. The exact peak of when the energy consumption was however found to be application-dependent. The study also found that in eight out of nine benchmarks, there was a decrease in execution time when transitioning from sequential execution on one thread to using multiple threads. It should be noted, though, that four of their benchmarks were embarrassingly parallel, while only one was embarrassingly serial. Moreover, decreased execution time does not necessarily imply decreased energy consumption, because in six out of nine benchmarks, the lowest energy consumption was found in the sequential version. Furthermore, the study used Energy-Delay-Product, the product of energy consumption and execution time and found that in general parallel execution was favorable. However, increasing the number of threads was not an improvement for all of the benchmarks.[7]

In [8], found that a larger number of cores in the execution pool results in a lower running time and energy consumption, and conclude that parallelism can help reduce energy consumption for genetic algorithms. When considering parallel software, it also found asynchronous implementations to use less energy, because there are no idle cores waiting for data in asynchronous implementations, while in synchronous implementations cores can be blocked during runtime, while waiting for responses from other cores.

In [9], four different language constructs which can be used to implement parallelism in C# are tested. Furthermore, they uses varying amounts of threads and a sample of micro- and macro-benchmarks. They found that workload size has a large influence on run time and energy efficiency and that a certain limit must be reached before improvements can be observed when changing a sequential program into a parallel one. Additionally, it was found that execution time and energy consumption of parallel benchmarks do not always correlate. Comparing micro- and macrobenchmarks the findings remain consistent, although the impact becomes lower for the macrobenchmarks due to an overall larger energy consumption. Furthermore, it has included some recommendations, which are considered in our setup:[9]

- Static clock: Make the clock rate of the CPU as static as possible
- Turn off CPU turbo boost
- Turn off hyperthreading

2.3 Compilers

In [10], several C++ compilers were compared, with the goal of finding a balance between performance and energy efficiency. The study was conducted on different microbenchmarks, where the effect of different coding styles was explored.The C++ compilers used in [10] included MinGW GCC, Cygwin GCC, Borland C++, and Visual C++, and the energy measurements were taken using Windows Performance Analyzer (WPA). The compilers were used with their default settings, and no optimizations options were used. They found that when choosing a compiler and coding style, energy reduction depended on the specification of the target machine and the individual application. Based on the benchmark used, which involved an election sort algorithm, the lowest execution time was achieved with the Borland compiler, and the lowest energy consumption was observed with the Visual C++ compiler. When considering the coding styles, the study found that separating IO and CPU operations and interrupting CPU-intensive instructions with sleep statements both decreased energy consumption.

2.4 Asymmetric Multicore Processors

Asymmetric Multicore Processors are CPUs where not all cores are treated equally. One example of this could be the combination of P- and E-cores, as seen in Intel's Alder Lake and Raptor Lake. Intel's Thread Director (ITD) was introduced alongside Intel's Alder Lake, where the purpose of ITD was to assist the operating system (OS) when deciding which cores to run a thread. In [11], support for utilizing ITD in Linux was developed, and some SPEC benchmarks was conducted to analyze the estimated Speedup Factor (SF) from the ITD compared to the observed SF. SF is the relative benefit a thread receives from running on a P-core. The study examined which classes were assigned to different threads in the benchmark and found that 99.9% of class readings were class 0 or 1. Class 0 are for threads performing similarly on P- and E-cores, while Class 1 are for threads where P-cores are preferred.[12] Class 3, which are for threads preferred to be on an E-core, was not used. The experiment indicated that the ITD overestimated the SF of using the P-cores for many threads but also underestimated it for some threads. Overall, it was found that the estimated SF had a low correlation coefficient (< 0.1) with the observed values. Furthermore, a performance monitoring counter (PMC) based prediction model was trained. The model outperformed ITD, but still produced errors. However, the correlation coefficient was higher at (> 0.8). The study implemented support for the IDT in different Linux scheduling algorithms and compared the results from using the IDT and the PMC-based model. It found that the PMC-based model provided superior SF predictions compared to ITD.[11] Official support for ITD has since been released.

3 Background

In the following section, different technologies used later will be introduced.

3.1 CPU States

CPU-states (C-state) manage a systems energy consumption during different operational conditions. On a CPU, each core has its own state, which dictates how much the core is shut down in order to conserve power. The C0 state represents the normal operation of a core under load.[13, 14] The number of states vary between CPUs and the number of supported states vary between motherboards. The CPU used in [3] had 10 states, where states higher than C0 represents an increasingly shut down core and the highest states will mean the core is almost inactive.[3].

The C-states can have a large impact on the energy consumption of the benchmarks, especially the idle case as was found in [3].

3.2 Performance and Efficiency cores

For the CPU architecture x86, the core layout has historically comprised of identical cores. However, the ARM architecture introduced the big.LITTLE layout in 2011[15]. big.LITTLE is an architecture utilizing two types of cores, a set for maximum energy efficiency and a set for maximum computer performance.[16]. Intel introduced a hybrid architecture in 2021[17] similar to big.LIGGLE, codenamed Alder lake. Alder lake has two types of cores: P-cores and E-cores, each optimized for different tasks. P-cores are standard CPU cores focusing on maximizing performance and E-cores are designed to maximize performance per watt and are intended to handle smaller non-time critical jobs, such as background services[18].

```csharp
void ExecuteWithAffinity(string path)
{
    var process = new Process();
    process.StartInfo.FileName = path
    process.Start();

    // Set affinity for the process
    process.ProcessorAffinity =
        new IntPtr(0b0000_0011)
}
```

Listing 1: An example of how to set affinity for a process in C#

3.3 CPU Affinity

Affinity is a feature in the OS enabling processes to be bound to specific cores. Jobs and threads are constantly rescheduled for optimal system performance on an OS, meaning that the same process can be assigned to different cores on the CPU. Processor affinity allows applications to bind or unbind a process to a specific set of cores. When a process is pinned to a core, the OS ensures the process only executes on the assigned core(s) each time it is scheduled.[19]

Processor affinity is particularly useful for scaling performance on multi-core processor architectures sharing the same global memory and have local caches referred to as the Uniform memory access architecture.[19]

When setting the affinity for a process in C#, it is done through a bitmask, where each bit represents a CPU core. An example is found in Listing **1**, where the process is allowed to execute on core #0 and #1.

3.4 Scheduling Priority

Scheduling threads on Windows, is done based on each thread's scheduling priority level and the priority class of the process. The priority class can be either `IDLE`, `BELOW NORMAL` `NORMAL`, `ABOVE NORMAL`, `HIGH` or `REALTIME`, where the default is `NORMAL`. It is noted that `HIGH` priority should be used with care, as other threads in the system will not get any processor time while that process is running. If a process needs `HIGH` priority, it is recommended to raise the priority class temporarily. The `REALTIME` priority class should only be used for applications that "talk" to hardware directly, as this class will interrupt threads managing mouse input, keyboard inputs, etc.[20]

The priority level can be either `IDLE`, `LOWEST`, `BELOW NORMAL`, `NORMAL`, `ABOVE NORMAL`, `HIGHEST` and `TIME CRITICAL`, where the default is `NORMAL`. A typical strategy is to increase the level of the input threads for applications to ensure they are responsive, and to decrease the level for background processes, meaning they can be interrupted as needed.[20]

The scheduling priority is assigned to each thread as a value from zero to 31, where this value is called the base priority. The base priority is decided using both the thread priority level and the priority class, where a table showing the scheduling priority given these two parameter can be found in [20].

```
void ExecuteWithPriority(string path)
{
  var process = new Process();
  process.StartInfo.FileName = path
  process.Start();

  // Set priority class for process
  process.PriorityClass =
    ProcessPriorityClass.High;

   // Set priority level for threads
  foreach (var t in process.Threads)
  {
    thread.PriorityLevel =
      ThreadPriorityLevel.Highest;
  }
}
```

Listing 2: An example of how to set priorities for a process in C#

When setting scheduling priority, the priority class is supported for both Windows and Linux, while the priority level is only supported for Windows. An example of how both priorities are set for a process and its threads can be seen in Listing **2**.

3.5 Open Multi-Processing

Open Multi-Processing (OpenMP) is a parallel programming API consisting of a set of compiler directives and runtime library routines, with support for multiple OSs and compilers.[21] The directives provide a method to specify parallelism among multiple threads of execution within a single program without having to deal with low-level details, while the library provides mechanisms for managing threads and data synchronization.[21]

When executing using OpenMP, the parallel mode used is the Fork-Join Execution Model. This model begins with executing the program with a single thread, called the master thread. This thread is executed serially until parallel regions are encountered, in which case a thread group is created, consisting of the master thread, and additional worker threads. After splitting up, each thread will execute until an implicit barrier at the end of the parallel region. When all threads have reached this barrier, only the master thread continues.[21]

```
#pragma omp directive-name [
    clause[ [,] clause]...
    ]
```

Listing 3: The basic format of OpenMP directive in C/C++

The basic format of using OpenMP can be seen in Listing **3**. By default, the parallel regions are executed using the number of present threads in the system, but this can also be specified using `num_threads(x)`, where `x` represents the number of threads.[21]

4 Experimental Setup

In the following section a detailed description of the equipment, benchmarks, and procedures used in the study will be presented.

4.1 Measuring Instruments

The measuring instruments used in this work are based on what what used in [3], with a few additions. Thew new additions will get a more detailed introduction, while the others are briefly introduced but can be found with more detail in [3].

Intel's Running Average Power Limit (RAPL): is a software-based measuring instrument most frequently used in the litterature.[3]. RAPL uses model-specific-registers (MSRs) and Hardware performance counters to calculate how much energy the CPU uses.

The MSRs RAPL uses include *MSR_PKG_ENERGY_STATUS*, *MSR_DRAM_ENERGY_STATUS*, *MSR_PP0_ENERGY_STATUS* and *MSR_PP1_ENERGY_STATUS*. RAPL is only supported on Linux and Mac. In [3] RAPL was found to have a high correlation of 0.81 with a hardware measurement.[3]

Intel Power Gadget (IPG): is a software tool created by Intel, which can estimate the power of Intel processors. It uses the same hardware counters and MSRs as RAPL[22], therefore it is expected to have similar measurements to RAPL. Which was also found in [3], where IPG had a high correlation of 0.78 with the ground truth on Windows and a high correlation of 0.83 with RAPL on Linux.[3]

Libre Hardware Monitor (LHM): is a fork of Open Hardware Monitor, without a GUI.[23] Both projects are open source and LHM uses the same hardware counters and MSRs as RAPL and IPG. Therefore, a similar measurement is expected between LHM, IPG and RAPL. In [3] LHM on Windows was found to have a high correlation of 0.76 with our ground truth on Windows and a high correlation of 0.85 with IPG.

MN60 AC Current Clamp (Clamp): is a current clamp connected to the phase of the wire going into the power supply unit (PSU), which serves as the ground truth. The clamp is connected to an Analog Discovery 2, where the Analog Discovery 2 is connected to a Raspberry Pi 4 in order to measure and log measurements continuously.[3] The accuracy is reported to be 2%[24].

CloudFree EU smart Plug (Plug): is used, as an alternative lower-priced hardware-based measuring instrument, which also has greater ease of use than the Clamp setup. The accuracy and sampling rate of the plug is unknown.[25]

Scaphandre (SCAP): is described as a monitoring agent that can measure energy consumption.[26] SCAP is designed for Linux and uses RAPL and can in addition to this measure the energy consumption of some virtual machines, specifically Qemu and KVM hypervisors. SCAP can also be used on Windows, as a kernel driver exists which allows SCAP to read RAPL measurements from Windows.[27]. The Windows version of SCAP can report the energy consumption of the power domain PKG using the MSR *MSR_PKG_ENERGY_STATUS*. SCAP can also estimate the energy consumption for individual processes by storing CPU usage statistics alongside the energy counter values and then calculating the ratio of CPU time for each Process ID (PID). Using the calculated ratio SCAP estimates the subset of energy consumption that belong to a specific PID. In this work, the performance of SCAP and SCAPs ability to isolate the energy of a process will both be used, where the latter will be referenced as SCAPI.

4.2 Dynamic Energy Consumption

Dynamic Energy Consumption (DEC) was utilized in [3, 28] to enable comparison between the software-based measuring instruments and the hardware-based measuring instruments, where the former measures energy consumption of the CPU only and the latter the entire DUT. DEC was also used in our work. A brief explanation of DEC based on [28] is given:

$$E_D = E_T - (P_S * T_E) \tag{1}$$

In Equation (1) E_D is the DEC, E_T is the total energy consumption of the system, P_S is the energy consumption when the system is idle and T_E is the duration of the program execution. With this equation the energy consumption of the benchmark is isolated. Using DEC requires also measuring the energy consumption on an idle case. [28]

4.3 Statistical Methods

In this sections the statistical method used to analyze our results are presented. This section is based on what was found in [3] and can be referred to for further detail.

Values	Label
$< .20$	Almost negligible correlation
$.20 - .40$	Low correlation
$.40 - .70$	Moderate correlation
$.70 - .90$	High Correlation
$.90 - 1$	Very high correlation

Table 1: The values for the scale presented by Guildford in [29, p. 219]

Shapiro-Wilk Test: was used to examine if the data followed a normal distribution. We expected or data to not be normally distributed, because that was found to be the case is [3]. Understanding the distribution of the data was essential when choosing subsequent statistical methods.[30]

Mann-Whitney U Test: to evaluate if there is a statistical significant difference between samples the Mann-Whitney U Test was used, because it is a non-parametric test that does not assume normality in the data.[31]

Kendall's Tau Correlation Coefficient: was used to asses the correlation between our measurements. It is a non-parametric measure of association that can evaluate the strength and direction of relationships between ordinal variables, when the underlying data does not adhere to a normal distribution.[32] The correlation can be evaluated using the Guilford scale [29, p. 219] as can be seen in Table **1**.

Cochran's Formula: To determine an appropriate sample size for our measurements, Cochran's formula was used. With this formula a required sample size to achieve a desired level of statistical power can be calculated.[33]

In summary, the selection of the Shapiro-Wilk test, Mann-Whitney U test, Kendall's Tau correlation coefficient, and Cochran's formula allowed us to effectively analyze our data, taking into account its non-normal distribution and ordinal nature while determining statistically significant differences, correlations, and an appropriate sample size for our measurements.

4.4 Device Under Tests

Two workstations were used as DUTs in the experiments. These were chosen to enable comparison between CPUs with and without P- and E-cores. When the two DUTs were set up, they were updated to have the same version of Windows and Linux. In Tables **2** and **3** the specifications of the two workstations can be seen. They will be referred to as DUT 1 and DUT 2.

Workstation 1 (DUT 1)	
Processor:	Intel i9-9900K
Memory:	DDR4 16GB
Disk:	Samsung MZVLB512HAJQ
Motherboard:	ROG STRIX Z390 -F GAMING
PSU:	Corsair TX850M 80+ Gold
Ubuntu:	22.04.2 LTS
Linux kernel:	5.19.0-35-generic
Windows 11:	10.0.22621 Build 2262

Table 2: The specifications for DUT 1

Workstation 2 (DUT 2)	
Processor:	Intel i5-13400
Memory:	DDR4 32GB
Disk:	Kingston SNV2S2000G
Motherboard:	ASRock H610M-HVS
PSU:	Cougar GEX 80+ Gold
Ubuntu:	22.04.2 LTS
Linux kernel:	5.19.0-35-generic
Windows 11:	10.0.22621 Build 22621

Table 3: The specifications for DUT 2

When running the experiments, the recommendations presented in [9] were followed. These included that the WiFi, Intel Turbo Boost and hyperthreading was disabled. Lastly, the CPU was set to static, which was achieved by disabling the C-states in the bios.

4.5 Compilers

This section introduces the various C++ compilers that were used in the first experiment. Some of the chosen compilers were based on [10], which found that applications compiled by Microsoft Visual C++ and MinGW exhibited the lowest energy consumption. Additionally, the Intel OneApi C++ compiler and Clang were included as both can be found on lists of the most popular C++ compilers[34–36].

C++ Compilers	
Name	Version
Clang	15.0.0
MinGW	12.2.0
Intel OneAPI C++	2023.0.0.20221201
MSVC	19.34.31942

Table 4: C++ Compilers

Clang: is an open source compiler that builds on the LLVM optimizer and code generator. It is available for both Windows and Linux[37]

Minimalist GNU for Windows (MinGW): is an open-source project which provides tools for compiling code using the GCC toolchain on Windows. It includes a port of GCC. Additionally, MinGW can be cross-hosted on Linux.[38]

Intel's oneAPI C++ (oneAPI): is a suite of libraries and tools aimed at simplifying development across different hardware. One of these tools is the C++ compiler, which implements SYCL, this being an evolution of C++ for heterogeneous computing. It is available for both Windows and Linux.[39]

Microsoft Visual C++ (MSVC): comprises a set of libraries and tools designed to assist developers in building high-performance code. One of the included tools is a C++ compiler, which is only available for Windows[40].

4.6 Benchmarks

Our work employed microbenchmarks and macrobenchmarks to asses the measuring instruments. This section outlines the selected benchmarks and the rationale behind their selection.

Microbenchmarks: are small, focused benchmarks that test a specific operation, algorithm or piece of code. They are useful for measuring the performance of some particular code precisely while minimizing the impact of other factors. However microbenchmarks may not provide an accurate representation of overall performance.[41]

The microbenchmarks are from the Computer Language Benchmark Game [1]. The selected benchmarks include both single- and multi-threaded microbenchmarks, which are compatible with the chosen compilers, as well as with both Windows and Linux. Certain libraries, such as `<sched.h>`, were used in many implementations and was not available on Windows, which limited the pool of compatible microbenchmarks. The microbenchmarks were executed using the highest parameters specified in the Computer Language Benchmark Game as input for each benchmark. The chosen microbenchmark benchmarks and their abbreviation are presented in Table **5**. During compilation, the only parameter given is `-openmp` for the multi-core benchmarks, ensuring optimization for all cores of the DUT.

Microbenchmarks		
Name	Parameter	Focus
NBody (NB)	$50 * 10^6$	single core
Spectra-Norm (SN)	5.500	single core
Mandelbrot (MB)	16.000	multi core
Fannkuch-Redux (FR)	12	multi core

Table 5: Microbenchmarks

Macrobenchmarks: are large-scale benchmarks testing the performance of an entire application or system. They provide a more comprehensive overview of how the system performs in real-world scenarios. Macrobenchmarks are more suitable for understanding the overall performance of an application or system rather than focusing on specific operations.[41] Application-level benchmarks are a type of marco benchmarks that test an application, which provides a more realistic benchmark scenario. Two macro benchmarks developed by UL were used. The first one was 3DMark (3DM) which is a set of benchmarks for scoring both GPU's and CPU's based on gaming performance. We only used the 3DM benchmark CPU Profile, because we were only interested in loading the CPU and not the GPU, which the other benchmarks does. The CPU Profile benchmarks runs a 3D graphic, but the main component of the workloads is from a boids flocking behavior simulation.[42]. The second one was PCMark 10 (PCM) which is a benchmark meant to test various different task which could be seen at a workplace. It has three test groups that includes e.g. web browsing, video conferencing, working in spreadsheets and photo editing, the full list can be seen in Tables **11** and **12**. This benchmark simulated common task in office workspace.[43] The versions of both macrobenchmarks can be seen in Table **6**.

Macrobenchmarks	
Name	Version
3D Mark (3DM)	2.26.8092
PC Mark 10 (PCM)	5.61.1173.0

Table 6: Macrobenchmarks

4.7 Background Processes

To limit background processes on Windows, a few step were taken. When the DUTs were set up, all startup processes in the Task Manager on Windows were also disabled, in addition to non-Microsoft background services found in System Configuration. Exceptions were however made to processes related to Intel.

During runtime, different background processes were also stopped. These processes were found by looking at the running processes using command `Get-Process`. A list of processes was found which are killed using the `Stop-Process` command before running the experiments. The list can be found in Table **7**.

Background Processes
Name
searchapp
runtimebroker
phoneexperiencehost
TextInputHost
SystemSettings
SkypeBackgroundHost
SkypeApp
Microsoft.Photos
GitHubDesktop
OneDrive
msedge
AsusDownLoadLicense
AsusUpdateCheck

Table 7: Background Processes

5 Experiments

In the following section, the conducted experiments are described. All experiments carried out in this section will utilize the framework detailed in Appendix **B**, with the results stored in the database introduced in Appendix **C**. During the experiments, the `ProcessPriorityClass` for the measuring instrument,

[1] https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

framework, and benchmarks was set to `High`, unless specified otherwise by the particular experiment.

5.1 Experiment One

The first experiment investigated RQ **1**. This experiment employed both multi-core benchmarks presented in subsection **4.6**, and the measurements were performed using IPG. IPG was chosen based on its performance in [3], where it was found to produce similar measurements to LHM. Since the objective of this experiment was to identify the most energy-efficient compiler, the expectation was that a similar conclusion would be made if multiple measuring instruments were used. This experiment was conducted on DUT 1. This experiment was made based on an hypothesis that the different compilers would produce assembly with a varying energy consumption, as was also found in [10].

Compiler Initial Measurements: As was presented in subsection **4.3**, Cochran's formula was used to ensure there was confidence in the measurements made. The initial measurements were taken to gain insight into the number of measurements required before making additional measurements if required. The number chosen for the initial measurements was 30, as the central limit theorem suggests that a sample size of at least 30 is usually sufficient to ensure that the sampling distribution of the sample mean approximates normality, regardless of the underlying distribution of the population[44].

After 30 measurements, the results from Cochran's formula can be seen in Table **8**, where it was evident that the required samples varied between compilers and benchmarks. When the benchmarks were analyzed it was found that MB deviates less than FR, with MB requiring as little as 3 measurements with MinGW, while FR requires up to 62.086 samples with Clang. Given these results, more measurements were necessary. When the compilers were analyzed interestingly oneAPI had the lowest required samples for FR, but the highest for MB. oneAPI also displayed the lowest energy consumption. 550 additional measurements were conducted for the next step.

Initial Measurements		
Name	FR	MB
Clang	61.086	40
MinGW	1.644	3
oneAPI	550	222
MSVC	2.994	10

Table 8: The required samples to gain confidence in the measurements made by IPG on Windows

Compiler Results: After 550 measurements were obtained, the reported values by Cochran's formula still indicated that MSVC, MinGW, and Clang needed more measurements compared to oneAPI. Between the different compilers, Clang stands out where 61.086 measurements are required. Because this number is so much higher than other compilers, additional measurements were taken using this compiler. After 10.000 measurements, Cochran's formula now indicated that 1.289 measurements were required, which is more in line with other compilers.

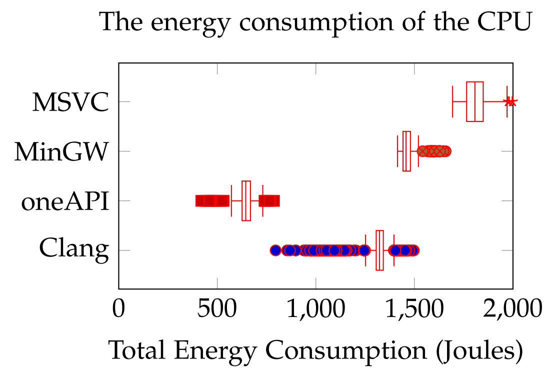


Figure 1: CPU measurements by IPG on DUT 1 for benchmark(s) FR

When looking at the results for FR in Figures **1** and **2**, and for MB in Appendix **E**, oneAPI had the lowest DEC for both benchmarks. Clang deviated the most in Figure **2**.

In the first experiment, it was concluded that the different compilers have a huge impact on the DEC but also how many measurements were required to be confident in the results. In the end, oneAPI had the lowest DEC and was used in the next experiment.

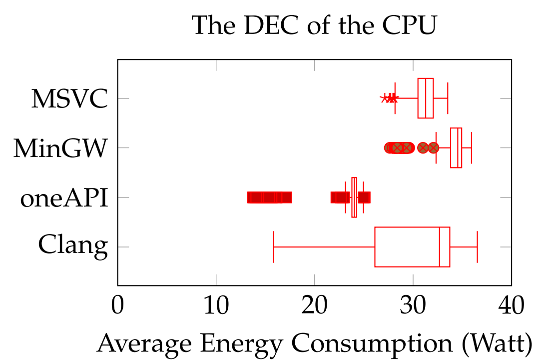


Figure 2: CPU measurements by IPG on DUT 1 for benchmark(s) FR

5.2 Experiment Two

The second experiment investigated RQ **2**, in order to identify our preferred measuring instrument on Windows. The measuring instrument was chosen based on a combination of different factors, including its correlation with our ground truth, ease of use and cost.

A couple of changes were made in the experimental setup for experiment two. Firstly, due to some issues with SCAP, where its sampling rate significantly decreased when the DUT was under full load, the process priority class of the benchmark was set to `Normal`. Secondly, due to an execution time of less than a second for `MB` when compiled with oneAPI, `MB`'s input parameter was changed from 16.000 to 64.000 which increased the duration of the benchmark execution time to $\sim$ 14 seconds. This avoided a scenario where the Plug only had a single data point per measurement. For this experiment, `FR` was executed 550 times, while `MB` was executed 222 times, based on Table 8.

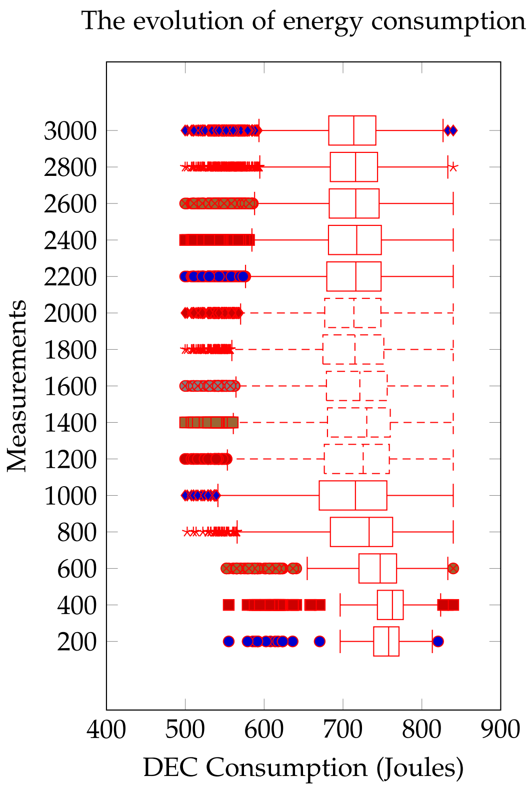


Figure 3: A visual representation of how the energy measurements evolve as more measurements are made by clamp on DUT 2 for benchmark MB

Measuring Instrument Initial Measurements: When analyzing how many measurements is required when applying Cochrans to the results can be seen in Appendix **G**. The Clamp requires significantly more measurements in this case compared to other measuring instruments, which is why a more in depth analysis was conducted. In Figure 3 boxplots showing the evolution of the DEC when performing between $200 - 3.000$ measurements. The median decreased by 5.84% from 200 measurements to 3.000 measurements, and by 0.3% between 2.800 and 3.000 measurements. A pattern was observed, where the median decreased as more measurements are made, until measurements 1.000, after which the DEC increases until measurement 1.400 by 2%, after which it decreases again. In the last 1.400 measurements the DEC has converged where the DEC increases by 0.2%. The DEC at 1.000 measurements is 0.29% from the DEC at 3.000, and due to the excessive time required to run the experiments, we have capped the maximum amount of measurement at 1000 for this experiment. When looking at the evolution of Cochran's formula for the different measurements, 15.137 ends up being the amount of measurements required, where the evolution of this number can be found in Appendix **H**. This number is higher compared to other measuring instruments, and this will be analyzed further in the discussion.

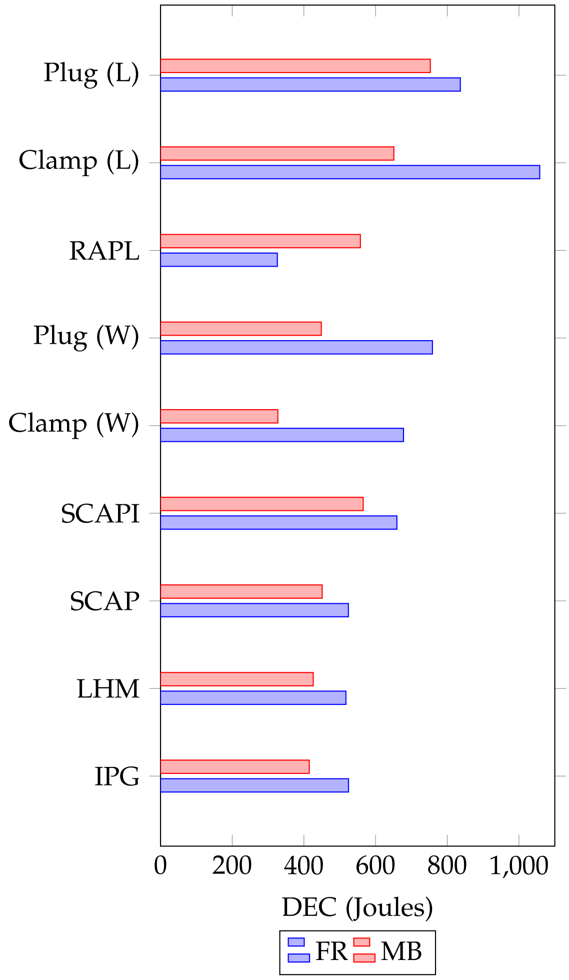


Figure 4: The average DEC for DUT 1, where both benchmarks are compiled on oneAPI

Measuring Instrument Results: In Figure 4 `MB` had a lower energy consumption than `FR` for all measuring instruments except RAPL. SCAP, LHM and IPG had measurements within 25 joules of each other. The Clamp (W) measurement are lower than the Plug (W) on both benchmarks, while compared to SCAP, SCAPI, LHM and IPG it is lower for `MB`, but higher for `FR`. When comparing between OSs, Windows can be observed to have a lower DEC and Linux. Boxplots for both DUTs can be found in Appendix **F**.

When applying statistical methods from subsection **4.3**, it was discovered that some of the data did not follow a normal distribution and were significantly different from each other, previous studies [3, 45] have had similar results. Thus, Kendall's Tau Correlation Coefficient was used.

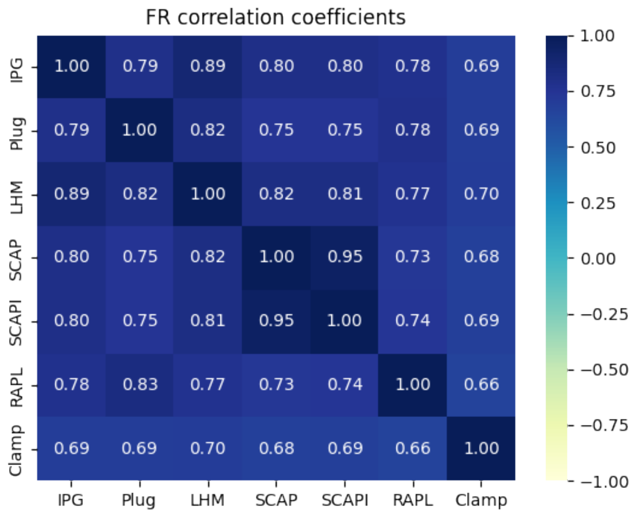


Figure 5: Heatmap showing the correlation coefficient between all of the measurement instrument on Windows for FR

All the measuring instruments showed moderate to high correlation with the ground truth (Clamp) when assessed with the Guildford Scale. On FR in Figure **5** the software-based measuring instruments had a correlation of 0.7 - 0.75 while the Plug had a correlation of 0.55, indicating lower accuracy. While on MB they correlations were a little lower for the software based measuring instruments as shown in Appendix **G**. We expected the correlation of the software based measuring instruments to be similar since they are using the same hardware counters and MSRs. For the remaining experiments, we chose the software-based instrument based on considerations of accuracy, ease of use, and availability as expressed in RQ **2**. While SCAPI had the highest correlation on both MB and FR, it and SCAP had a low sample rate and was tedious to set up on Windows, therefore it is not picked. LHM and IPG were close as IPG was slightly more correlated with the Clamp (W) on MB, but slightly less on FR. However, LHM had more problems in the setup phase than IPG and also required calculations for getting the measurements in joules, therefore, we chose IPG.

5.3 Experiment Three

The third experiment investigated RQs **3** and **4**, by taking a look at the per-core performance. In this experiment only IPG and the Clamp was used to conduct measurements. This experiment will explore what benefit macrobenchmarks gain from additional allocated cores, by executing PCM and 3DM on an increasing number of cores. Before this is done, an analysis on the per-core performance of both CPUs will be conducted, where the single-core benchmarks introduced in subsection **4.6** will be used. This allowed a comparison between the energy consumption of the P- and E-cores on DUT 2 and the P-cores on DUT 1. When performing measurements, the limit of 1.000 measurements set in subsection **5.2** is still used.

Per-Core Initial Measurements: An initial 250 measurements were made for each benchmark on each core. Then Cochran's formula was calculated to determine if more measurements were required. Results from Cochran's can be found in Appendix **J**.

Per-Core Results: The results, presented here are based on DUT 2, where the results for DUT 1 can be found in Appendix **I**. For SN, as seen in Table **9**, the run time was on average 76.32% lower on the P-cores compared to the E-cores and The total DEC was on average 94.59% lower on P cores, however the P cores had a 254.71% higher energy consumption per second. When comparing the percent difference between P-and E cores between the energy consumption and DEC, a larger difference was found for DEC. This is a result of DEC excluding the idle energy consumption from the measurements, resulting in lower values which means the difference being larger values. The largest difference between two cores of the same type was found on DUT 1 with benchmark NB, where the performance was 11.61% worse on core 1 than core 6. The smallest difference was found on DUT 2, benchmark NB on a E core, where the energy consumption was 1.17% higher on core 6 than core 9.

SN measurements on DUT 2			
Metric	E-core	P-core	Difference
Duration	58.96 s	13.96 s	−76.32%
Energy	336.88 j	99.53 j	−70.45%
DEC	253.85 j	16.26 j	−93.59%
DEC per second	0.53 w	1.88 w	+254.71%

Table 9: The average performance difference between E and P cores on DUT 2, SN

Macrobenchmark Initial Measurements: An initial 30 measurements were made for 3DM and PCM on an increasing number of cores. 30 measurements was chosen as the per-core experiment illustrated how 250 was too much for DUT 2, illustrated in Appendix **J**. The initial idea was to start at one core, which is done for 3DM for both DUTs and PCM on DUT 1. On DUT

2, PCM could not execute web browsing on a single core, and was unable to execute spreadsheet and photo editing. Because of this, DUT 2 will start at 2 cores. For DUT 1, web browsing was unable to execute, so this scenario is excluded for this DUT. The order of cores used in this experiment was done by using the cores with the lowest DEC found in Appendix **I**. After an the initial 30 measurements, Cochrans formula was applied to the data, to take additional measurements if required. The amount of required measurements can be found in Appendix **K**.

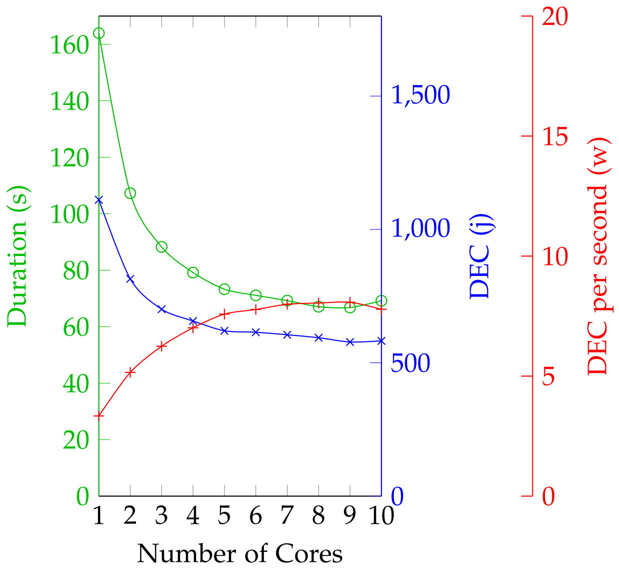


Figure 6: The evolution of the DEC (blue), DEC per second (red) and duration (green) as more cores are allocated to 3DM on DUT 2

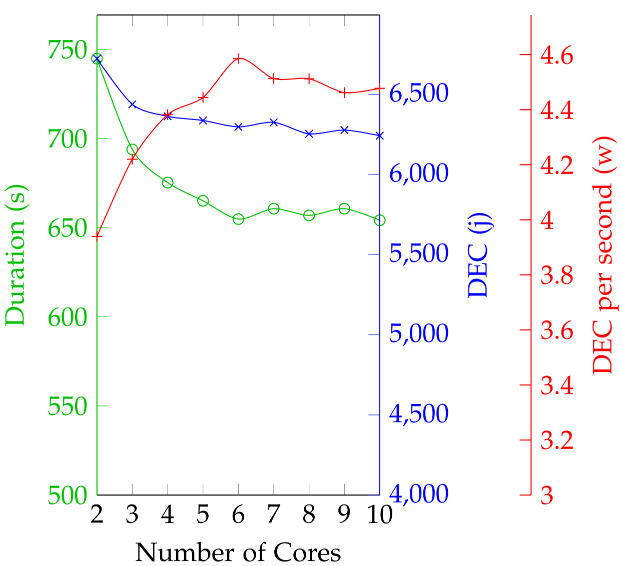


Figure 7: The evolution of the DEC (blue), DEC per second (red) and duration (green) as more cores are allocated to PCM on DUT 2. Note that the x- and y-axis does not start at zero.

Macrobenchmark Results: The results for DUT 1 can be seen in Figure **6** and Figure **7** for 3DM and PCM respectively, and for DUT 1 in Appendix **K**, while the results has been combined into a table for all DUTs and benchmarks in Appendix **L**. For both PCM and 3DM on both DUTs, similar observations can be made, where when more cores are allocated, the duration and DEC is exponentially decreasing, while the DEC per second is increasing. It can be seen in Table **19** that the duration decrease more than the DEC which shows a diminishing return in terms of energy savings. A difference between 3DM and PCM is how the duration and energy consumption decrease more for 3DM. This is because a large portion of PCM is single thread tasks, meaning only some parts of the benchmarks can benefit from the additional allocated cores, and even for those parts benefitting, the performance gained from the last few allocated cores is very limited, as can be seen in Table **19**. For 3DM, the benchmark itself is embarrassingly parallel, but measurements will include a startup and shutdown period, which means that the numbers reported in Figure **6** would be higher if the startup and shutdown periods were excluded. The diminishing return gained from allocating more resources to PCM is also illustrated, discussed and compared to 3DM in Appendix **N**.

P vs. E Initial Measurements: When running both macrobenchmarks on an increasing number of cores, starting from the most energy efficient one, the E cores were the last four. This showed that when comparing the energy consumption, it is higher compared to the P cores, given the higher duration. As was presented in subsection **3.2**, the point of a E cores are for small non-critical jobs. In this experiment, PCM will be run on four cores, either with four P cores (4P), four E cores (4E) or two of each (2P2E), to emulate a more realistic setting where E cores could flourish. This is because PCM will not utilize the entire CPU, meaning that the P and E cores could be used when the OS sees it fit. For this experiment, 30 initial measurements were made, and additional were made after Cochran's formula was applied to the results, if required. This can be found in

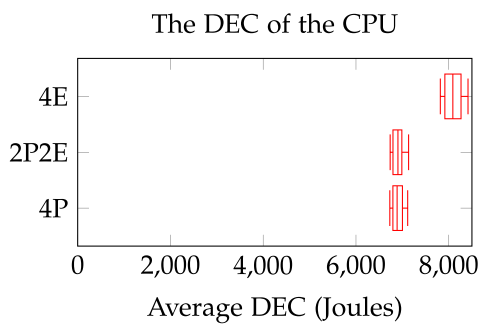


Figure 8: CPU measurements by IPG on DUT 2 for test case(s) PCM

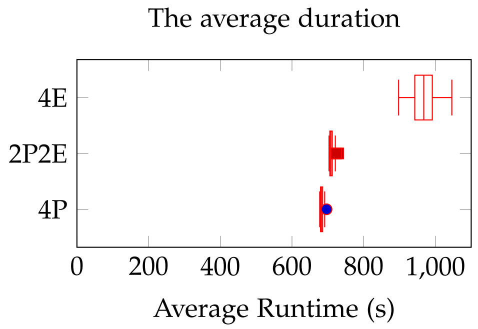


Figure 9: Runtime measurements by IPG on DUT 2 for test case(s) PCM

P vs. E Results: The results for the duration and DEC can be seen in Figure **9** and Figure **8** respectively, while the DEC per second can be seen in Appendix **M**. When looking at the DEC and duration, 4E has a higher duration and DEC, while 4P has the lowest. When combining P- and E cores, the duration was 3.8% higher and the DEC was 0.23% higher compared to 4P. While this still showed that P cores performed best, it also showed an almost equivalent performance despite two cores having a lower frequency.

6 Discussion

In the following section, results from Section **5** will be discussed.

6.1 Deviating Results

In this work Cochran's formula was used to determine how many measurements were required in order to gain confidence in our results. When analyzing the results from Cochran's formula, it was found that the amount of measurements could deviate a lot between benchmarks, measuring instrument, DUTs and even cores on the same DUT.

Results can deviate from core to core as a result of the variability in the fabrication process, where the exact characteristics of each core can change, despite being assembled in the same way.[46] In our work we explored how much the variability in the fabrication process can effect the performance of the cores, and found the energy consumption deviated between 1.17% and 11.61% among cores with the same specs.

When comparing the results from Cochran's formula between DUTs, DUT 2 required less measurements than DUT 1. The cause of this can be either software or hardware based. When setting up both DUTs, effort was put into ensuring all software was the same version. Both DUTs run on a fresh install of windows, and have the same software downloaded, and the same background processes are disabled, so it seems unlikely this is the cause. When comparing hardware, the two DUTs are from different generations of intel CPUs, released five years apart. DUT 1 is the older of the two, and has been used for multiple years, while DUT 2 is brand new. The different components of the two DUTs are also from different brands. Given DUT 1 is older and has been used more could mean it also deviates more, but this is something to explore in a future work.

6.2 C++ Benchmarks Analysis

In the first experiment in subsection **5.1**, different compilers were compared. This experiment found that both the energy consumption, and measurements required deviated between compilers. This was especially clear when comparing the oneAPI to the other compilers. The initial fear was that the benchmark would have been removed as dead code by the compiler, thus resulting in the low durations observed, but given that the executable from oneAPI (668 mb) was three times as large as for MinGW (220 mb) it seemed unlikely. The analysis was conducted by comparing the oneAPI against MinGW, by decompiling the executables and comparing the instructions.

When comparing the Assembly code structure between the main functions from the compilers, the oneAPI used several function calls unique for intel process such as `___intel_new_feature_proc_init` and `__intel_fast_memset`, where both functions are part of Intel's default C++ libraries[47]. The MinGW used general purpose registers more and utilizes C++ Standard Library, in the assembly more than oneAPI.

When moving on to the Mandelbrot function, where the differences with the larges effect on runtime are expected, Mingw only use standard X86 instruction[48] to perform the calculations on the floating points and `xmm` registers to store the values. Opposed to this, oneAPI's implementation also utilized Advance Vector extensions (AVX)[**AVX**], to perform the calculations. The usage of AVX results in a significant increase in the speed of calculations as it allows for multiple calculations to be performed in parallel. The AVX technology is a set of instructions introduced by Intel to enhance the performance of floating-point-intensive applications[**AVX**]. Compared to MinGW, oneAPI is better at optimizing the code for the AVX architecture and take full advantage of the processing power of the CPU.

The use of AVX to perform large floating point calculations in parallel resulting in a similar energy consumption but a lower duration is an observation also found by others studies about energy consumption and parallelism[9].

Besides the speed up gained from utilizing AVX, oneAPI also practice loop unrolling where no evidence is found of this in MinGW. This and the inclusion of extension libraries could be the reasons why the executable made by oneAPI is larger but with a lower runtime than MinGW and the the other compilers.

6.3 Energy usage trends

The trend shown in subsection **5.2** where the median DEC decreased as more measurements were taken until 1000 measurements after which it increased until 1400 after which it decreased again, could be seen on both the Clamp and Plug, which indicates that it is not caused by faulty measurements. However, the same trend could not be observed on the software-based measuring instruments. Therefore we hypothesized that the observed reduction in energy consumption may be caused by changes in the reactive energy[49] consumption occurring between the power outlet and the power supply of the DUTs.

In a circuit, two types of energy can be identified: active energy, which performs useful work, and reactive energy, which does not. The combination of these two energies is called apparent energy, which is what is measured by our hardware-based measuring instruments. Reactive energy occurs because of inductive or capacitive loads in a circuit, resulting in an energy loss that is not utilized by the circuit[49]. The ratio between active and reactive energy is known as the power factor[49].

Based on this, two hypotheses have been constructed to explain why the energy consumption of the DUTs are changing over measurements. Firstly, noise on the electrical network could interfere with the phase synchronization. This may be due to many machines being connected to the same electrical network, and disrupting the harmonics of the network[50]. However, if the power supply generates reactive energy because it is out of phase with the electrical network, a reduction in noise could help synchronize them again. Therefore, the observed changes in energy consumption may be related to the time of the day and week where the measurements are taken, with consumption decreasing when there is less devices connected to the electrical network, during the night and weekends.

Alternatively, the DUTs' PSU may be correcting the phase over time. PSU's can contain a power factor correction circuit that attempts to reduce the amount of reactive power by correcting the phase. There are two main types of power factor correction: passive and active [51]. The behavior seen in the results may be the result of an active power factor correction circuit. Unfortunately we were unable to determine if such a circuit was present in the PSU after we contacted both the manufactures of the PSUs used in the DUTs, but neither answered.

To try and confirm or reject these, smaller additional measurements were made to compare the measurements from night and day to see if the trends in the two are different to confirm or reject the first hypothesis the results shows that the seems to be an increasing trend during the day of 0.633, While during the night -1.288(WARNING THESE ARE BASED ON A SINGLE 24 hour cycle UPDATE LATER), a more detail description can be found in the appendixsubsection **N.1**. For the second hypothesis, a mail was writing to one of the producers of the power Supplies in the DUTs for further details, but no additional information was provided. We are unable to determine the exact cause of the changes in energy consumption.

Previous research in this field, which also utilizes hardware measurements, has not addressed this phenomenon. For example, [52], [45], and [53] did not report similar findings, although their studies might have had a similar environmental setting to ours. While these studies are not directly comparable, we would have anticipated some resemblance, indicating that previous research utilizing hardware measurements might not have been extensive enough, as this trend has not been revealed previously.

6.4 Time synchronization

In our work, four different devices were used to take the measurements - the DUTs, a Raspberry Pi, and an Analog Discovery 2. Each of these devices kept its own time, which could cause issues if they were not synchronized. This was particularly problematic for external measurement instruments, as even small differences in time could result in inaccurate data.

To address this issue, the data acquisition process was changed to ensure that the devices were synchronized every second. However, some problems may still exist, as small time drifts can occur over time. For example, the Raspberry Pi did not have a real-time clock[54] and would therefore become increasingly inaccurate over time. Additionally, the execution time of IO events for the clamp and plug could result in a slight time difference, although this is expected to have minimal impact on the results, since resynchronization happens every second.

6.5 Measuring Instruments

One thing to note is that our determination of accuracy is based of the accuracy of the Clamp, which means that if the Clamp is not accurate, then we do not know if the other measuring instruments are.

For IPG, a bluescreen issue was encoutered when using the API, similarly to [3], which occured when requesting new measurements ten times per second. The issue was most likely related to requesting measurements, when no data was available, despite Intel claiming IPG is able to sample 1000 times per second, but is something to explore in a future work. This issue was however resolved by using the GUI instead, which could be executed using the command `IntelPowerGadget.exe -start` and `IntelPowerGadget.exe -stop`.

6.6 P and E Cores

In subsection **5.3**, we examined the impact of P- and E-cores in DUT2, as outlined in RQ **4**. Our findings showed that E-cores had lower DEC per second but higher execution time and total DEC per benchmark. However, certain aspects of our experimental setup influenced these results.

Firstly, as detailed in subsection **3.2**, E-cores are designed for smaller, non-time critical tasks such as background services. Our benchmarks, which placed the cores under heavy load, did not align with this intended use. As a result, our comparison of E-cores and P-cores in terms of DEC does not accurately reflect their performance in real-life computing scenarios where E-cores might effectively handle smaller tasks.

Secondly, we disabled Intel Turbo Boost and C-states, causing core frequencies to remain static at their base clock of 1.8 GHz for E-cores and 2.5 GHz for P-cores. This decreased energy consumption per second under heavy load but increased it when idling, thus impacting our measurements.

Given the mismatch between our benchmarks and the intended use of E-cores, as well as the static core clock speeds, our experimental setup does not offer a realistic evaluation of the effects of P- and E-cores.

6.7 Windows

This work stands out compared to existing work, by its use of Windows over Linux[52, 53, 55]. Windows is interesting as it is a very popular OS, and because no study, to our knowledge, exists which looks into the measuring instruments and the energy consumption on Windows.

When comparing results between Linux and Windows in Appendix **F**, Windows was found to have a lower DEC, similar to what was found in [3]. One issue on Windows was finding compatible benchmarks. Because most studies are made on Linux, most micro- and macrobenchmarks are made for Linux, which does not guarantee they are compatible for Windows. This was a problem in the first experiment, where the benchmarks had to be compatible for all four compilers on Windows. The original idea was also to find macrobenchmarks written in C++, compiled on the most energy efficient compiler, which we were not able to find. Instead PCM and 3DM was chosen, where each had their own issues. For PCM, each DUT had some scenarios it was unable to run, making it difficult to compare the performance of the two DUTs. For 3DM, when starting multiple times after each other, loading times became increasingly large, until 3DM was restarted. These loading times did not effect the energy measurements, but meant the experiments took additional time. 3DM also caused bluescreen with stop code `VIDEO_TDR_FAILURE` on DUT 2 in rare cases, which was found go be GPU related issues on the `igdkmdn64.sys` process. Neither of the mentioned issues related to PCM or 3DM was resolved, but is something to explore in a future work.

One thing to be aware of when working on Windows over Linux is the additional background processes. It is important to disable startup processes, windows update and other processes running in the background. The exact effect the background processes has on the measurements is unknown and is something to look into in a future work.

6.8 C++ Benchmarks Analysis

In the first experiment in subsection **5.1**, different compilers were compared. This experiment found that both the energy consumption, and measurements required deviated between compilers. This was especially clear when comparing the oneAPI to the other compilers. The initial fear was that the benchmark would have been removed as dead code by the compiler, thus resulting in the low durations observed, but given that the executable from oneAPI (668 mb) was three times as large as for MinGW (220 mb) it seemed unlikely. The analysis was conducted by comparing the oneAPI against MinGW, by decompiling the executables and comparing the instructions.

When comparing the Assembly code structure between the main functions from the compilers, the oneAPI used several function calls unique for intel process such as `___intel_new_feature_proc_init` and `__intel_fast_memset`, where both functions are part of Intel's default C++ libraries[47]. The MinGW used general purpose registers more and utilizes C++ Standard Library, in the assembly more than oneAPI.

When moving on to the Mandelbrot function, where the differences with the larges effect on runtime are expected, Mingw only use standard X86 instruction[48] to perform the calculations on the floating points and `xmm` registers to store the values. Opposed to this, oneAPI's implementation also utilized Advance Vector extensions (AVX)[**AVX**], to perform the calculations. The usage of AVX results in a significant increase in the speed of calculations as it allows for multiple calculations to be performed in parallel. The AVX technology is a set of instructions introduced by Intel to enhance the performance of floating-point-intensive applications[**AVX**]. Compared to MinGW, oneAPI is better at optimizing the code for the AVX architecture and take full advantage of the processing power of the CPU.

The use of AVX to perform large floating point calculations in parallel resulting in a similar energy consumption but a lower duration is an observation also found by others studies about energy consumption and parallelism[9].

Besides the speed up gained from utilizing AVX, oneAPI also practice loop unrolling where no evidence is found of this in MinGW. This and the inclusion of extension libraries could be the reasons why the executable made by oneAPI is larger but with a lower runtime than MinGW and the the other compilers.

7 Conclusion

This work set out to explore parallelism and the effects it has on energy consumption, where Windows is used as the primary OS with Linux as a reference point, and Cochran's formula to ensure confidence in the reported numbers. With Cochrans formula, it is found that more measurements are required in order to gain confidence in measurements than are generally found in existing work. When looking at OSs, while Windows will provide valuable depth to any analysis about energy consumption as similar observations are not guaranteed between OSs, Linux is overall the easier choice in this domain. This is because Linux is a more minimalist OS with less pre-installed software and background processes to be aware of. In this domain, reaching definitive conclusions has proven to be challenging as the performance has found to be very OS, hardware and compiler dependent.

Since RAPL is not available on Windows, alternatives are compared on C++ microbenhmarks, compiled on the most energy efficient compiler. The most energy efficent C++ compiler is found through the first experiment, where tests shows that there is a big difference in performance between compilers. The most energy efficient compiler is found be Intel's oneAPI, because of its use of parallelism and YMM registers, only found on Intel CPUs.

When comparing microbenhmarks compiled on oneAPI between different measuring instruments for Windows, a similar performance is found when comparing each measuring instruments correlation with a ground truth obtained by a current clamp. This is expected the be the cause of all measuring instruments utilizing the same registers when reporting the energy consumption, but is something to look into in a future work. In the end, Intel's Intel Power Gadget is chosen, because of its usability compared to other measuring instruments. In a future work it could be interesting to extend this analysis to include things like the overhead of the measuring instruments for Windows, to see how they compare to RAPL.

In the third experiment, the performance of P- and E were analyzed, which showed a lower duration and DEC for P cores but a higher DEC per second compared to E cores. This shows that for most workloads, the P cores are preferred. The intended workload for E cores is small non-critical jobs, which is not covered in this work, which could be interesting in a future work.

In the third experiment parallelism and its effect on energy consumption is explored on two macrobenchmarks, this being PC-Mark-10 and 3D-Mark, where one represents a realistic usecase of videochats and web browsing, while the other simulates a more demanding workload. Both macrobenchmarks were executed on an increasing number of cores, to find what performance benefit the additional cores bring. For both macrobenchmarks a relationship is found between the DEC, duration and DEC per second, where as more cores are allocated, the duration and DEC decreases, while the DEC per second increases, with a diminishing return. The relationship is however nonlinear, where the duration decreases by more than the DEC which illustrates diminishing return. This diminishing return means that at a certain number of cores, additional cores will have no effect on the duration or the DEC, where this number of cores will be higher for a more demanding workloads.

Acknowledgements

References

1. Jones, N. *et al.* How to stop data centres from gobbling up the world's electricity. *Nature* **561,** 163–166 (2018).
2. Andrae, A. S. & Edler, T. On global electricity usage of communication technology: trends to 2030. *Challenges* **6,** 117–157 (2015).
3. Holt, J., Kusk, M. H. & Pedersen, J. B. *A Comparison Study of Measuring Instruments* (Aalborg University Department of Computer Science, 2023).
4. Amdahl, G. M. *Validity of the single processor approach to achieving large scale computing capabilities* in *Proceedings of the April 18-20, 1967, spring joint computer conference* (1967), 483–485.
5. Woo, D. H. & Lee, H.-H. S. Extending Amdahl's law for energy-efficient computing in the many-core era. *Computer* **41,** 24–31 (2008).
6. Prinslow, G. Overview of performance measurement and analytical modeling techniques for multi-core processors. *UR L: http://www. cs. wustl. edu/~ jain/cse567-11/ftp/multcore* (2011).
7. Pinto, G., Castor, F. & Liu, Y. D. Understanding Energy Behaviors of Thread Management Constructs. *SIGPLAN Not.* **49,** 345–360. ISSN: 0362-1340. https://doi.org/10.1145/2714064.2660235 (2014).
8. Abdelhafez, A., Alba, E. & Luque, G. A component-based study of energy consumption for sequential and parallel genetic algorithms. *The Journal of Supercomputing* **75,** 1–26 (Oct. 2019).
9. Lindholt, R. S., Jepsen, K. & Nielsen, A. Ø. *Analyzing C# Energy Efficiency of Concurrency and Language Construct Combinations* (Aalborg University Department of Computer Science, 2022).
10. Hassan, H., Moussa, A. & Farag, I. Performance vs. Power and Energy Consumption: Impact of Coding Style and Compiler. *International Journal of Advanced Computer Science and Applications* **8** (Dec. 2017).
11. Saez, J. C. & Prieto-Matias, M. *Evaluation of the Intel thread director technology on an Alder Lake processor* in *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems* (2022), 61–67.
12. Intel. *Intel performance hybrid architecture & software optimizations Development Part Two: Developing for Intel performance hybrid architecture* https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwj1j9no9c79AhX9S_EDHXGiDMgQFnoECA8QAQ&url=https%3A%2F%2Fcdrdv2-public.intel.com%2F685865%2F211112_Hybrid_WP_2_Developing_v1.2.pdf&usg=AOvVaw2dfqExqLBgFMeS5To1sjKM. 09/03/2023.
13. Intel. *Energy-Efficient Platforms* https://www.intel.com/content/dam/develop/external/us/en/documents/green-hill-sw-20-185393.pdf. 2011. 07/03/2023.
14. hardwaresecrets. *Everything You Need to Know About the CPU Power Management* https://hardwaresecrets.com/everything-you-need-to-know-about-the-cpu-c-states-power-saving-modes/. 2023. 07/03/2023.
15. ARM. *MEDIA ALERT: ARM big.LITTLE Technology Wins Linley Analysts' Choice Award* https://www.arm.com/company/news/2012/01/media-alert-arm-biglittle-technology-wins-linley-analysts-choice-award. 2012. 09/03/2023.
16. ARM. *Processing Architecture for Power Efficiency and Performance* https://www.arm.com/technologies/big-little. 09/03/2023.
17. Intel. *Intel Unveils 12th Gen Intel Core, Launches World's Best Gaming Processor, i9-12900K* https://www.intel.com/content/www/us/en/newsroom/news/12th-gen-core-processors.html. 2021. 09/03/2023.
18. Rotem, E. *et al.* Intel alder lake CPU architectures. *IEEE Micro* **42,** 13–19 (2022).
19. *1.3.1. processor affinity or CPU pinning* https://www.intel.com/content/www/us/en/docs/programmable/683013/current/processor-affinity-or-cpu-pinning.html. 03/03/2023.
20. Karl-Bridge-Microsoft. *Scheduling priorities - win32 apps* https://learn.microsoft.com/en-us/windows/win32/procthread/scheduling-priorities. 03/03/2023.
21. Michael Womack, A. W. *Parallel Programming and Performance Optimization With OpenMP* https://passlab.github.io/OpenMPProgrammingBook/cover.html. 03/03/2023.
22. Mozilla. *tools/power/rapl* https://firefox-source-docs.mozilla.org/performance/tools_power_rapl.html. 24/02/2023.
23. source, O. *Libre Hardware Monitor* https://github.com/LibreHardwareMonitor/LibreHardwareMonitor. 03/03/2023.
24. Arnoux, C. *AC current clamps user's manual* 2013.
25. CloudFree. *CloudFree EU Smart Plug* https://cloudfree.shop/product/cloudfree-eu-smart-plug/. 10/03/2023.
26. Hubblo. *Scaphandre* https://github.com/hubblo-org/scaphandre. 23/02/2023.

27. Hubblo. *windows-rapl-driver* https://github.com/hubblo-org/windows-rapl-driver. 23/02/2023.
28. Fahad, M., Shahid, A., Manumachu, R. R. & Lastovetsky, A. A comparative study of methods for measurement of energy of computing. *Energies* **12,** 2204 (2019).
29. Guilford, J. P. Fundamental statistics in psychology and education (1950).
30. Razali, N. M., Wah, Y. B., *et al.* Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests. *Journal of statistical modeling and analytics* **2,** 21–33 (2011).
31. Mann, H. B. & Whitney, D. R. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics,* 50–60 (1947).
32. Han, A. K. Non-parametric analysis of a generalized regression model: the maximum rank correlation estimator. *Journal of Econometrics* **35,** 303–316 (1987).
33. Cochran, W. G. *Sampling Techniques, 3rd edition* ISBN: 0-471-16240-X (John Wiley & sons, 1977).
34. Saqib, M. *What are the Best C++ Compilers to use in 2023* 2023. https://www.mycplus.com/tutorials/cplusplus-programming-tutorials/what-are-the-best-c-compilers-to-use-in-2023/. 20/03/2023.
35. Pedamkar, P. *Best C++ Compiler* 2023. https://www.educba.com/best-c-plus-plus-compiler/. 20/03/2023.
36. *Top 22 Online C++ Compiler Tools* 2023. https://www.softwaretestinghelp.com/best-cpp-compiler-ide/. 20/03/2023.
37. *Clang Compiler Users Manual* https://clang.llvm.org/docs/UsersManual.html. 20/03/2023.
38. *MinGW FAQ* https://home.cs.colorado.edu/~main/cs1300/doc/mingwfaq.html. 20/03/2023.
39. *Intel oneAPI Base Toolkit* https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit.html#:~:text=The%20Intel%C2%AE%20oneAPI%20Base,of%20C%2B%2B%20for%20heterogeneous%20computing.. 20/03/2023.
40. *C and C++ in Visual Studio* 2022. https://learn.microsoft.com/en-us/cpp/overview/visual-cpp-in-visual-studio?view=msvc-170. 20/03/2023.
41. appfolio. *Microbenchmarks vs Macrobenchmarks (i.e. What's a Microbenchmark?)* https://engineering.appfolio.com/appfolio-engineering/2019/1/7/microbenchmarks-vs-macrobenchmarks-ie-whats-a-microbenchmark. 24/03/2023.
42. UL. *3DMark* https://www.3dmark.com/. 14/04/2023.
43. UL. *PCMark* https://benchmarks.ul.com/pcmark10. 14/04/2023.
44. *Central Limit Theorem* https://sphweb.bumc.bu.edu/otlt/mph-modules/bs/bs704_probability/BS704_Probability12.html. 20/03/2023.
45. Koedijk, L. & Oprescu, A. *Finding Significant Differences in the Energy Consumption when Comparing Programming Languages and Programs* in *2022 International Conference on ICT for Sustainability (ICT4S)* (2022), 1–12.
46. Mauzy, R. Parallelizing Neural Networks to Break Physical Unclonable Functions (2020).
47. *Intel Default Libraries* https://www.cita.utoronto.ca/~merz/intel_c10b/main_cls/mergedProjects/bldaps_cls/cppug_ccl/bldaps_def_libs_cl.htm.
48. *x86 and amd64 instruction reference* https://www.felixcloutier.com/x86/.
49. *MORE POWER WITH LESS COPPER* https://fortop.co.uk/knowledge/white-papers/reactive-power-reducing-compensating/.
50. Kullarkar, V. T. & Chandrakar, V. K. Power quality analysis in power system with non linear load. *Int. J. Electr. Eng* **10,** 33–45 (2017).
51. McDonald, B. & Lough, B. *Power Factor Correction (PFC) Circuit Basics* in *Texas Instruments Power Supply Design Seminar* (2020).
52. Georgiou, S. & Spinellis, D. Energy-delay investigation of remote inter-process communication technologies. *Journal of Systems and Software* **162,** 110506 (2020).
53. Khan, K. N., Hirki, M., Niemi, T., Nurminen, J. K. & Ou, Z. RAPL in Action: Experiences in Using RAPL for Power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* **3,** 1–26 (2018).
54. *Syncing Time from the Network on the Raspberry Pi* = https://pimylifeup.com/raspberry-pi-time-sync/. 2022.
55. Pereira, R. *et al. Energy efficiency across programming languages: how do energy, time, and memory relate?* in (Oct. 2017), 256–267.
56. Igor Wallossek, A. M. *Picking The Right Power Supply: What You Should Know* https://www.tomshardware.com/reviews/psu-buying-guide,2916-3.html. 2016.

A Abbreviations

On Table **10** a list of all the terms which are abbreviated in this work can be found. They are alphabetically sorted within their categories. Their first occurrence can also be seen.

Abbreviations used in this work		
General Technology and Hardware Terms	**Abbreviation**	**First Occurrence**
Device Under Test	DUT	Section **1**
Efficiency core	E-core	Section **1**
Information and Communications Technology	ICT	Section **1**
Operating System	OS	subsection **2.4**
Performance core	P-core	Section **1**
Power supply unit	PSU	subsection **4.1**
Measuring Instruments	**Abbreviation**	**First Occurrence**
Clamp Linux	Clamp (L)	subsection **5.2**
Clamp Windows	Clamp (W)	subsection **5.2**
CloudFree EU smart Plug	Plug	subsection **4.1**
Intel Power Gadget	IPG	subsection **2.1**
Libre Hardware Monitor	LHM	subsection **2.1**
MN60 AC Current Clamp	Clamp	subsection **4.1**
Plug Linux	Plug (L)	subsection **5.2**
Plug Windows	Plug (W)	subsection **5.2**
Running Average Power Limit	RAPL	subsection **2.1**
Scaphandre	Scap	subsection **4.1**
Scaphandre isolated	SCAPI	subsection **4.1**
Benchmarks	**Abbreviation**	**First Occurrence**
3DMark	3DM	subsection **4.6**
Fannkuch-Redux	FR	subsection **4.6**
Mandelbrot	MB	subsection **4.6**
Nbody	NB	subsection **4.6**
PCMark 10	PCM	subsection **4.6**
Spectra-Norm	SN	subsection **4.6**
Compilers	**Abbreviation**	**First Occurrence**
Intel's oneAPI C++	oneAPI	subsection **4.5**
Microsoft Visual C++	MSVC	subsection **4.5**
Minimalist GNU for Windows	MinGW	subsection **4.5**
Energy Consumption Terms	**Abbreviation**	**First Occurrence**
Dynamic Energy Consumption	DEC	subsection **4.2**
Other terms	**Abbreviation**	**First Occurrence**
Biks Diagnostics Energy	BDE	Appendix **B**
Intel's Thread Director	ITD	subsection **2.4**
Model-specific-registers	MSRs	subsection **4.1**
Open Multi-Processing	OpenMP	subsection **3.5**
Performance Monitoring Counter	PMC	subsection **2.4**
Speedup Factor	SF	subsection **2.4**

Table 10: Abbreviations used in this work and their first occurrences. In alphabetical order.

B The Framework

The framework used in this work is an extension to [3], where one key difference is it a command line tool, supporting all languages. The framework is called Biks Diagnostics Energy (BDE) and can be executed in two ways, as seen in Listing **4**, where one is with a configuration, and one is with a path to an executable file.

```
.\BDEnergyFramework --config path/to/config.json

.\BDEnergyFramework --path path/to/file.exe --parameter parameter
```

Listing 4: An example of how BDE can be started

When using `--config`, the user specifies a path to a valid json file of the format seen in Listing **5**. Through Listing **5**, it is possible to specify paths to executable files and assign each executable file with a parameter in `BenchmarkPaths` and `BenchmarkParameter` respectively. Information like the `compiler`, `language`, etc can also be specified about the benchmark in the configuration. It is also possible to specify the affinity of the benchmark through `AllocatedCores`, where an empty list represents the use of all cores and the list `1,2` specifies how the benchmark can only execute on core one and two. When multiple affinities are specified, each benchmark will be run on both. Limits for the temperature the benchmarks should be executed within can also be specified, and lastly, `AdditionalMetadata` can be used to specify relevant aspects about the experiment, which cannot already be specified through the configuration.

```
[
  {
    "MeasurementInstruments": [ 2 ],
    "RequiredMeasurements": 30,
    "BenchmarkPaths": [
        "path/to/one.exe", "path/to/two.exe"
    ],
    "AllocatedCores": [
        [], [1,2]
    ],
    "BenchmarkParameters": [
        "one_parameter", "two_parameter",
    ],
    "UploadToDatabase": true,
    "BurnInPeriod": 0,
    "MinimumTemperature": 0,
    "MaximumTemperature": 100,
    "DisableWifi": false,
    "ExperimentNumber": 0,
    "ExperimentName": "testing-phase",
    "ConcurrencyLimit": "multi-thread",
    "BenchmarkType": "microbenchmarks",
    "Compiler": "clang",
    "Optimizations": "openmp",
    "Language": "c++",
    "StopBackgroundProcesses" : false,
    "AdditionalMetadata": {}
  }
]
```

Listing 5: An example of a valid configuration for BDE

When using the parameters `--path`, the `--parameter` is an optional way to provide the executable with parameters. When using BDE this way, a default configuration is set up, containing all fields in the configuration, except `BenchmarkPath` and `BenchmarkParameter`.

```
public interface IDutService
{
    public void DisableWifi();
    public void EnableWifi();
    public List<EMeasuringInstrument> GetMeasuringInstruments();
    public string GetOperatingSystem();
    public double GetTemperature();
    public bool IsAdmin();
    public void StopBackgroundProcesses();
}
```

Listing 6: The DUT interface which allows BDE to work on multiple OSs

Both Windows and Linux is supported on BDE. This is supported through the `IDutService` seen in Listing **6**, where all OS dependent operations are located. This includes the ability to enable and disable the WiFi, stop background processes, ect. The `IDutService` has a Windows and Linux implementation on BDE where depending on the OS of the machine BDE is executed on, one of these will be initialized and used.

```
1  public class MeasuringInstrument
2  {
3
4      public (TimeSeries, Measurement) GetMeasurement()
5      {
6          var path = GetPath(_measuringInstrument, fileCreatingTime);
7          return ParseData(path);
8      }
9
10     public void Start(DateTime fileCreatingTime)
11     {
12         var path = GetPath(_measuringInstrument, fileCreatingTime);
13
14         StartMeasuringInstruments(path);
15
16         StartTimer();
17     }
18
19     public void Stop(DateTime date)
20     {
21         StopTimer();
22         StopMeasuringInstrument();
23     }
24
25     internal virtual int GetMilisecondsBetweenSamples()
26     {
27         return 100;
28     }
29
30     internal virtual (TimeSeries, Measurement) ParseData(string path) { }
31
32     internal virtual void StopMeasuringInstrument() { }
33
34     internal virtual void StartMeasuringInstruments(string path) { }
35
36     internal virtual void PerformMeasuring() { }
37 }
```

Listing 7: The implementation of the different measuring instruments on BDE

BDE also supports multiple measuring instruments, through a parent class `MeasuringInstrument` in Listing **7** the measuring instruments can inherit from. `MeasuringInstrument` implements a start (line 10) and stop (line 19) method, and a method to get the data measured between the start and stop in line 4. In terms of the virtual methods, each measuring instrument needs to override, these are measuring instruments specific. This includes a start (line 34) and stop (line 32) method, a method to parse the measurement data in line 30 and a method in line 36 which performs a measurement by default every 100ms by default. The method in line 36 is made for measuring instruments line RAPL, where an action is required to read the energy consumption.

```
1  public void PerformMeasurement(MeasurementConfiguration config)
2  {
3      var measurements = new List<MeasurementContext>();
4      var burninApplied = SetIsBurninApplies(config);
5
6      if (burninApplied)
7          measurements = InitializeMeasurements(config, _machineName);
8
9      do
10     {
11         if (CpuTooHotOrCold(config))
12             Cooldown(config);
13
14         if (config.DisableWifi)
15             _dutService.DisableWifi();
16
17         PerformMeasurementsForAllConfigs(config, measurements);
18
19         if (burninApplied && config.UploadToDatabase)
20             UploadMeasurementsToDatabase(config, measurements);
21
22         if (!burninApplied && IsBurnInCountAchieved(measurements, config))
23         {
24             measurements = InitializeMeasurements(config, _machineName);
25             burninApplied = true;
26         }
27
28     } while (!EnoughMeasurements(measurements));
29 }
```

Listing 8: An example of how BDE performs measurements

Listing 8 shows how BDE performs measurements given the configuration. In the configuration, the burn-in period can be set to any positive integer, where if this value is one, the boolean `burninApplied` will be set to `true`, and the measurements will be initialized in line 7. This initialization will, if the results should be uploaded to the database, mean BDE will fetch existing results from the database, where the configuration is the same, and continue where it was left off. Otherwise, an empty list will be returned. If `burninApplied` is set to `false`, the amount of burn-in specified in the configuration will be performed before initializing the measurements.

Next, a do-while loop is entered in line 9, which will execute until the condition EnoughMeasurements from line 28 is met. Inside the do-while loop, a cooldown will occur in line 12, until the DUT is below and above the temperature limits specified in the configuration. Once this is achieved, the WiFi/Ethernet is disabled, and `PerformMeasurementsForAllConfigs` will then iterate over all measuring instruments and benchmarks specified, and perform one measurement for all permutations. Afterward, a few checks are made. If the burn-in period is over, and the configuration states that the results should be uploaded to the database, `UploadMeasurementsToDatabase` is called. If the burn-in period is not over yet, but `IsBurnInCountAchieved` is true, the measurements are initialized similarly to line 7, and the boolean *burninIsApplied* is set to `true`, indicating that the burn-in period is over, and the measurements are about to be taken.

C The Database

In [3], a MySQL database was used to store the measurements made by the different measuring instruments. In this work, a similar database will be used, but with some modifications to accommodate the different focus compared to [3]. The design of the database can be seen in Figure **10**, where the `MeasurementCollection` table defines under which circumstances the measurements were made. This includes which measuring instrument was used, which benchmark was running, which DUT the measurements were made on, whether or not there was a burn-in period, etc. Compared to [3], a few extra columns have been added to `Benchmark`, this includes metadata like compiler, optimizations, and parameters used.

In the `MeasurementCollection`, the columns `CollectionNumber` and `Name` represents which experiment the measurement is from, and the name of the experiment respectively. A column found in both `MeasurementCollection`, `Measurement` and `Sample` is `AdditionalMetadata`. This column can be used to set values unique for specific rows, where an example could be how some metrics are only measured by one measuring instrument.

The `Measurement` contains values for the energy consumption during the entire execution time of one benchmark, while the `Sample` represents samples taken during the execution of the benchmark. This means for one row in the `MeasurementCollection` table, there can exist one to many rows in `Measurement`. Each row in `Measurement` is associated with multiple rows in the `Sample` table, where the samples will be a time-series illustrating the energy consumption over time.

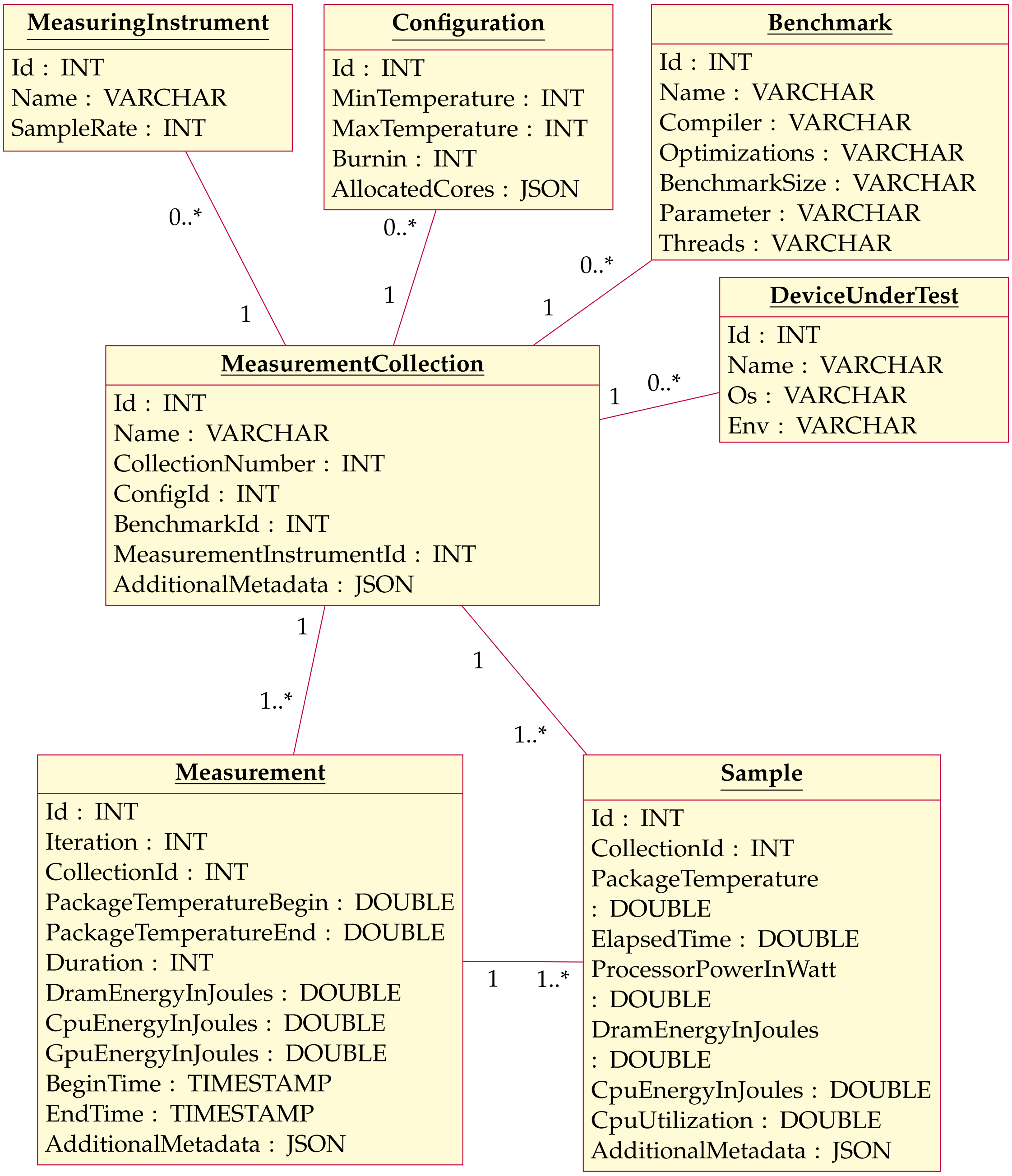


Figure 10: An UML diagram representing the tables in the SQL database

D PCMark 10

Not all of the benchmarks are executed on our DUTs, because some of them would crash. The different workloads and whether they are included or not are shown on Tables **11** and **12**. Further detail about the workloads can be found in [43].

Essentials		**Productivity**		**Digital Content Creation**	
App Start-up		**Writing**		**Photo Editing**	
Chromium	×	Writing simulation	×	Editing one photo	✓
Firefox	×			Editing a batch of photos	✓
LibreOffice Writer	×				
GIMP	×				
Web Browsing		**Spreadsheets**		**Video Editing**	
Social media	×	Common use	✓	Downscaling	✓
Online shopping	×	Power use (More complex)	✓	Sharpening	✓
Map	×			Deshaking filtering	✓
Video 1080p	×				
Video 2160p	×				
Video Conferencing				**Rendering and Visualization**	
Private call	✓			Visualization of a 3D model	✓
Group call	✓			Calculating a simulation	✓

Table 11: List of PCM benchmarks used on DUT1.

Essentials		**Productivity**		**Digital Content Creation**	
App Start-up		**Writing**		**Photo Editing**	
Chromium	×	Writing simulation	×	Editing one photo	×
Firefox	×			Editing a batch of photos	×
LibreOffice Writer	×				
GIMP	×				
Web Browsing		**Spreadsheets**		**Video Editing**	
Social media	✓	Common use	×	Downscaling	✓
Online shopping	✓	Power use (More complex)	×	Sharpening	✓
Map	✓			Deshaking filtering	✓
Video 1080p	✓				
Video 2160p	✓				
Video Conferencing				**Rendering and Visualization**	
Private call	✓			Visualization of a 3D model	✓
Group call	✓			Calculating a simulation	✓

Table 12: List of PCM benchmarks used on DUT2.

E Experiment One

Measurements made on benchmark MB for the first experiment, found in subsection **5.1**.

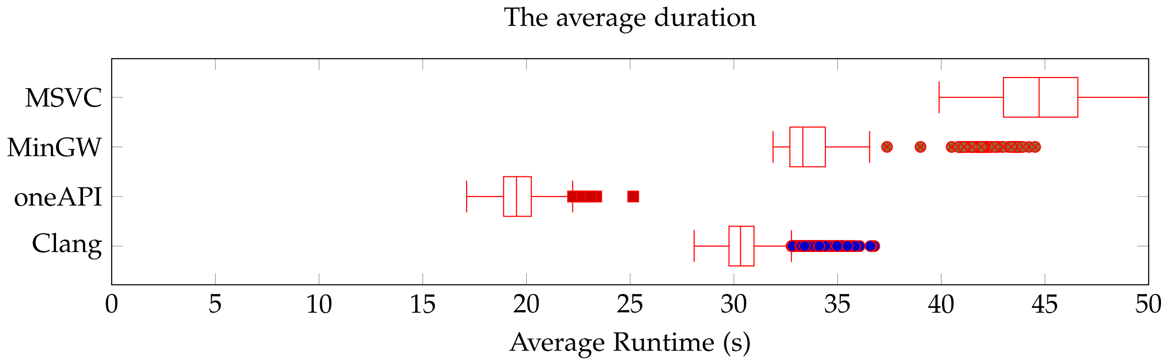


Figure 11: Runtime measurements by IPG on DUT 1 for benchmark(s) FR

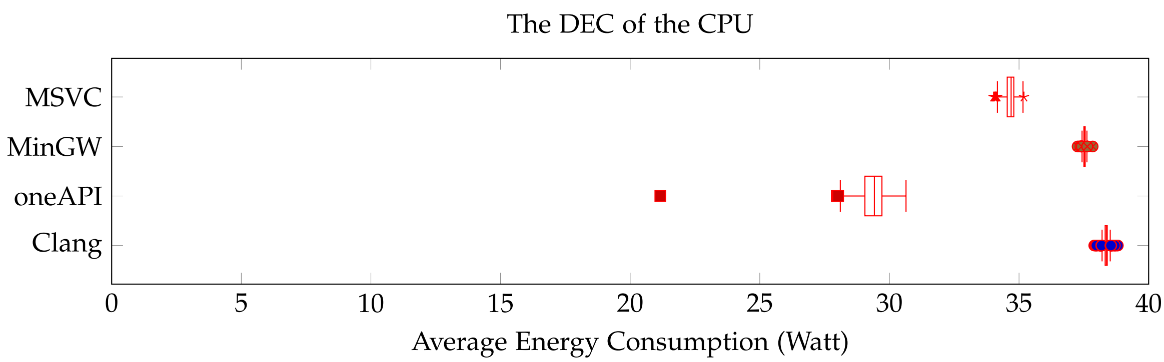


Figure 12: CPU measurements by IPG on DUT 1 for benchmark(s) MB

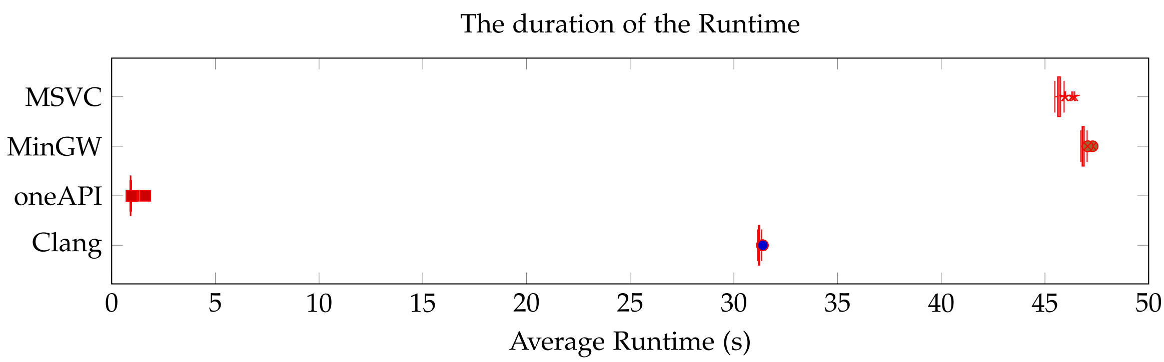


Figure 13: Runtime measurements by IPG on DUT 1 for benchmark(s) MB

F Experiment Two

Measurements made on for the second experiment, aiming to find the best measuring instrument, found in subsection **5.2**.

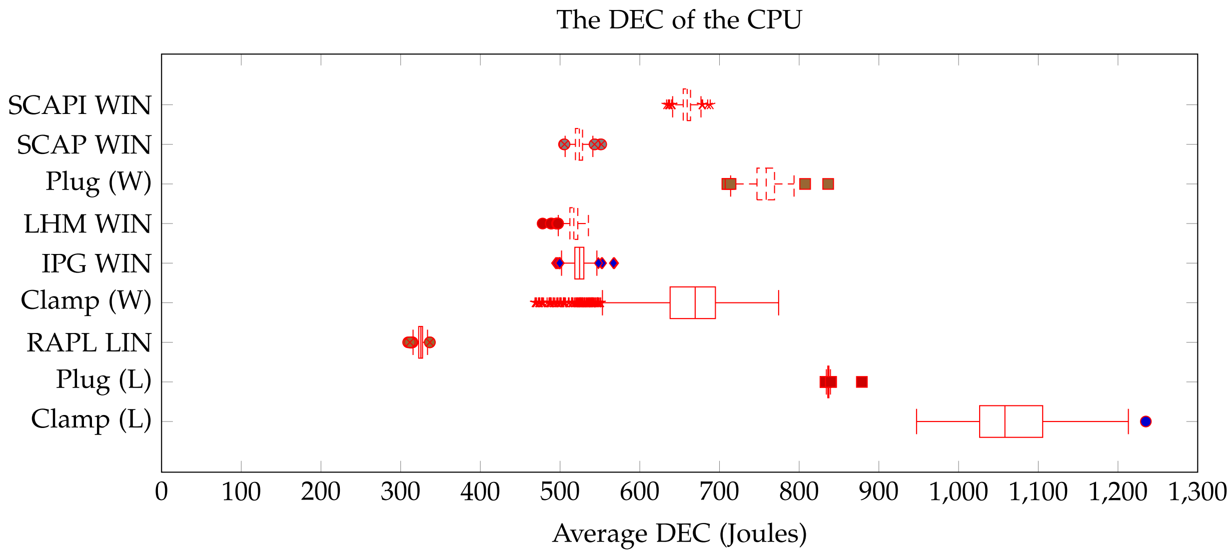


Figure 14: CPU measurements on DUT 1 for test case(s) FR compiled on oneAPI

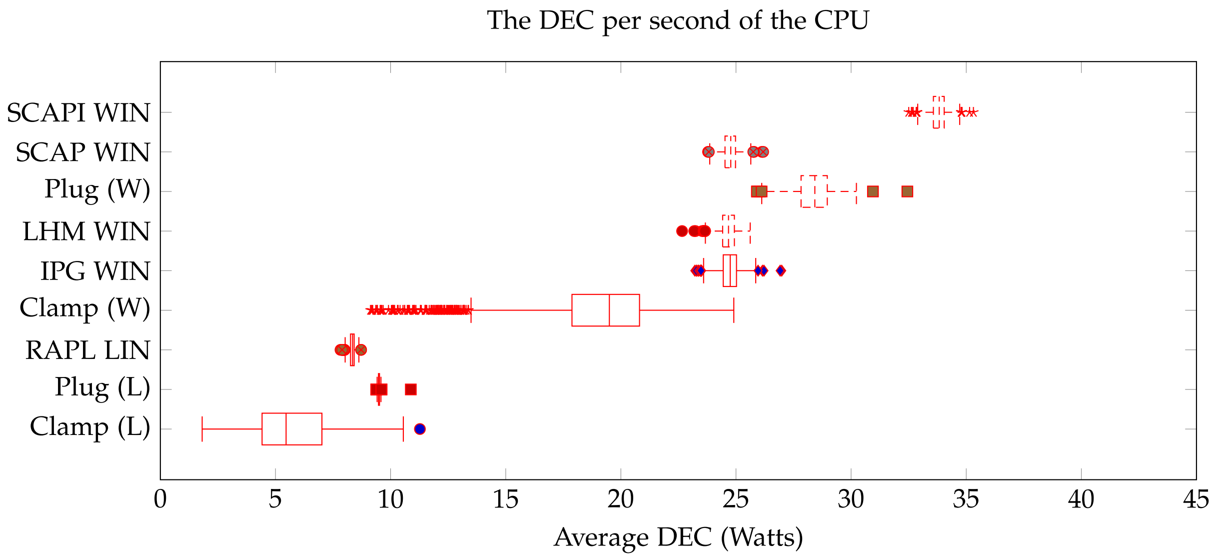


Figure 15: CPU measurements on DUT 1 for test case(s) FR compiled on oneAPI

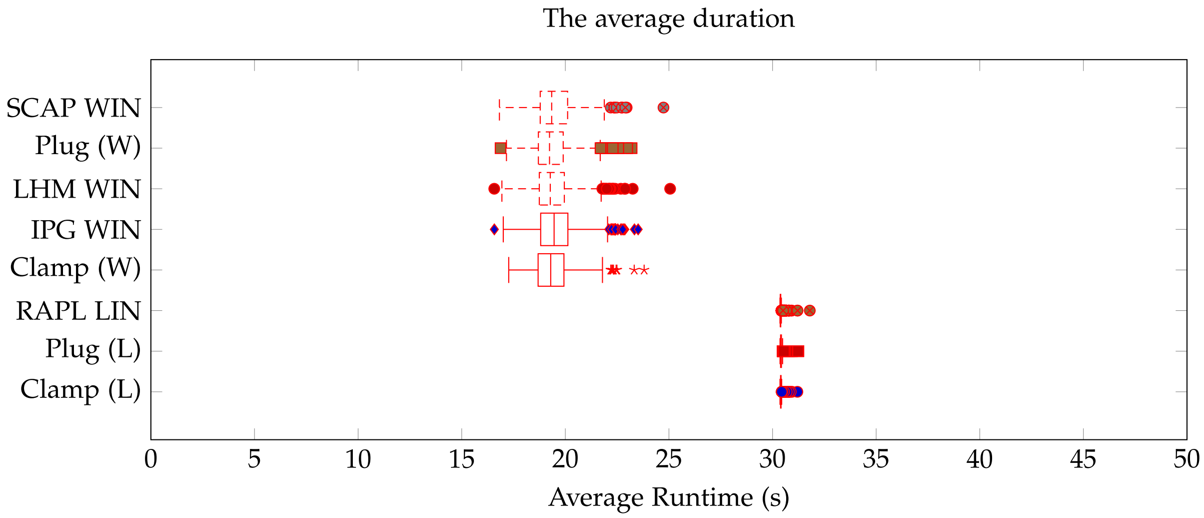


Figure 16: Runtime measurements on DUT 1 for test case(s) FR compiled on oneAPI

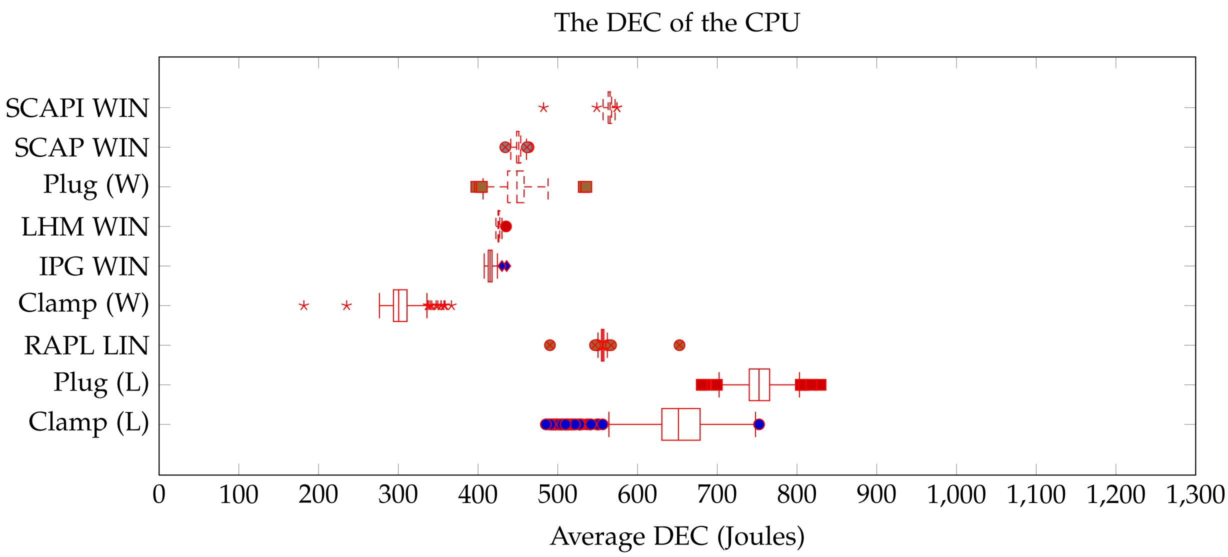


Figure 17: CPU measurements on DUT 1 for test case(s) MB compiled on oneAPI

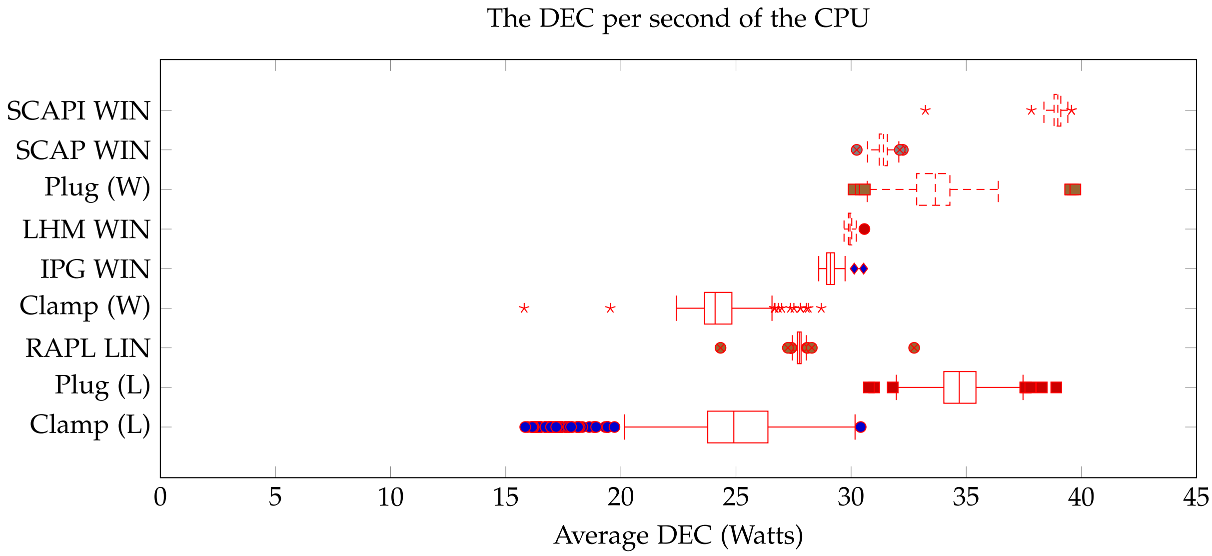


Figure 18: CPU measurements on DUT 1 for test case(s) MB compiled on oneAPI

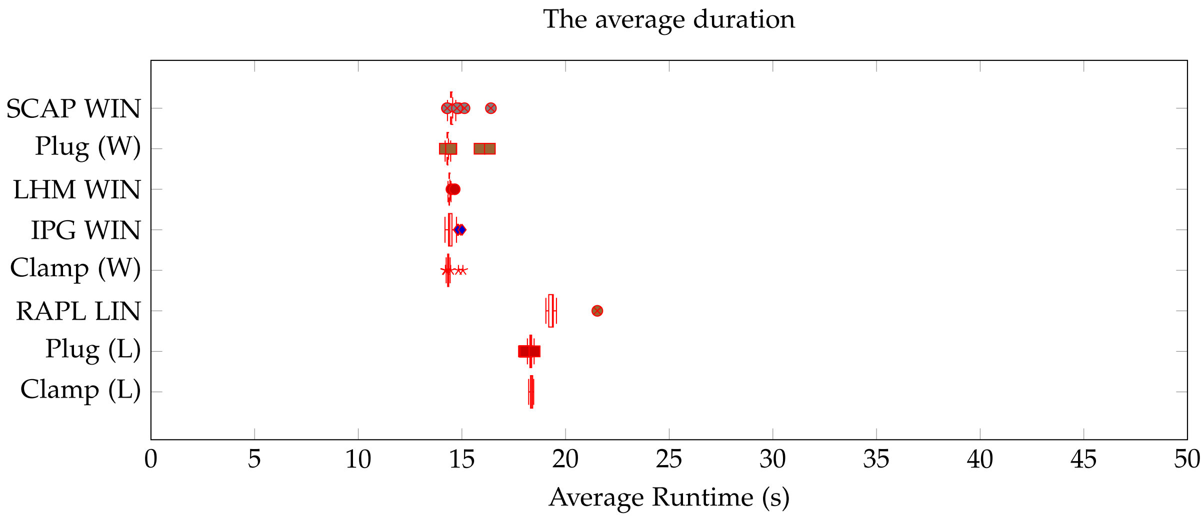


Figure 19: Runtime measurements on DUT 1 for test case(s) MB compiled on oneAPI

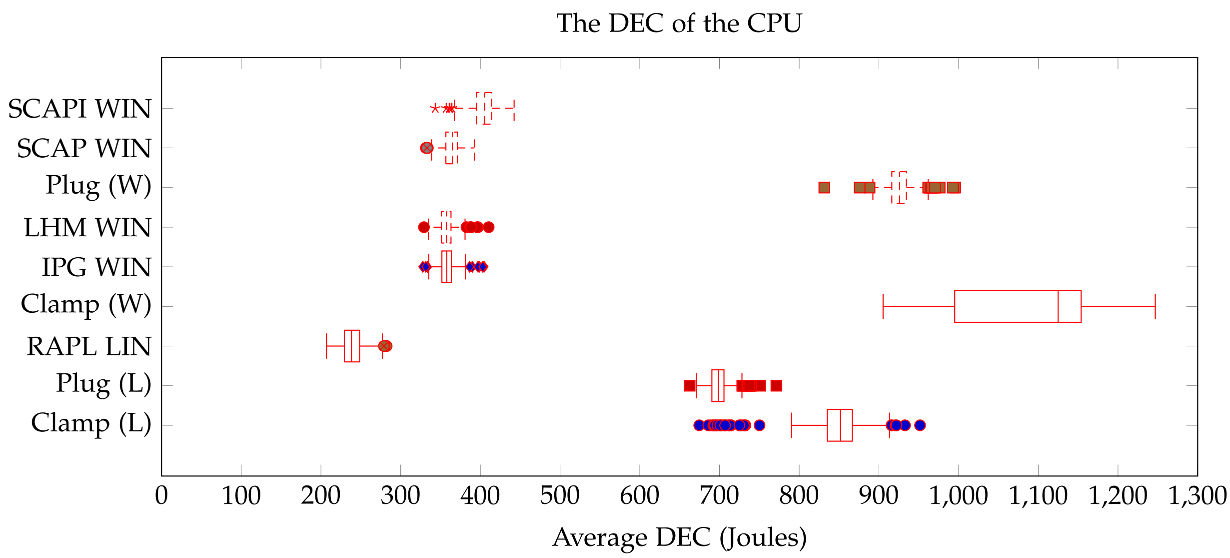


Figure 20: CPU measurements on DUT 2 for test case(s) FR compiled on oneAPI

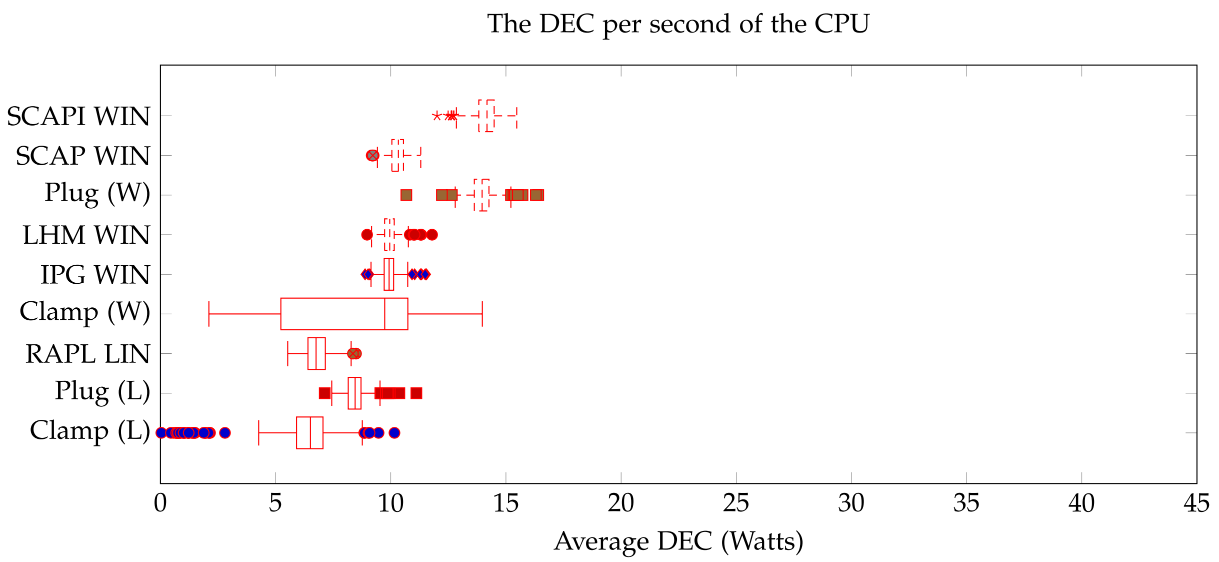


Figure 21: CPU measurements on DUT 2 for test case(s) FR compiled on oneAPI

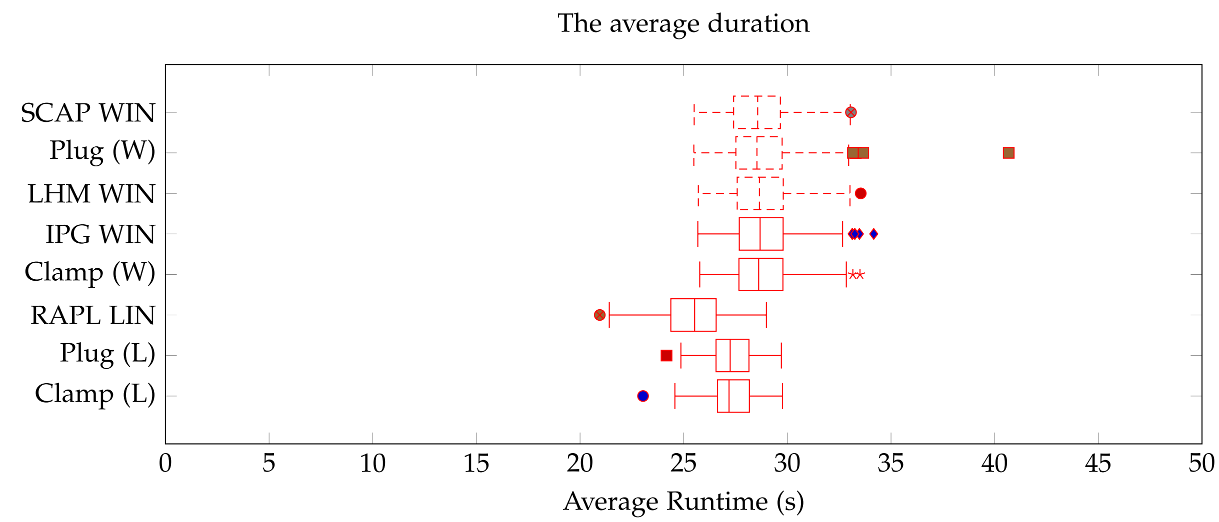


Figure 22: Runtime measurements on DUT 2 for test case(s) FR compiled on oneAPI

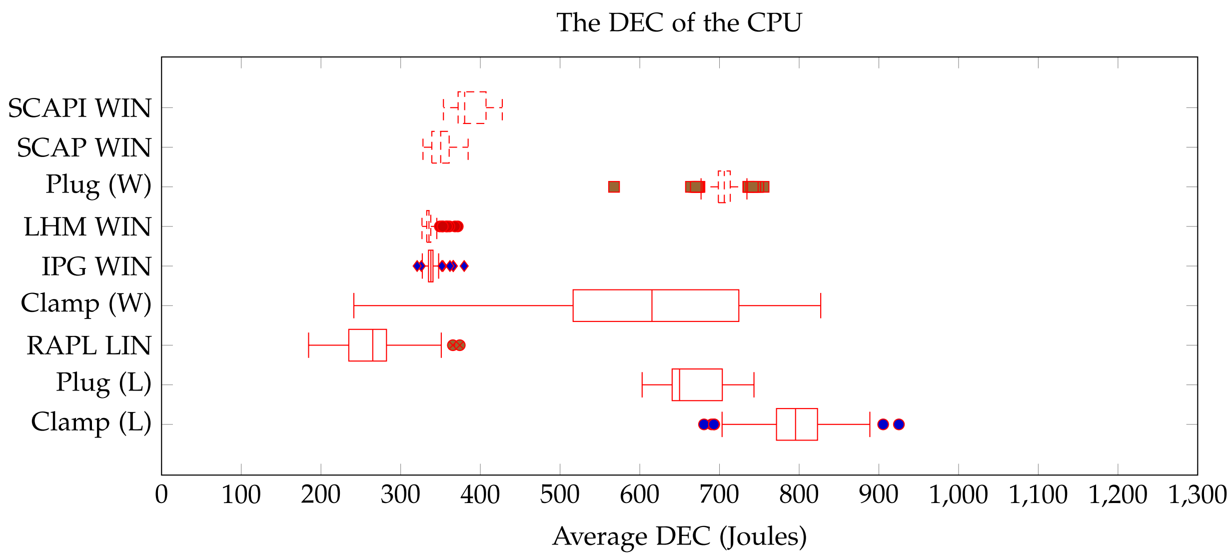


Figure 23: CPU measurements on DUT 2 for test case(s) MB compiled on oneAPI

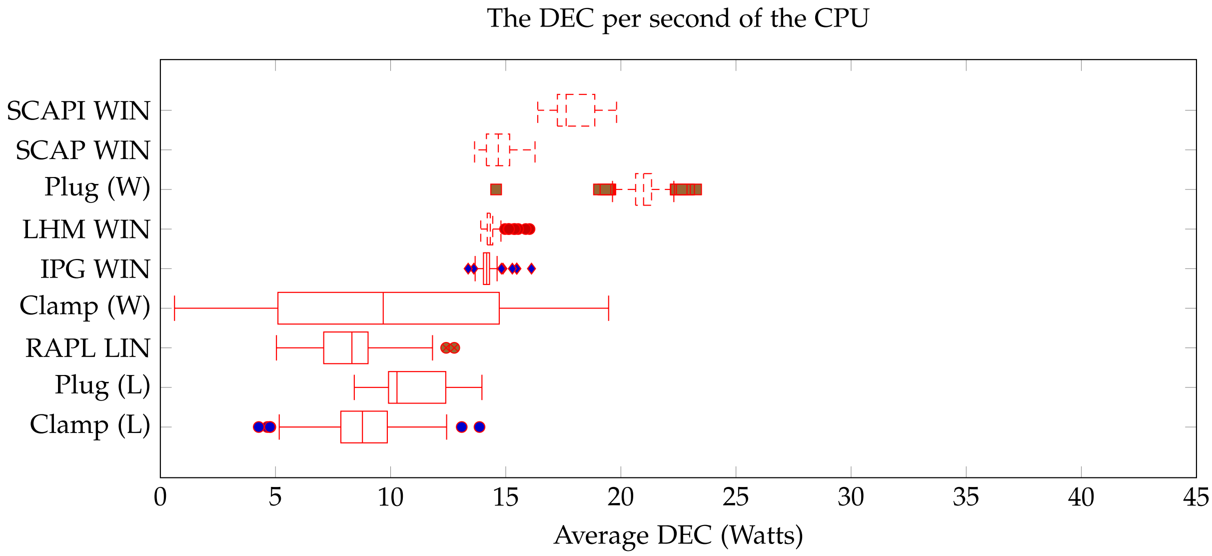


Figure 24: CPU measurements on DUT 2 for test case(s) MB compiled on oneAPI

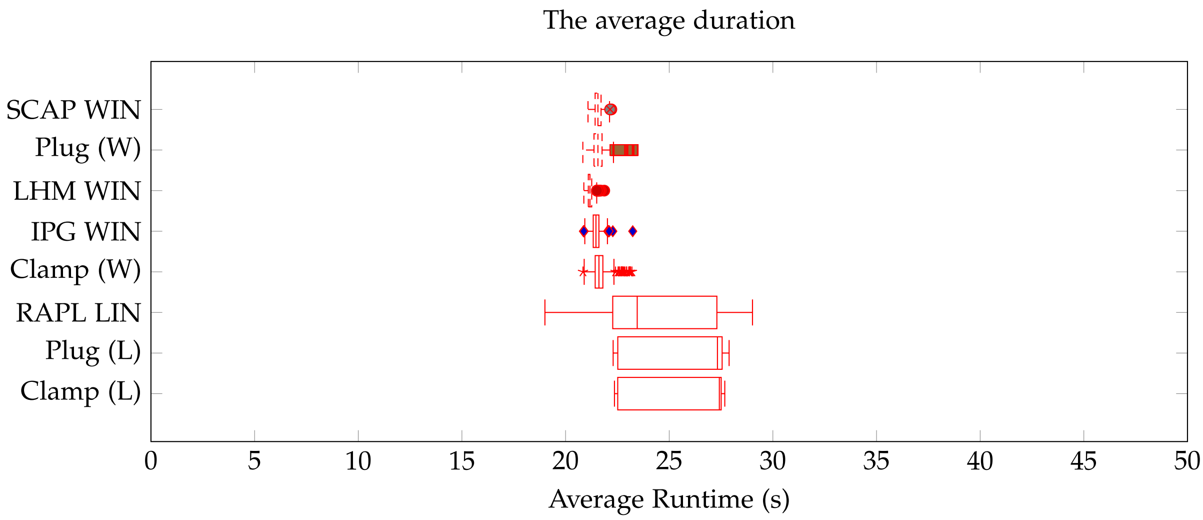


Figure 25: Runtime measurements on DUT 2 for test case(s) MB compiled on oneAPI

G Initial Measurements in Experiment Two

This section illustrates how many measurements are required from the different measuring instruments in order to gain confidence in them, and are used in subsection **5.2**.

Initial Measurements		
Name	FR	MB
Plug (W)	2.474	1.790
Plug (L)	5	4.818
Clamp (W)	16.908	2.855
Clamp (L)	12.837	11.518
RAPL	52	53
SCAP	459	74
SCAPI	453	153
IPG	714	216
LHM	604	45

Table 13: The required samples to gain confidence in the measurements made by the different measuring instruments, on both OSs for DUT 1

Initial Measurements		
Name	FR	MB
Plug (W)	916	1.088
Plug (L)	738	1056
Clamp (W)	36.558	44.106
Clamp (L)	2.869	7.021
RAPL	1.298	4.340
SCAP	416	1.478
SCAPI	840	3.095
IPG	379	88
LHM	379	31

Table 14: The required samples to gain confidence in the measurements made by the different measuring instruments, on both OSs for DUT 2

Correlation from Experiment Two

This section shows the correlation heatmap from subsection **5.2**.

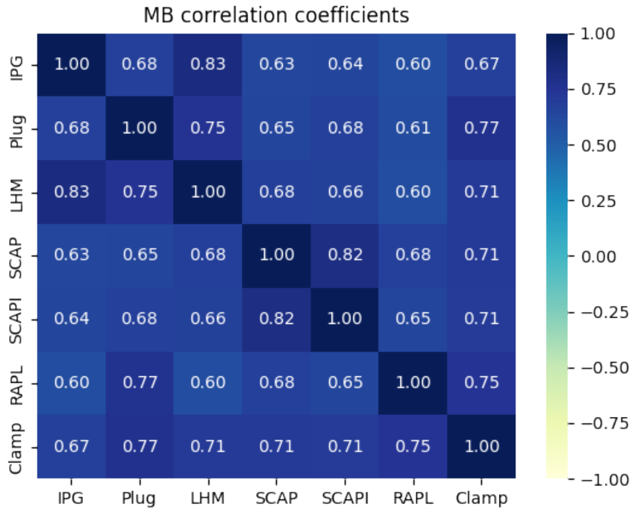
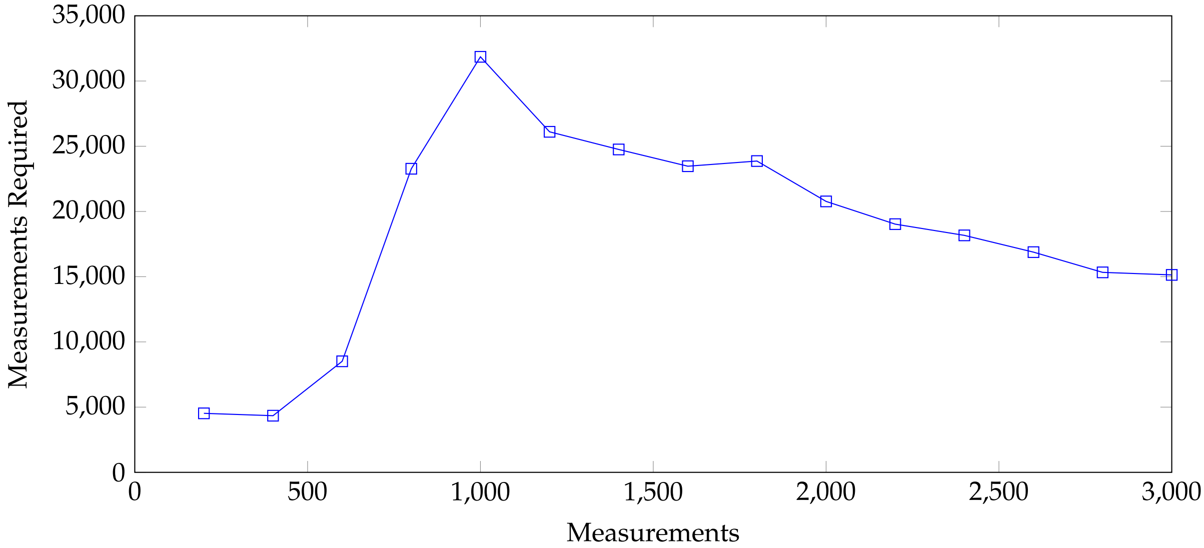


Figure 26: Heatmap showing the correlation coefficient between all of the measurement instruments on windows for the MB benchmark.

H Cochran's Formula Evolution for Experiment Two

This section shows the evolution of how many measurements are required, calculated using Cochran's formula on different amount of measurements, as used in subsection **5.2**.



I Experiment Three

This section includes the results from the third experiment, as can be found in subsection **5.3**

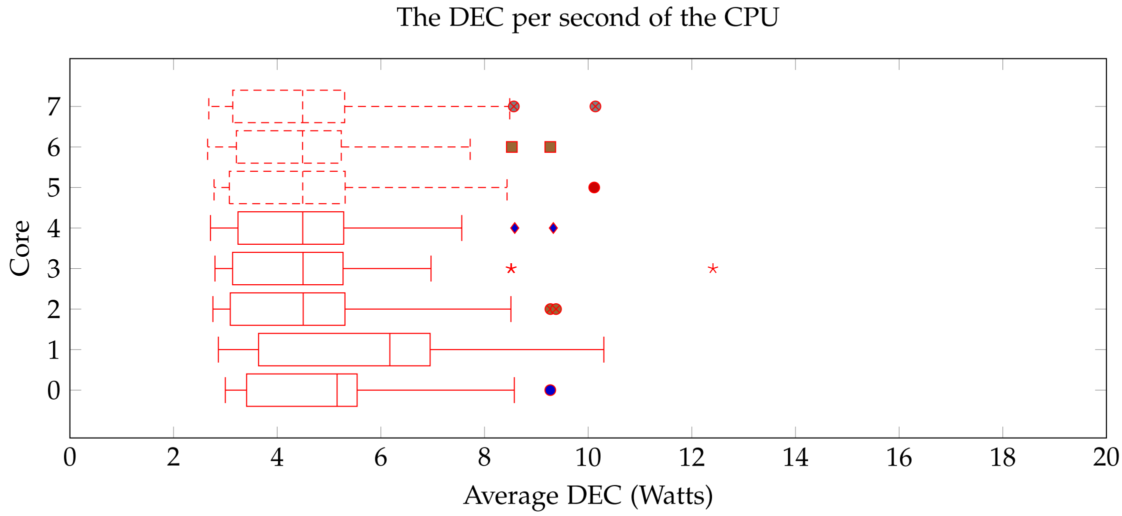


Figure 27: CPU measurements by IPG on DUT 1 for benchmark(s) NB compiled on oneAPI

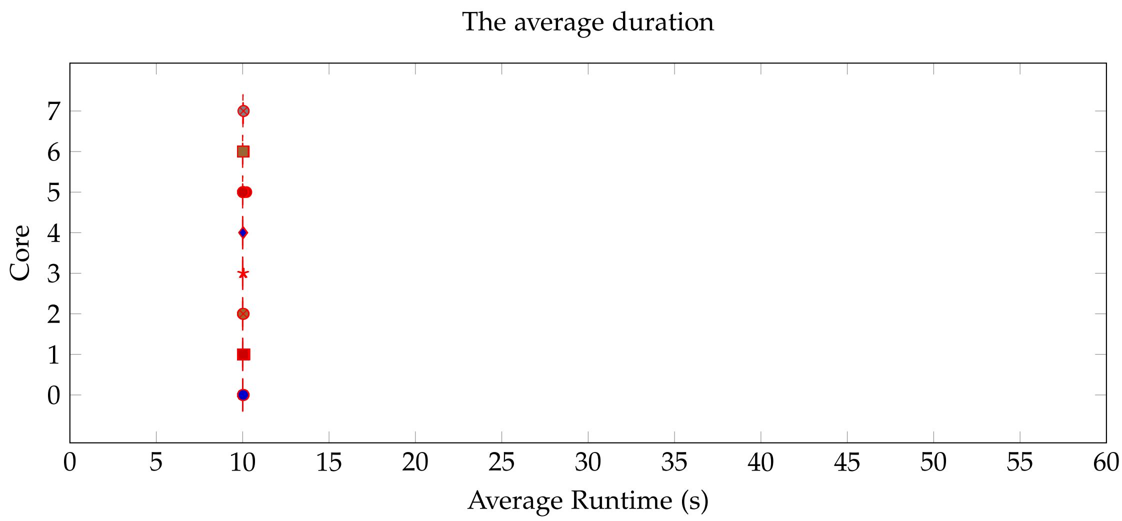


Figure 28: Runtime measurements by IPG on DUT 1 for benchmark(s) NB compiled on oneAPI

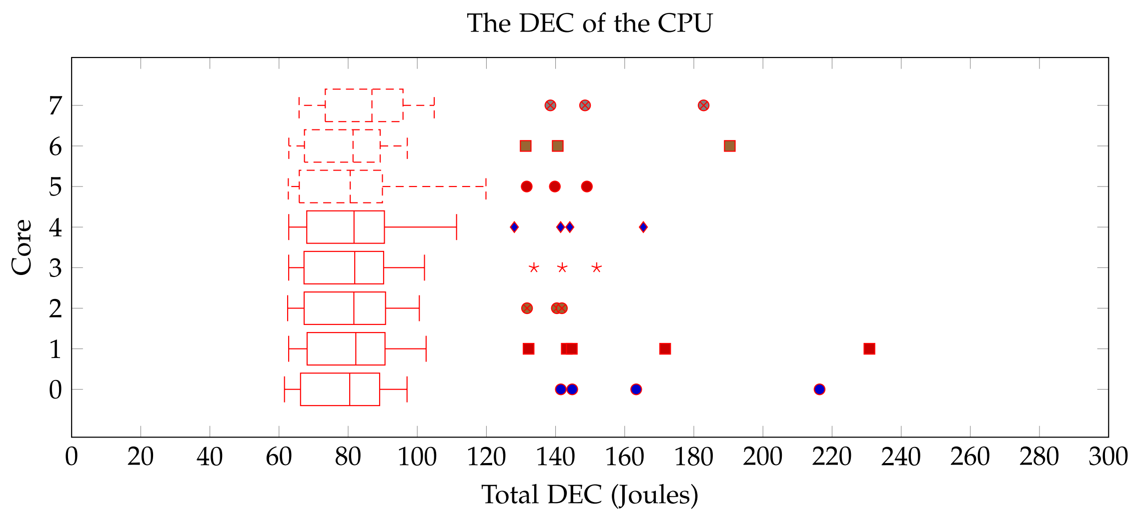


Figure 29: CPU measurements by IPG on DUT 1 for benchmark(s) SN compiled on oneAPI

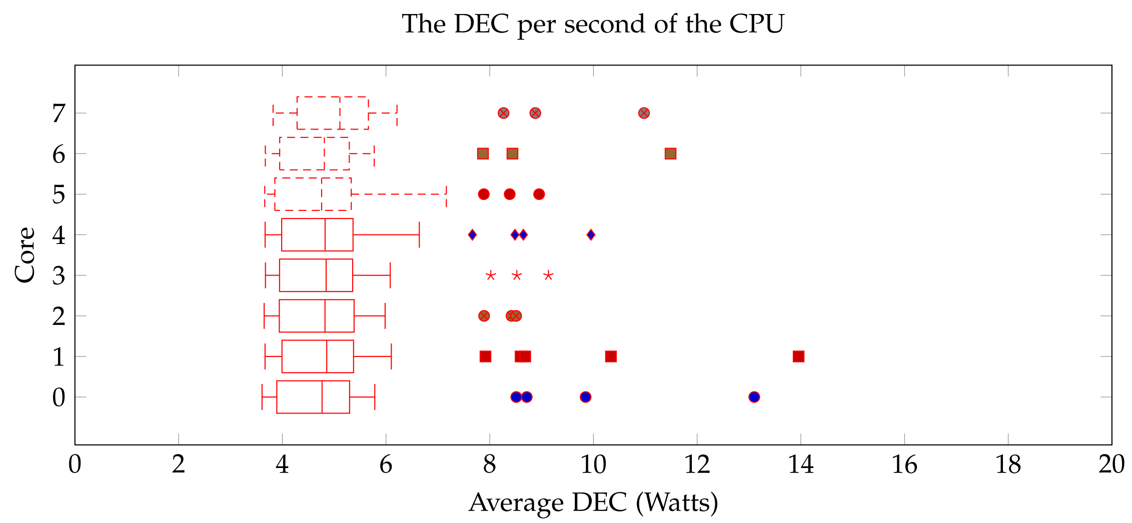


Figure 30: CPU measurements by IPG on DUT 1 for benchmark(s) SN compiled on oneAPI

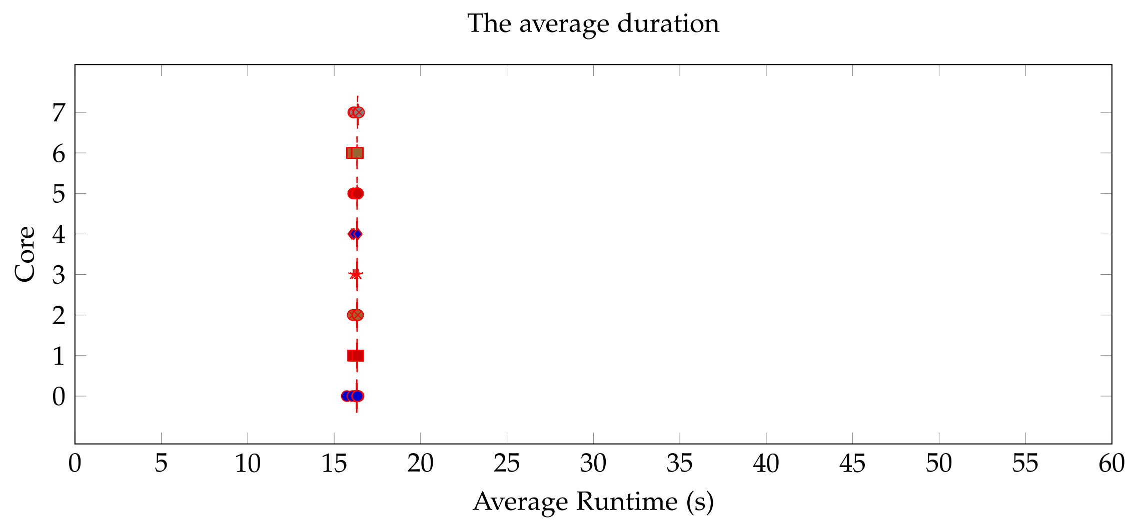


Figure 31: Runtime measurements by IPG on DUT 1 for benchmark(s) SN compiled on oneAPI

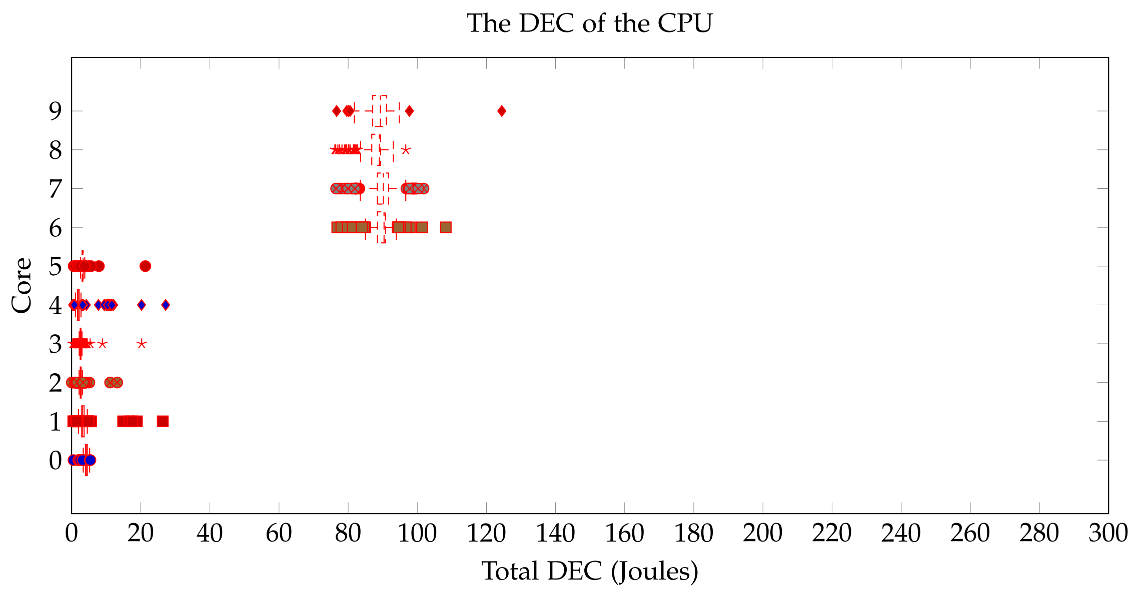


Figure 32: CPU measurements by IPG on DUT 2 for benchmark(s) NB compiled on oneAPI

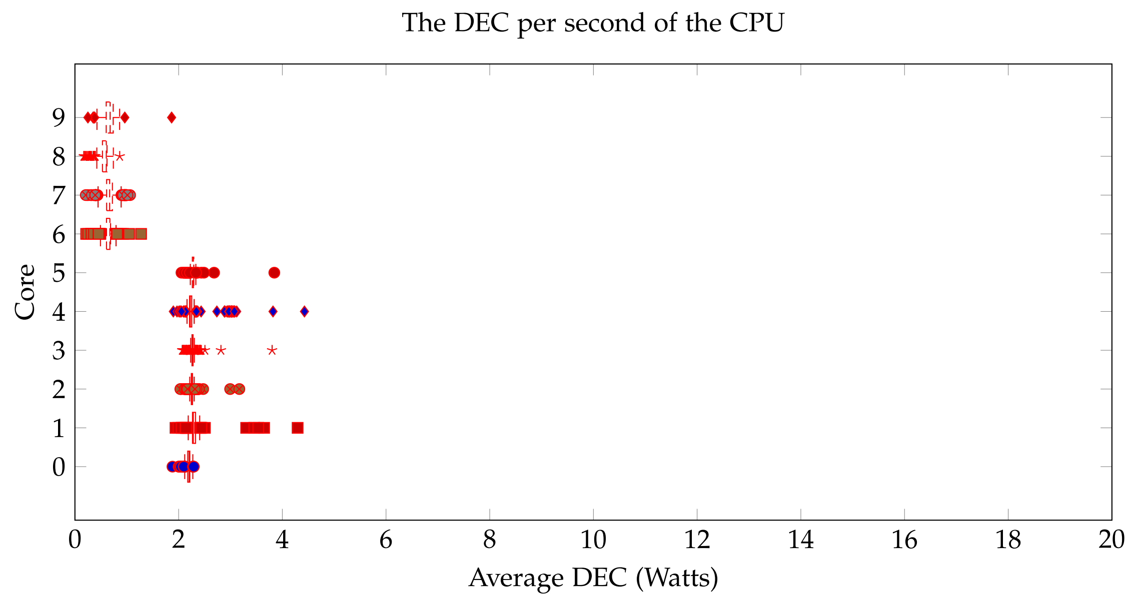


Figure 33: CPU measurements by IPG on DUT 2 for benchmark(s) NB compiled on oneAPI

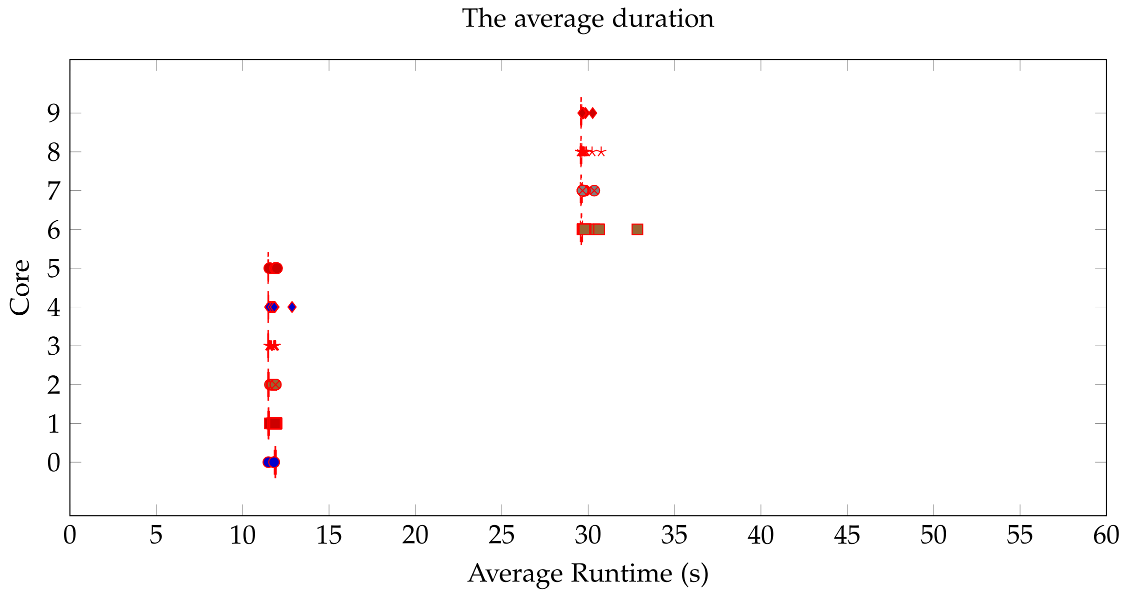


Figure 34: Runtime measurements by IPG on DUT 2 for benchmark(s) NB compiled on oneAPI

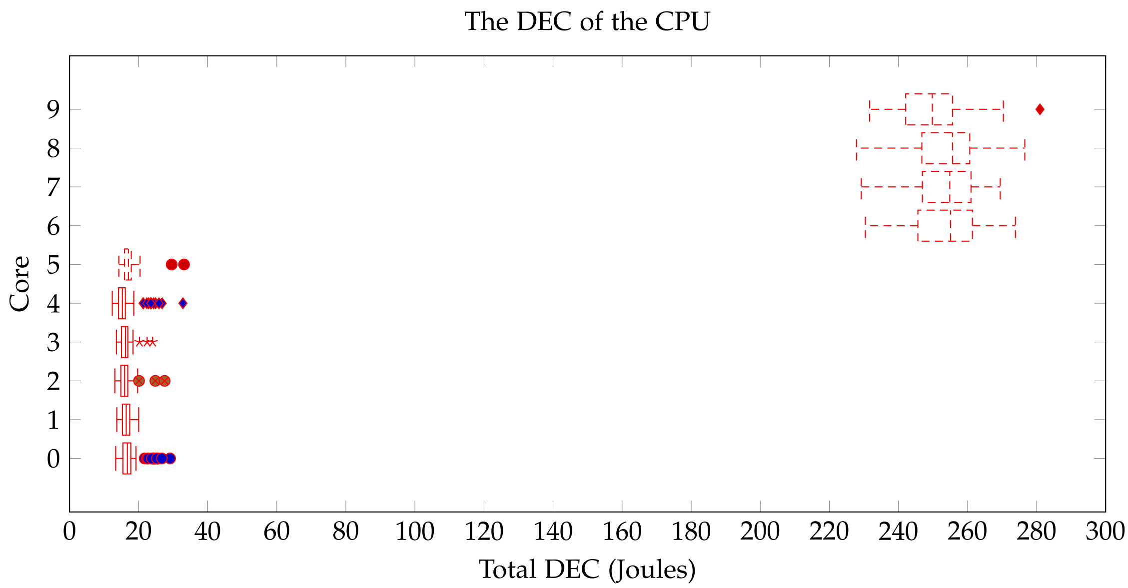


Figure 35: CPU measurements by IPG on DUT 2 for benchmark(s) SN compiled on oneAPI

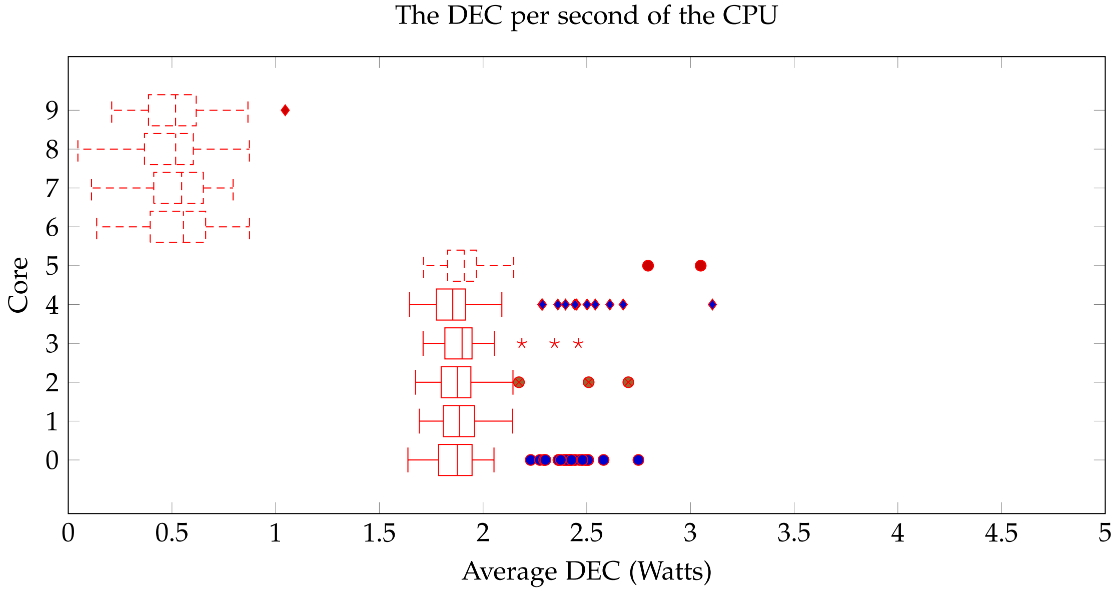


Figure 36: CPU measurements by IPG on DUT 2 for benchmark(s) SN compiled on oneAPI

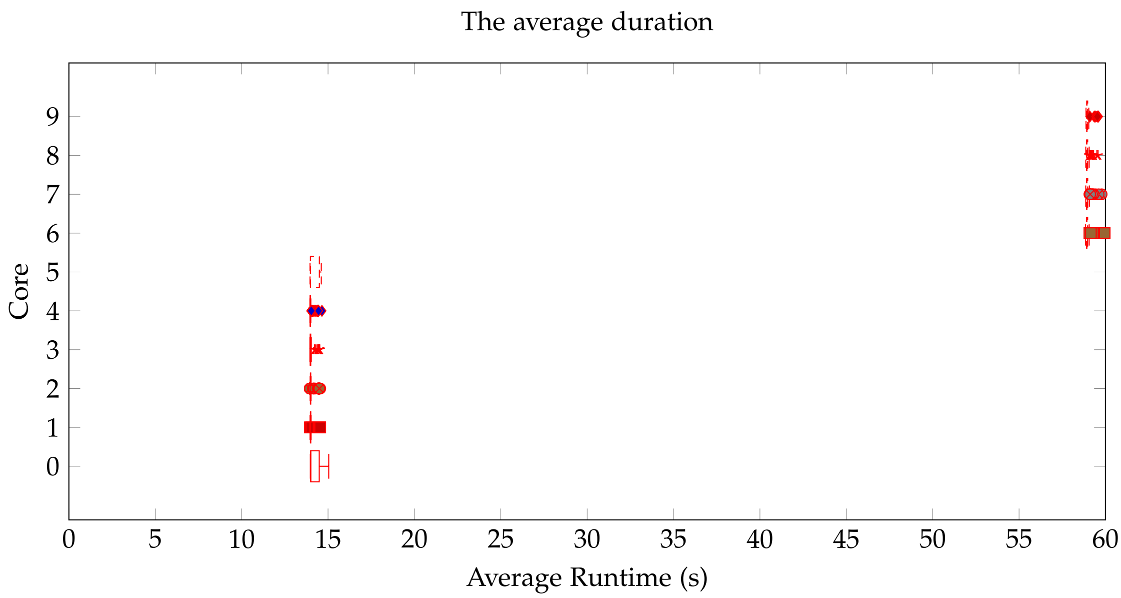


Figure 37: Runtime measurements by IPG on DUT 2 for benchmark(s) SN compiled on oneAPI

J Measurements Required in Experiment Three Cores

This section illustrates how many measurements are required from IPG when measuring the energy consumption on different cores in order to gain confidence in them, and are used in subsection **5.3**.

Initial Measurements		
Name	NB	SN
Core 0	5.162	1.991
Core 1	11.771	1.999
Core 2	5.119	2.047
Core 3	4.678	2.039
Core 4	4.597	1.979
Core 5	5.005	2.082
Core 6	4.622	1.852
Core 7	4.996	1.945

Table 15: The required samples to gain confidence in the measurements made by IPG in different cores for DUT 1

Initial Measurements		
Name	NB	SN
Core 0	4	36
Core 1	7	33
Core 2	1	35
Core 3	1	28
Core 4	3	33
Core 5	2	30
Core 6	17	115
Core 7	22	99
Core 8	18	121
Core 9	39	92

Table 16: The required samples to gain confidence in the measurements made by IPG in different cores for DUT 2

K Measurements Required in Experiment Three Macrobenchmarks

This section illustrates how many measurements are required from IPG when measuring the energy consumption of the macrobenchmarks in order to gain confidence in them, and are used in subsection **5.3**.

Initial Measurements		
Name	3DM	PCM
1 Core	1207	630
2 Cores	1470	579
3 Cores	1531	770
4 Cores	1524	913
5 Cores	2054	820
6 Cores	2359	883
7 Cores	1810	997
8 Cores	1391	811

Table 17: The required samples to gain confidence in the measurements made by IPG for the macrobenchmarks for DUT 1

Initial Measurements		
Name	3DM	PCM
1 Core	22	
2 Cores	183	84
3 Cores	135	80
4 Cores	56	71
5 Cores	79	54
6 Cores	53	78
7 Cores	25	76
8 Cores	20	78
9 Cores	42	77
10 Cores	44	100

Table 18: The required samples to gain confidence in the measurements made by IPG on for different macrobenchmarks for DUT 2

Results for Macrobenchmarks in the Third Experiment

In this section the energy consumption for an increasing number of cores can be found, referenced in subsection **5.3**.

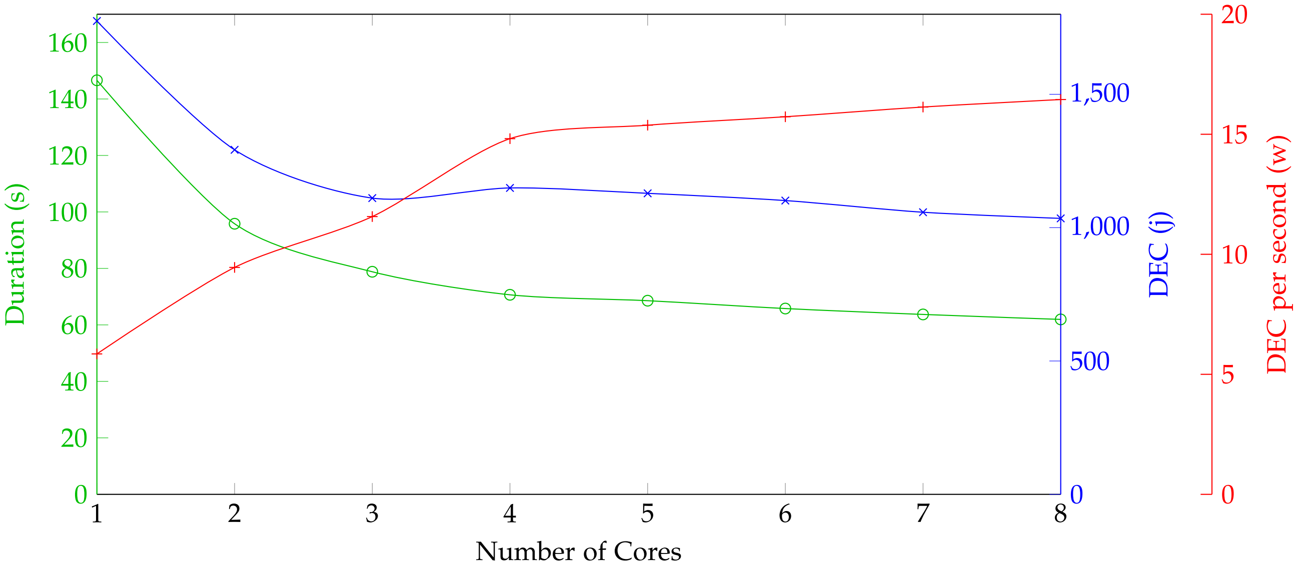


Figure 38: The evolution of the DEC (blue), DEC per second (red) and duration (green) as more cores are allocated to 3DM on DUT 1

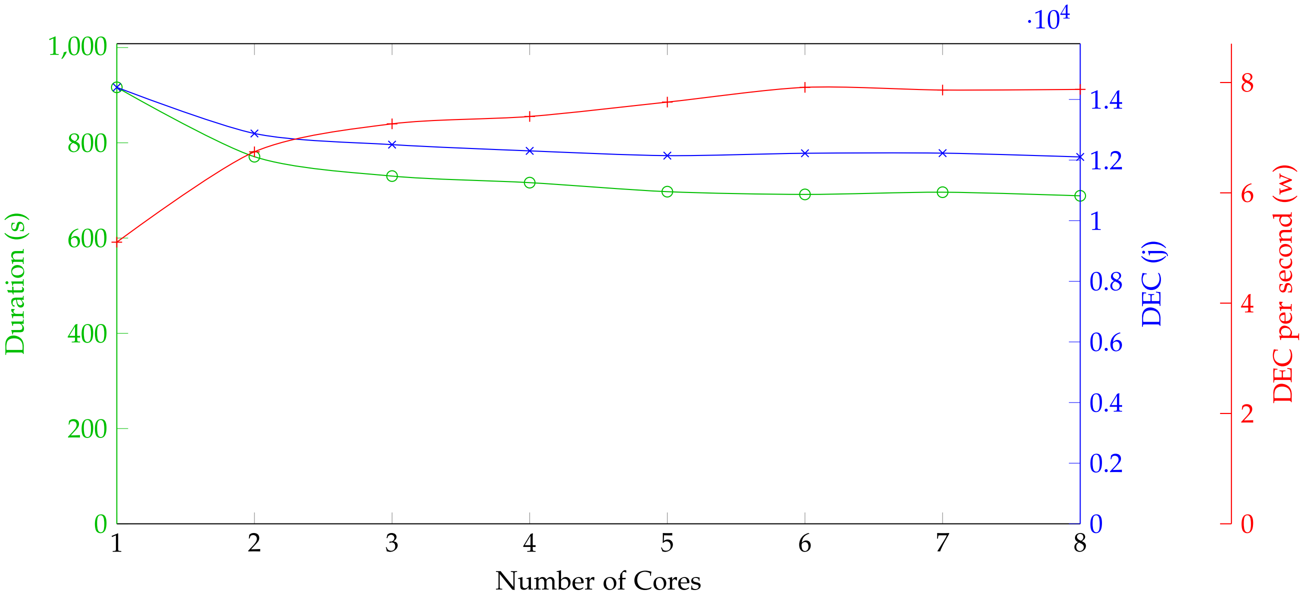


Figure 39: The evolution of the DEC (blue), DEC per second (red) and duration (green) as more cores are allocated to PCM on DUT 1

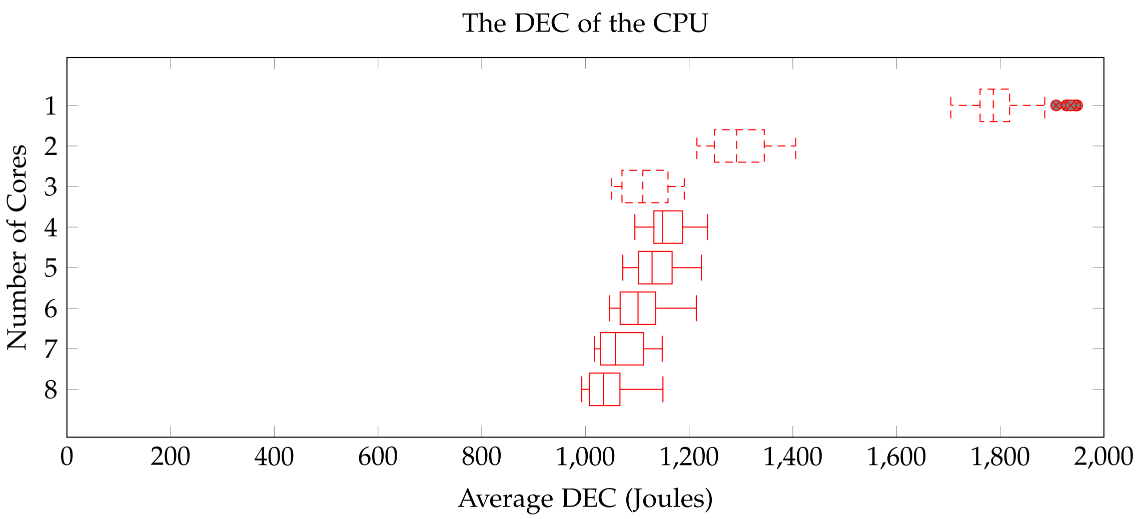


Figure 40: CPU measurements by IPG on DUT 1 for test case(s) 3DM

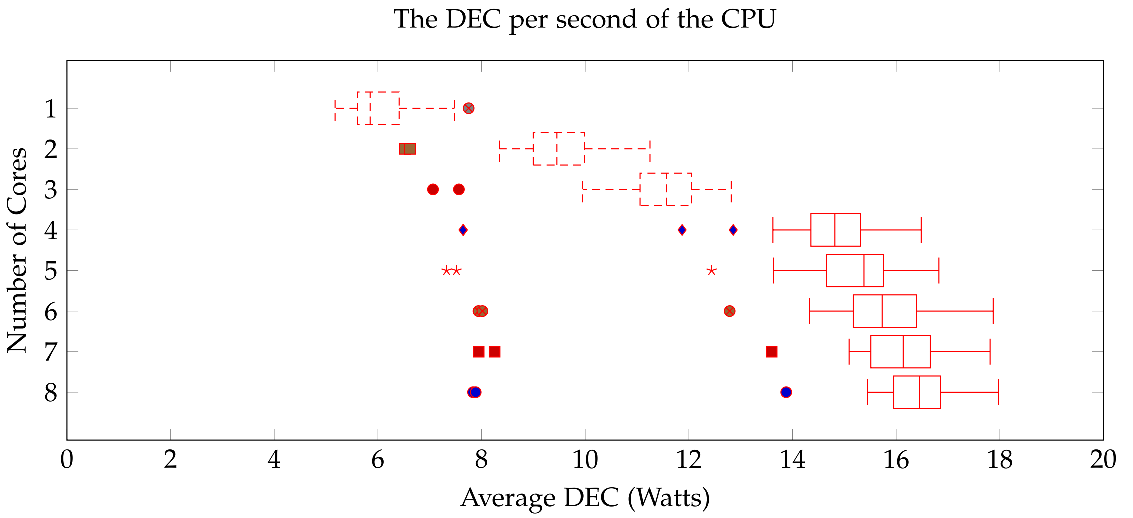


Figure 41: CPU measurements by IPG on DUT 1 for test case(s) 3DM

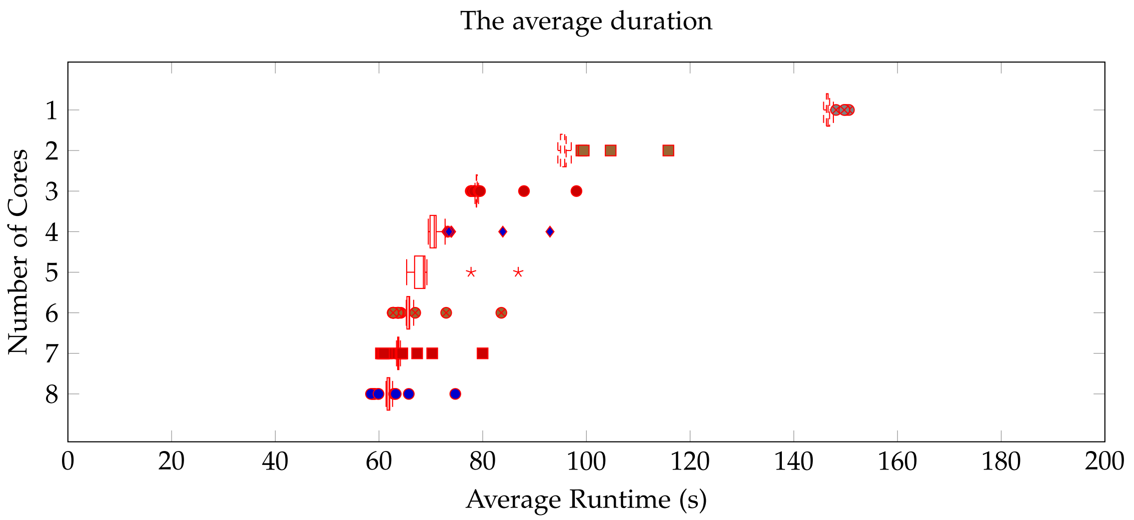


Figure 42: Runtime measurements by IPG on DUT 1 for test case(s) 3DM

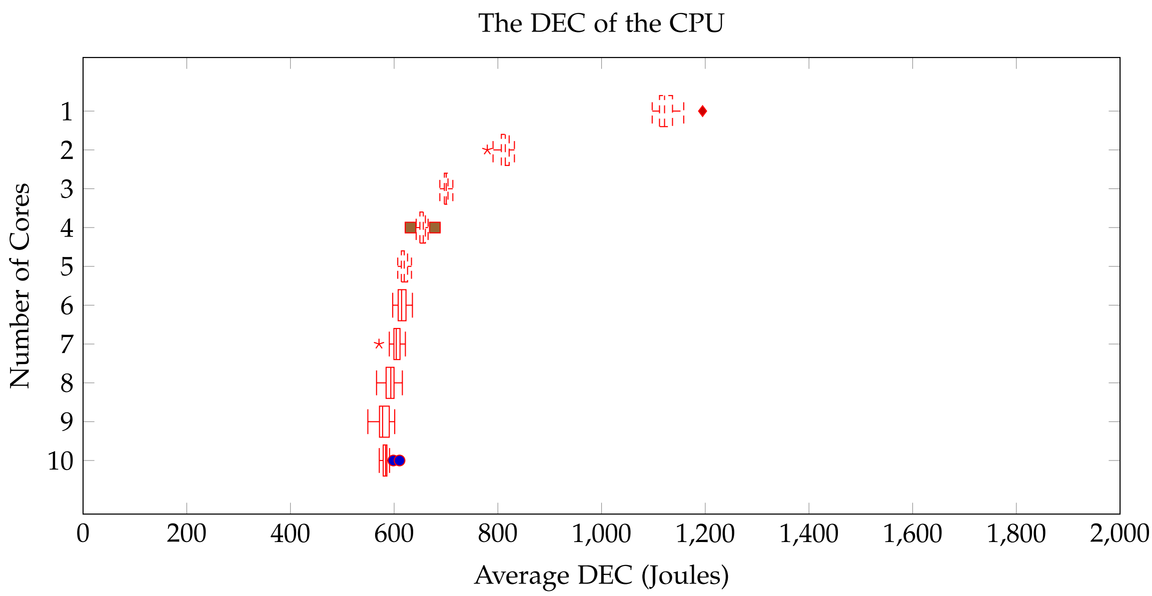


Figure 43: CPU measurements by IPG on DUT 2 for test case(s) 3DM

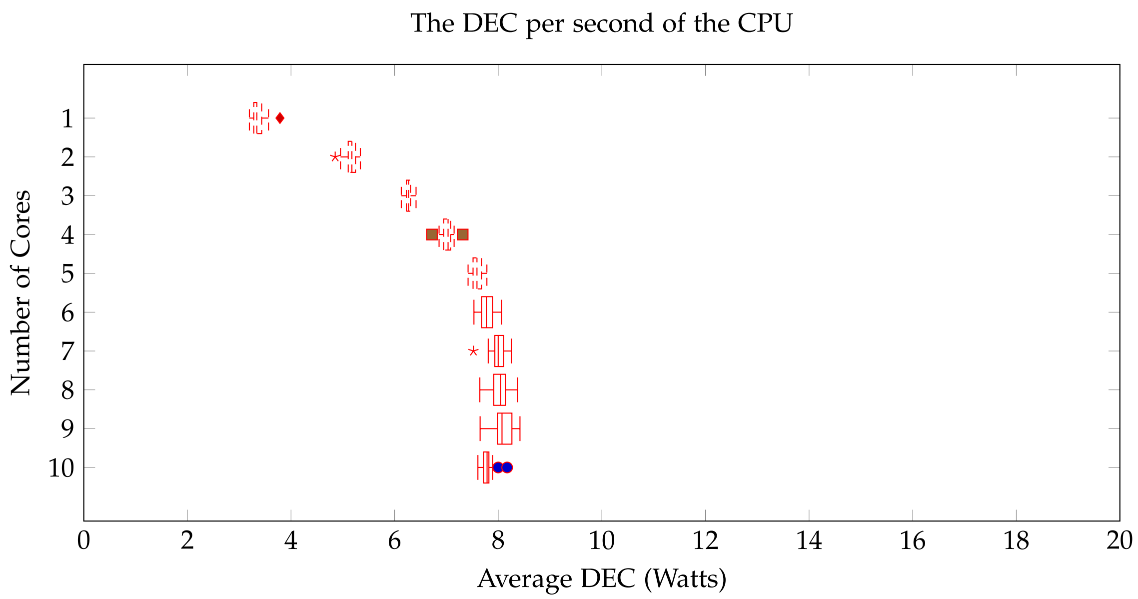


Figure 44: CPU measurements by IPG on DUT 2 for test case(s) 3DM

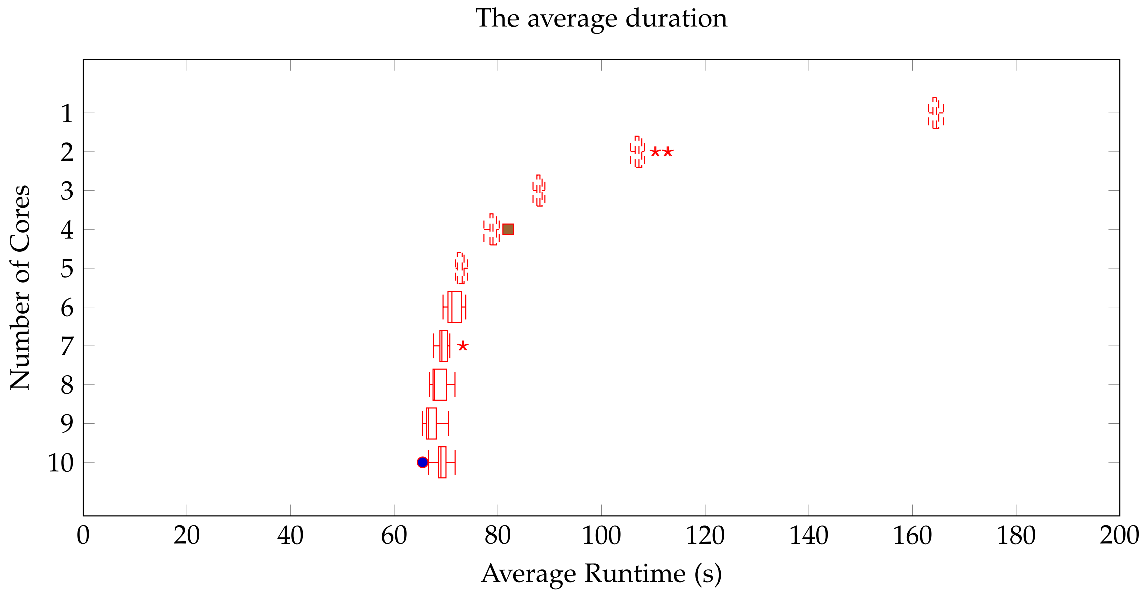


Figure 45: Runtime measurements by IPG on DUT 2 for test case(s) 3DM

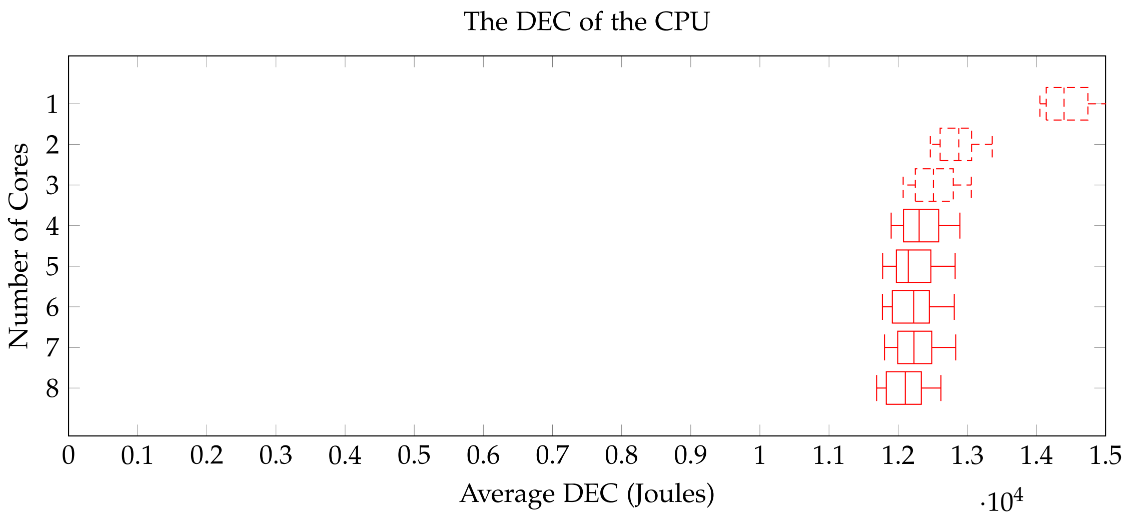


Figure 46: CPU measurements by IPG on DUT 1 for test case(s) PCM

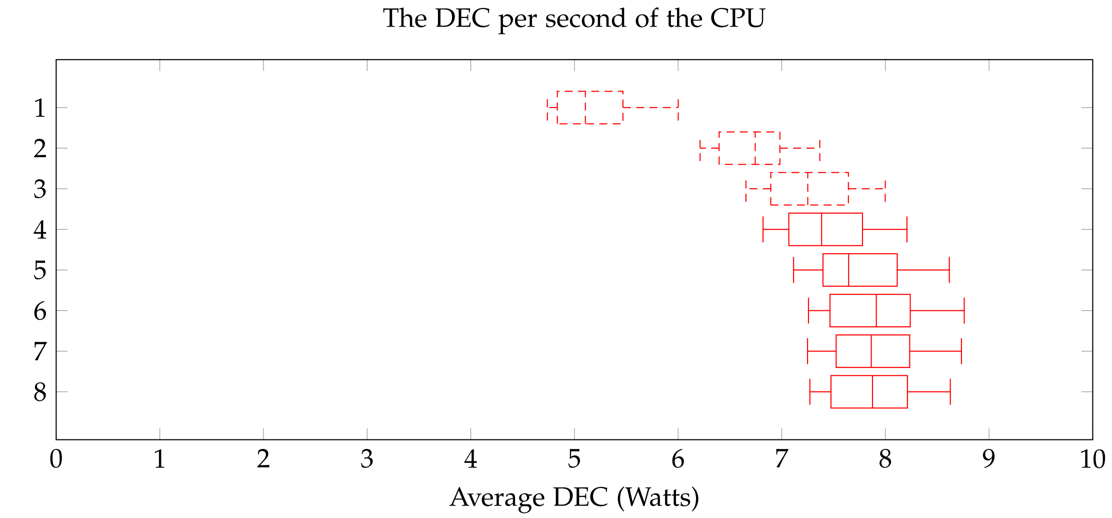


Figure 47: CPU measurements by IPG on DUT 1 for test case(s) PCM compiled on

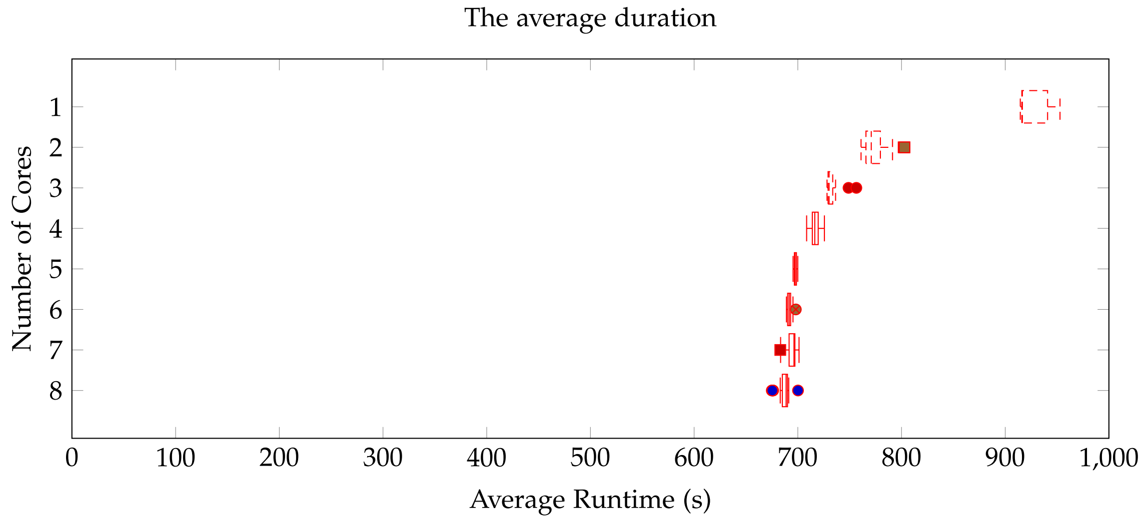


Figure 48: Runtime measurements by IPG on DUT 1 for test case(s) PCM compiled on

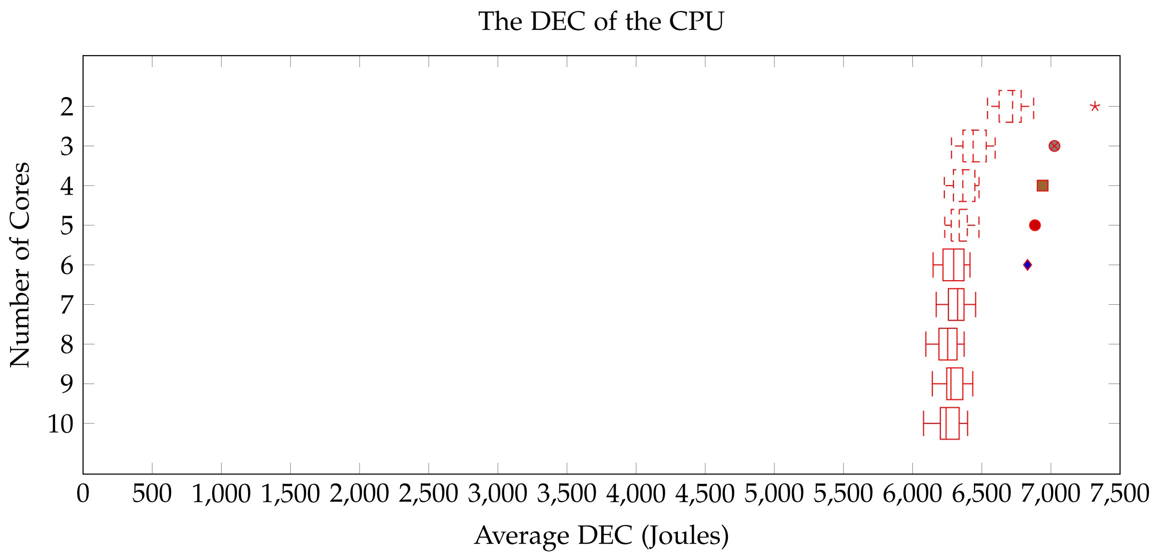


Figure 49: CPU measurements by IPG on DUT 2 for test case(s) PCM

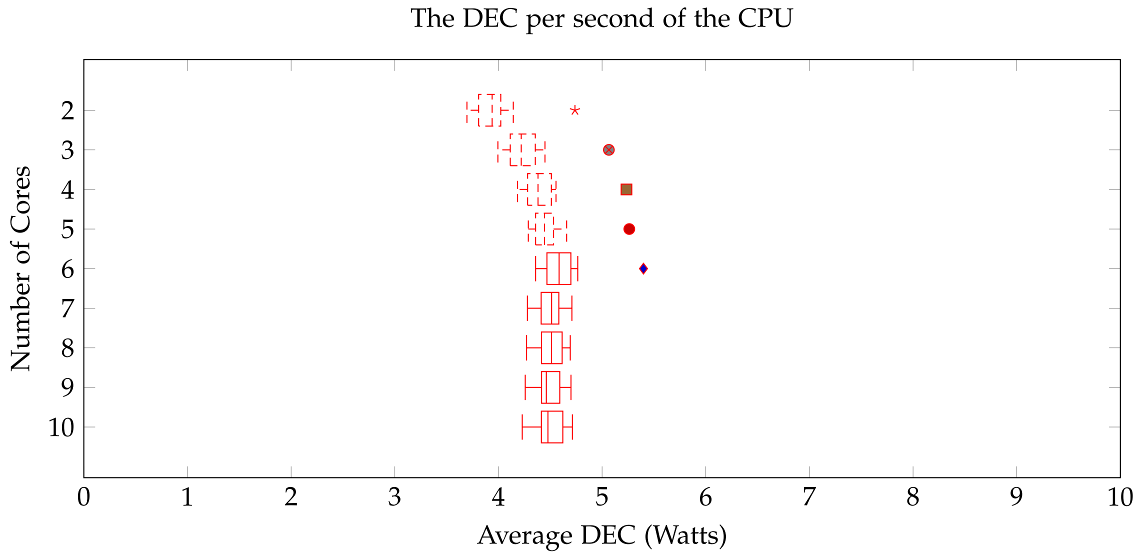


Figure 50: CPU measurements by IPG on DUT 2 for test case(s) PCM

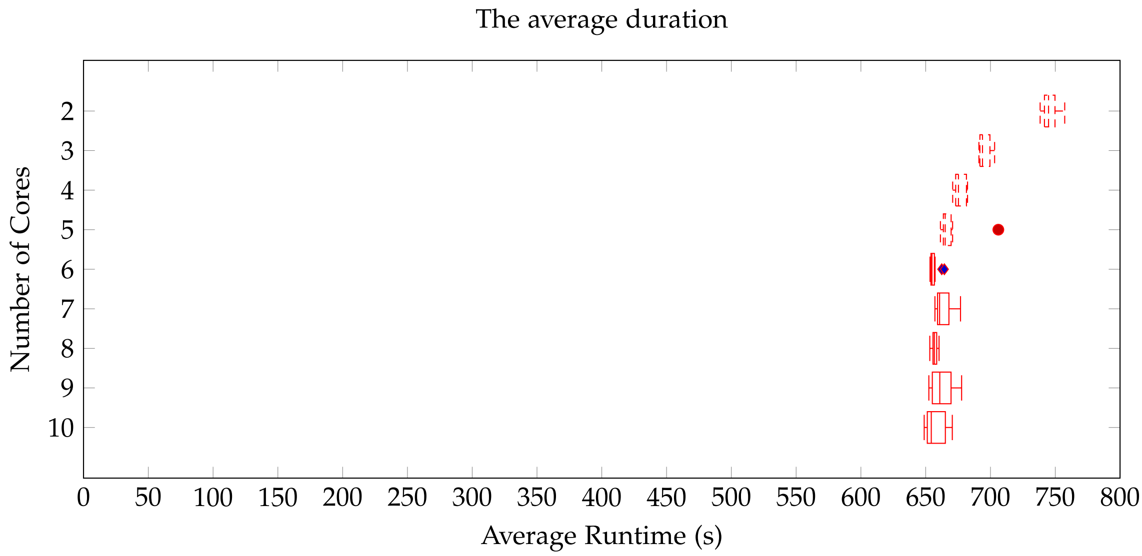


Figure 51: Runtime measurements by IPG on DUT 2 for test case(s) PCM

L Experiment Three Combined Results

This section shows the results illustrated in Appendix **K** from experiment three as a table. This table is used in subsection **5.3**.

Performance Evolution								
	DUT 1				DUT 2			
	PCM		3DM		PCM		3DM	
Number of Cores	Duration	DEC	Duration	DEC	Duration	DEC	Duration	DEC
1	916.5 s	14399.3 j	146.6 s	1774.6 j			164.0 s	1111.2 j
2	−15.9%	−10.6%	−34.6%	−27.2%	744.8 s	6722.8 j	−34.5%	−26.7%
3	−5.2%	−2.9%	−17.7%	−14.0%	−6.8%	−4.2%	−17.7%	−14.0%
4	−1.9%	−1.6%	−10.4%	+3.4%	−2.7%	−1.2%	−10.3%	−6.3%
5	−2.6%	−1.3%	−2.9%	−1.8%	−1.5%	−0.4%	−7.4%	−5.6%
6	−0.8%	−0.6%	−4.9%	−2.4%	−1.5%	−0.6%	−3.0%	−0.9%
7	+0.7%	+0.1%	−3.2%	−4.0%	+0.8%	+0.4%	−2.6%	−1.5%
8	−1.1%	−1.0%	−2.8%	−2.2%	−0.5%	−1.1%	−3.1%	−1.8%
9					+0.6%	+0.4%	−0.4%	−2.7%
10					−1.0%	−0.5%	+3.5	+0.7%

Table 19: The results when executing PCM and 3DM on DUT 1 and 2, where each row represents the percent difference from the previous row.

Measurements Requied in Experiment Three P vs. E Cores

This section illustrates how many measurements are required from IP when measuring the energy consumption of PCM on four P cores, four E cores, or two of each. These results are referenced in subsection **5.3**

Initial Measurements	
Name	PCM
4P	131
4E	125
2P2E	44

Table 20: The required samples to gain confidence in the measurements made by IPG when comparing P and E cores for DUT 2

M Results for P vs. E Cores in Third Experiment

This section shows the results obtained when comparing the performance of four P cores, four E cores or two of each when running PCM on DUT 2. The results are referenced in subsection **5.3**

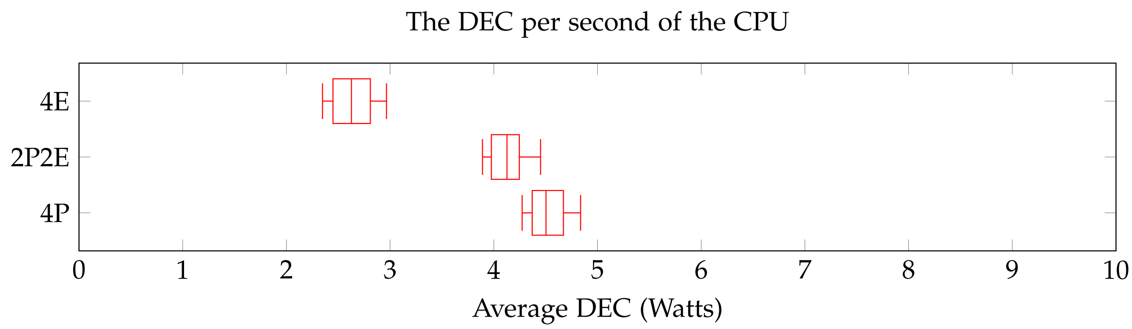


Figure 52: CPU measurements by IPG on DUT 2 for test case(s) PCM compiled on

N Energy Consumption Over Time

This section shows how both macrobenchmarks energy consumption evolves over time for DUT 2, used in subsection **5.3**. In this section 3DM and PCM are plottet with two cores and all cores to illustrate the difference the additional resources make.

3DM can be seen in Figure **53** and Figure **54**. In these graphs, the different phases of 3DM can be seen. For both two and ten cores there is a startup period until around 14 seconds, after which the benchmark starts. On ten cores, load is on 25 watts for approximately 18 seconds, while for two cores the energy consumption is on 12 watts for 60 seconds.

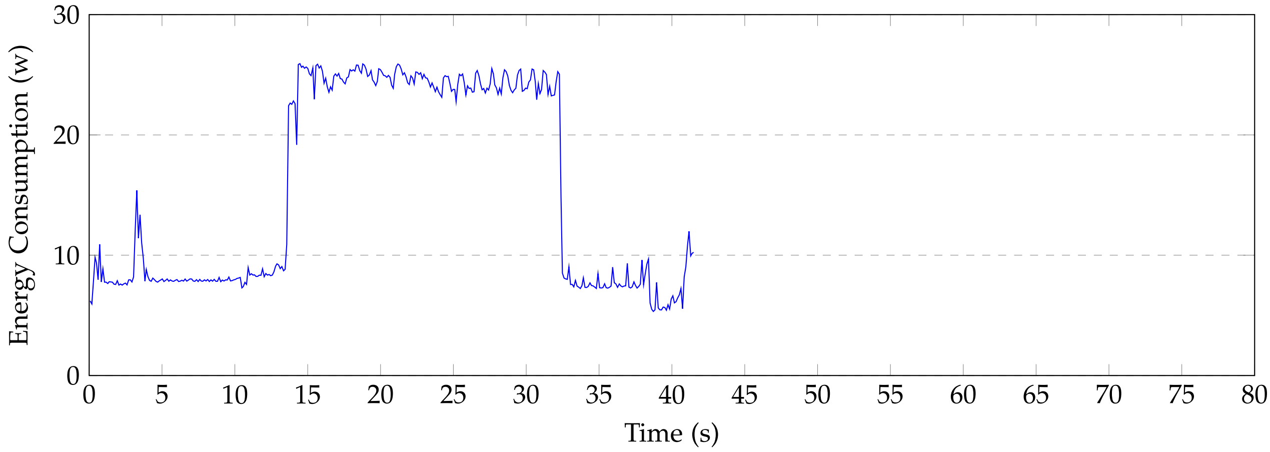


Figure 53: A timeseries of the energy consumption over time for DUT 2 when running 3DM for all cores

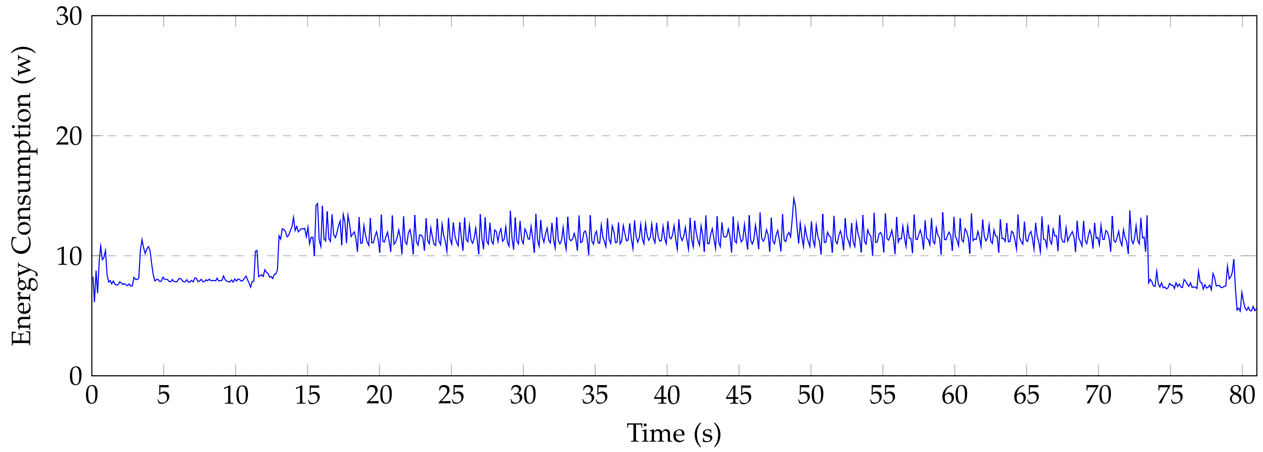


Figure 54: A timeseries of the energy consumption over time for DUT 2 when running 3DM on two core

For PCM, the graphs can be seen in Figure **55** and Figure **56**. For PCM, a smaller difference can be found between two and ten cores compared to 3DM, and there are a few reasons for this. The load is first of all lower for this benchmark, which means the additional resources gives a diminishing return. When looking at Figure **56** it can however also be observed that the upper limit exceeds what was found for two cores for 3DM, which was 12 watts. 12 watts is exceeded at three points during runtime, this being $230 - 260$, $390 - 400$, $580 - 600$, which amounts to 8% of the total runtime. This indicates that we did not find all background processes related to PCM when setting affinity, resulting in too many resouces being allocated to some processes. An effort was put into finding these processes, but without success. This means that the table in Table **19** represents a lower limit for PCM, as all cores are used for some processes, resulting in a lower execution time and DEC, the results are however still deemed valid.

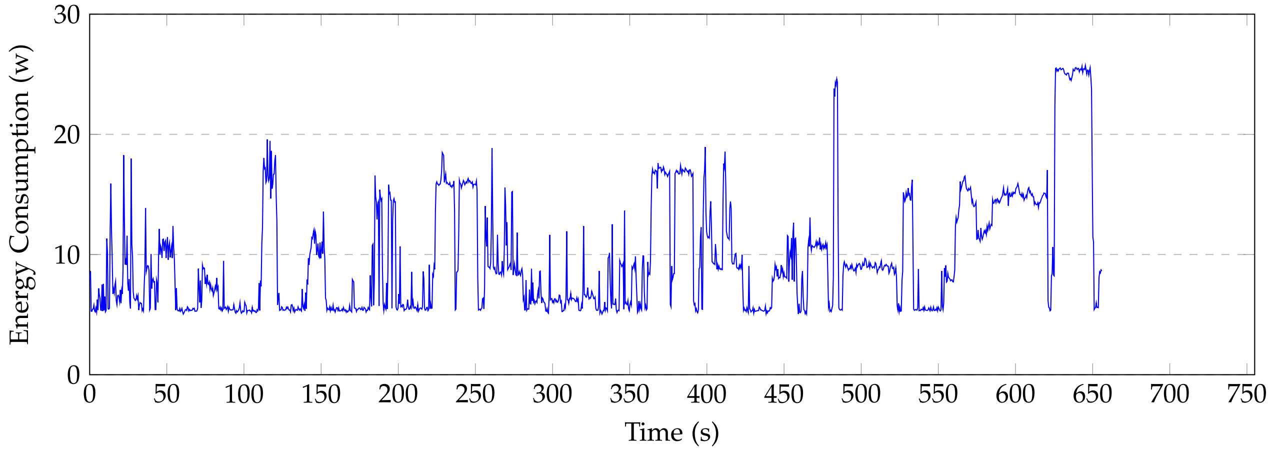


Figure 55: A timeseries of the energy consumption over time for DUT 2 when running PCM for all cores

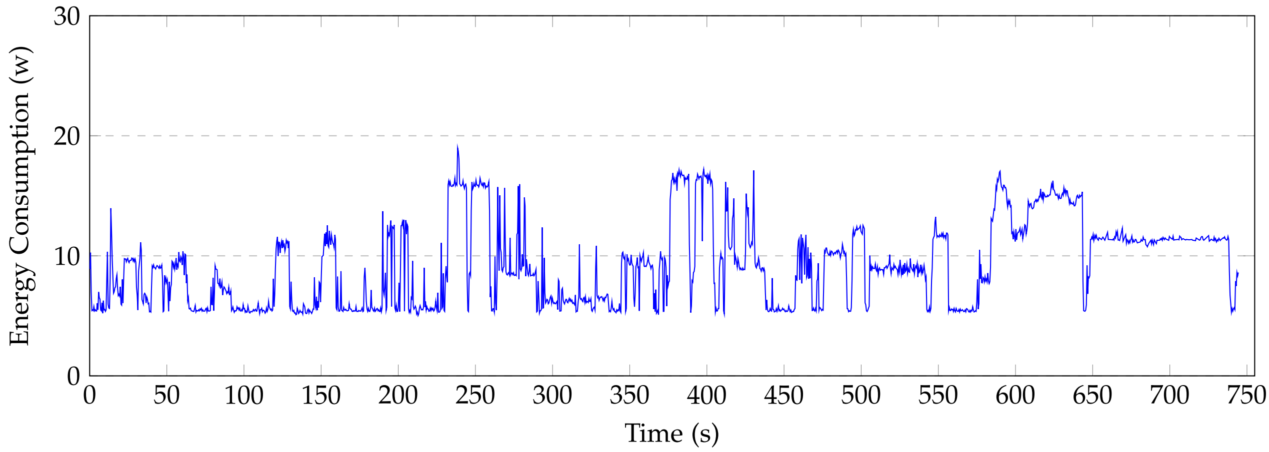


Figure 56: A timeseries of the energy consumption over time for DUT 2 when running on two cores

N.1 Energy usage trends analysis

To explore our hypothesis regarding electrical network noise interfering with phase synchronization, we categorized the data into `working hours` (7:00 to 17:00) and `non-working hours` (17:00 to 7:00). We found no significant variation in power consumption peaks between the two categories. However, periods of low energy usage were higher during `working hours`, suggesting that they did not affect benchmark measurements but did impact idle case measurements. This observation aligns with the results presented in Figure **3**, which represent the DEC values. To better understand this, consider that the 3,000 DEC measurements, each consist of the total energy consumption during benchmark execution and a corresponding idle case measurement as explained in subsection **4.2**.

Power supplies are typically less efficient at lower loads[56], causing reactive energy to contribute more to overall usage. As we observed these effects only during low energy usage periods, we focused on valleys in the time series data by identifying local minimums in each 1-minute window. Analyzing data trends using linear regression, we found that `working hours` exhibited a slight increase with a slope of 0.633, while `non-working hours` showed a slight decrease with a slope of -1.288.