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

April 11, 2023

Abstract

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

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 C++ test cases using different measuring instruments 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 P-core setup and an Intel Raptor Lake CPU with P- and E-cores. We analyze the impact of Asymmetric Multicore Processors (AMPs) on parallel execution compared to traditional Symmetric MultiCore Processors.

- **RQ1**: How does the compiler used to compile the test cases impact the energy consumption?
- **RQ2**:What are the advantages and drawbacks of the different types of measuring instruments in terms of accuracy, ease of use, and cost?
- **RQ3**:What effect does parallelism have on the energy consumption of the test cases?
- **RQ4**:What effect do P- and E-cores have on the parallel execution of a process, compared to a traditional desktop CPU?

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 **6** the results are presented whereafter they are discussed in Section **7** and finally a conclusion is made in Section **8**.

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] where 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] and as such will not be repeated, however, it will be expanded upon.

2.2 Parallel Software

Amdahl's law describes the maximum potential speedup that can be achieved through the parallelization of an algorithm 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. Then 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. However, the comparison was based on an analytical model and not real measurements.[5]

[6] presented a program that solves Laplace equations and compares the observed speedup for computing the 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 speedup and the estimates 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 regarding energy consumption. It was found that as the number of threads increased, energy consumption would do the same. However, this was only up to a certain point, after which energy consumption would start to decrease as the number of threads approached the number of cores in the CPU. The peak of this energy consumption was 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 using one thread. Furthermore, the study investigated the energy-performance trade-off using the Energy-Delay-Product (EDP), which is the product of energy consumption and execution time. Using EDP, the study generally found parallel execution to be more favorable; however, depending on the benchmark, increasing the number of threads might not result in an improvement in EDP.[7]

In [8], the energy consumption for sequential and parallel genetic algorithms was explored, where one research question aimed to explore the impact on energy consumption when using different numbers of cores. It 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. Parallelism's ability to reduce energy consumption was argued to be due to the large number of cores working to solve the problem simultaneously, where the combination of more cores, and more parallel operations per time unit will require less energy. 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], the behavior of parallel applications and the relationship between execution time and energy consumption are explored. It tests four different language constructs which can be used to implement parallelism in C#. Furthermore, it uses varying amounts of threads and a sample of micro- and macro-benchmarks. It found that workload size has a large influence on running 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 macro-benchmarks the findings remain consistent, although the impact becomes low for the macrobenchmarks due to an overall larger energy consumption. Furthermore, it has included some recommendations, which should be considered:[9]

- Shield cores: Avoid unintended threads running on the cores used in the benchmarking
- PowerUp: Can be used to ensure that benchmark is not optimized away during compilation
- Static clock: Make the clock rate of the CPU as static as possible
- Interrupt request: Avoid interrupt requests being sent to cores used in the benchmarking

- Turn off CPU turbo boost
- Turn off hyperthreading

2.3 Compilers

In [10], the C++ language and different compilers were explored and compared to determine the impact of using various coding styles and compilers, with the goal of finding a balance between performance and energy efficiency. The different coding styles introduced examined the impact of splitting CPU and IO operations and interrupting CPU-intensive instructions with sleep statements. The C++ compilers used in [10] included MinGW GCC, Cygwin GCC, Borland C++, and Visual C++, and the energy measurements were performed using Windows Performance Analyzer (WPA). All compilers were used with default settings, and no optimizations were chosen. This decision was based on works like [11], where it was found that mainstream compilers apply multiple optimizations to the final code, which in the worst case may result in worse performance and increased energy consumption. The issue of optimizations being highly machine-dependent was also demonstrated in [12], where analysis and optimizations were conducted on a Texas Instruments C6200 DSP CPU. In [12], it was discovered that a significant portion of the energy was used by fetching instructions, which was addressed by introducing a fetch packet mechanism. The study also found loop-unrolling to reduce energy consumption. While these optimizations decreased energy consumption for the Texas Instruments C6200 DSP CPU, the authors noted that varying results were expected for other CPUs. A similar conclusion was also reached in [10], where it was found that when choosing a compiler and coding style, energy reduction depends on the nature of the target machine and application. Based on the test case used, which involved an election sort algorithm, the best performance 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 both separating IO and CPU operations and interrupting CPU-intensive instructions with sleep statements also decreased energy consumption.

2.4 Asymmetric Multicore Processors

AMPs are CPUs in which not all cores are treated equally. One example of this is the combination of performance cores and efficiency cores, as seen in Intel's Alder Lake and Raptor Lake. Intel's Thread Director (ITD) was introduced alongside Intel's Alder Lake. The purpose of ITD is to assist the operating system in deciding on which cores to run a thread. In [13], support for utilizing ITD in Linux was developed, although official support for ITD has since been released. Additionally, some SPEC benchmarks were 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 is for threads that perform similarly on P- and E-cores, while Class 1 is for threads where P-cores are preferred.[14] Furthermore, class 3, which is for threads that are 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 it still produced some errors. However, the correlation coefficient was higher at (> 0.8). The study then 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.[13]

3 Background

3.1 CPU States

This section provides an overview of CPU-states. The concept of CPU-states is concerned with how a system manages its energy consumption during different operational conditions. The C-states are a crucial aspect of CPU-states, as they dictate the extent to which a system shuts down various components of the CPU to conserve energy. The C0 state represents the normal operation of a computer under load.[15, 16] As the system moves from C0 to C10 [3], progressively more components of the CPU are shut down until, in C10, the CPU is almost inactive. It is important to note that the number of C-states supported may vary depending on the CPU and motherboard in use, in [3] the workstation used supported from C0 to C10 states.

In our work the C-states can have a large impact on the energy consumption of the test cases, especially the idle case as was found in [3].

3.2 Performance and Efficiency cores

For the CPU architecture x86, the core layout has comprised of identical cores. However, the ARM architecture introduced the big.LITTLE layout in 2011[17]. It is an architecture that utilizes two types of cores, a set for maximum energy efficiency and a set for maximum computer performance.[18]. Intel introduced a hybrid architecture in 2021[19] codenamed Alder lake, which is similar to ARM's big.LITTLE architecture. Alder lake also has two types of cores: performance cores (P-cores) and efficiency cores (E-cores). These types of cores are optimized for different tasks, where P-cores are standard CPU cores focusing on maximizing performance. In contrast, the E-cores are designed to maximize performance per watt and are intended to handle smaller non-time critical jobs, such as background services[20].

3.3 CPU Affinity

Affinity is a feature in operating systems(OSs) that enables processes to be bound to specific cores in a multi-core processor. In OSs, jobs and threads are constantly rescheduled for optimal system performance, which means that the same process can be assigned to different cores of the CPU. Processor affinity allows applications to bind or unbind a process to a specific set of cores or range of cores/CPUs. When a process is pinned to a core, the OS ensures it only executes on the assigned core(s) or CPU(s) each time it is scheduled.[21]

```
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#

Processor affinity is particularly useful for scaling performance on multi-core processor architectures that share the same global memory and have local caches referred to as the Uniform memory access architecture. Processor affinity is also useful for out study, as this allows the framework to assign a single or a set of cores and threads to a process.[21]

When setting the affinity for a process in C#, it is done through a bitmask, where each bi represents a CPU core. An example of how it is done in C# can be seen 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. For the priority the value 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.[22]

For the priority level, the levels 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.[22]

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 [22]. When assigning a base priority where both the priority class and thread priority are the default values, e.i.`NORMAL`, the base priority is 8.[22]

The idea of having different priorities is to treat threads with the same priority equally, by assigning time slices to each thread in a round-robin fashion, starting with the highest priority. In the case of none of the highest priority threads being ready to run, the lower priority threads will be assigned time slices. The lower-priority threads will then execute until a higher-priority thread is available, in which case the system will assign a full time slice to the thread, and stop executing the lower-priority threads, without time to finish using its time slice.[22]

```
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#

Note when setting priority class and priority level for a process through C#, the priority class is supported for both Windows and Linux, while the priority level is only supported for Windows. An example of how both the priority class and priority level can be set for a process and its threads can be seen in Listing **2**.

3.5 OpenMP

OpenMP (Open Multi-Processing) is a parallel programming API consisting of a set of compiler directives and runtime library routines, with support for multiple platforms like Linux, macOS, and Windows as well as multiple compilers like GCC, LLVM/Clan, and Intel's OpenApi. OpenMP allows programmers to write parallel code for multi-core CPUs and GPUs.[23]

The directives provide a way to specify parallelism among multiple threads of execution within a single program, while the library provides mechanisms for managing threads and data synchronization. When using OpenMP programmers can write parallel codes and take advantage of multiple processors without having to deal with low-level details.[23]

When executing using OpenMP, the parallel mode used is called the Fork-Join Execution Model. This model works by first 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. This process is called a fork. 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.[23]

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

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

When using OpenMP, the parallel regions are identified using a series of directives and clauses, where the basic format 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.[23]

4 Experimental Setup

4.1 Measuring Instruments

This section present the different measuring instruments utilized in our work. The measuring instruments utilized in our previous work will only be briefly introduced, however more detail can be found in [3]. In this paper, four software-based measuring instruments and two hardware-based measuring instrument were used, one of the latter was used as our ground truth.

Intel's Running Average Power Limit (RAPL): is in the literature a commonly used software-based measuring instrument.[3] It uses model-specific-registers (MSRs) and Hardware performance counters to calculate how much energy the processor uses. The MSRs RAPL uses include *MSR_PKG_ENERGY_STATUS*, *MSR_DRAM_ENERGY_STATUS*, *MSR_PP0_ENERGY_STATUS* and *MSR_PP1_ENERGY_STATUS*. Which corresponds to the power domains, PKG, DRAM, PP0, and PP1 which are explained in [3]. RAPL has previously only been directly accessible on Linux and Mac. In [3] we found that RAPL had a high correlation of 0.81 with our ground truth on Linux.[3]

Intel Power Gadget (IPG): is a software tool created by Intel, which can estimate the power of Intel processors. It contains a command line version called Powerlog which allows accessing the energy consumption using callable APIs. It uses the same hardware counters and MSRs as RAPL[24], therefore it is expected to observe similar measurements to that of RAPL. Which is also shown in [3] where we found that IPG had a high correlation of 0.78 with our ground truth on Windows. We also found that IPG had a high correlation of 0.83 with RAPL, although the measurements is on different operating systems.[3]

Libre Hardware Monitor (LHM): is a fork of Open Hardware Monitor, where the difference is that LHM does not have a UI.[25] Both projects are open source. LHM can use the same hardware counters and MSRs as RAPL and IPG and as such can measure the power domains PKG, DRAM, PP0, and PP1. Since it uses the methods to read energy consumption, a similar measurement is expected between LHM and IPG. We found that LHM correlated 0.76 with our ground truth on Windows. LHM was also found to have a high correlation of 0.85 with IPG.[3]

MN60 AC Current Clamp (Clamp): Serves as our ground truth measurement. It is a setup comprised of an MN60 AC clamp that is connected to the phase of the wire that goes into the PSU. It is also connected to an Analog Discovery 2 which is used as an oscilloscope which in turn is then connected to a Raspberry Pi 4. This setup allows us to continuously measure and log our data. The accuracy is reported to be 2% For more detail see [3].

CloudFree EU smart Plug (Plug): is used, as a lower-priced hardware-based measuring instrument, which also has greater ease of use than the AC Current Clamp setup. We have not found any information about the accuracy or sampling rate og the plug.[26]

Scaphandre (SCAP): is described as a monitoring agent that can measure energy consumption.[27] It is designed for Linux where it can use Powercap RAPL, a Linux kernel subsystem that reads data from RAPL. Additionally, SCAP can measure the energy consumption of some virtual machines, specifically Qemu and KVM hypervisors. A driver also exists for installing RAPL on Windows[28]. This enables SCAP to be used on a Windows computer, where the sensor is RAPL, which utilizes the MSRs to update its counters.

The Windows version of SCAP has some limitations but can report the energy consumption of the power domain PKG using the MSR *MSR_PKG_ENERGY_STATUS*. Moreover, it can estimate the energy consumption for individual processes. SCAP accomplishes this by storing CPU usage statistics alongside the energy counter values. It then calculates the ratio of CPU time for each Process ID (PID). Using the calculated ratio, SCAP determines the subset of energy consumption estimated to belong to a specific PID.

4.2 Dynamic Energy Consumption (DEC)

Dynamic Energy Consumption (DEC) was utilized in [3, 29] 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 [29] 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 test case is isolated. Using DEC requires also measuring the energy consumption on an idle case. [29]

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.

Shapiro-Wilk Test: was used to examine if the data followed a normal distribution, which is an important assumption for some statistical methods. Prior research suggested that our data wont be normally distributed [3], therefore we expected our data to not be normally distributed, this was tested using the Shapiro-Wilk test. Understanding the distribution of the data helped choose 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: to assess the correlation between our measurements, Kendall's Tau correlation coefficient was used. This non-parametric measure of association was chosen because it can evaluate the strength and direction of relationships between ordinal variables, even when the underlying data does not adhere to a normal distribution.[32]

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 **1** and **2** 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 1: 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 2: The specifications for DUT 2

When running the experiments, the recommendations presented in [9] were followed. These included that the WiFi, CPU 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 our study. 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].

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

Table 3: 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 compiler based on the GNU GCC project, designed to compile code for execution on Windows. 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 Test cases

Our work employed microbenchmarks and macrobenchmarks to asses the measuring instruments. This section outlines the selected test cases 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 first couple of experiments utilized microbenchmarks from the Computer Language Benchmark Game (CLBG)[1] as test cases. The selected test cases encompassed both single- and multi-threaded microbenchmarks. A challenge in choosing test cases involved ensuring compatibility with the chose 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 CLBG as input for each test case. The chosen microbenchmark test cases and their abbreviation are presented in Table 4. During compilation, the only parameter given is `-openmp` for the multi-core test cases, 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 4: Microbenchmarks

Macrobenchmarks: are large-scale benchmarks that test 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 benchmarks in a real-world ap-

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

plication, which provides a more realistic test case. TBC

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.

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

Table 5: Background Processes

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 5.

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 **A**, with the results stored in the database introduced in Appendix **B**. During the experiments, the `ProcessPriorityClass` for the measuring instrument, framework, and test cases was set to `High`, unless specified otherwise by the particular experiment.

5.1 Experiment One

The first experiment investigated **RQ1**. This experiment employed both multi-core test cases presented in Section **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.

Initial Measurements: As was presented in **??**, Cochran's formula was used to ensure 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[42]. In this experiment, the process' priority class for the framework, the test case, and the measuring instrument were set to `HIGH`.

Initial Measurements		
Name	Fannkuch Redux	Mandelbrot
Clang	61.086	40
MinGW	1.644	3
oneAPI	550	222
MSVC	2.994	10

Table 6: The required samples to gain confidence in the measurements made by IPG on Windows

After 30 measurements, the results from Cochran's formula can be seen in Table **6**, where it was evident that the required samples varied between compilers and test cases. When the test cases were analyzed it was found that MB deviates less than FR, with MB requiring as little as 3 samples 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 energy consumption for FR, but the highest for MB. oneAPI also displayed the lowest energy consumption. As a result, 550 additional measurements were conducted for the next step.

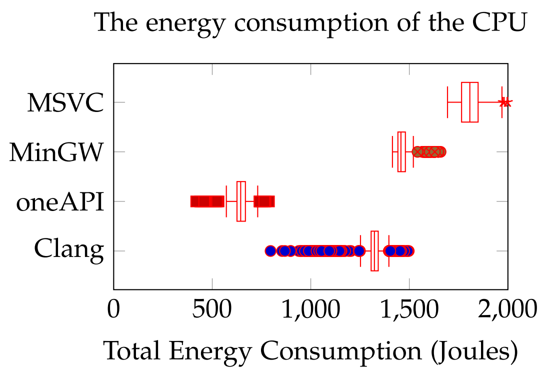

Figure 1: CPU measurements by IPG on DUT 1 for test case(s) FR

Results: After 550 measurements were obtained, the reported values by Cochran's formula still indicated that MSVC, MinGW, and Clang needed more measurements. 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.

When looking at the results for FR in Figures **1** and **2**, and for MB in Appendix **C**, oneAPI had the lowest DEC and total energy consumption for both test cases. Clang deviated the most in Figure **2**.

In the first experiment, it was concluded that the different compilers have a huge impact on the energy consumption but also how many measurements were required to be confident in the results. In the end, oneAPI had the lowest energy consumption and was used in the remaining experiments, unless otherwise specified.

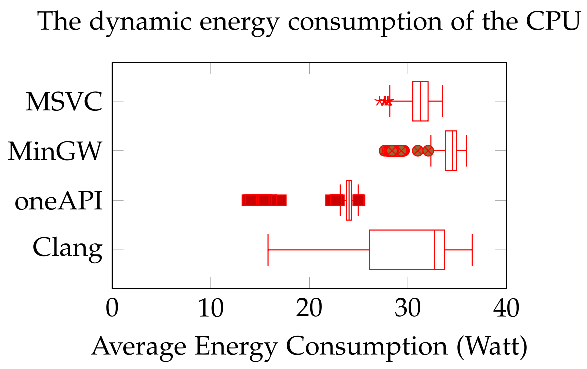

Figure 2: CPU measurements by IPG on DUT 1 for test case(s) FR

5.2 Experiment Two

In this second experiment, the best measuring instrument on Windows will be found between those introduced in Section **4.1**, in order to anwer **RQ2**. The best measuring instrument will be defined as a combination of which measuring instrument will be most correlated with the ground truth, but aspects like ease of use was also considered. Due to some issues with the measuring instrument Scaphandre, the process priority class of the test case was in this experiment set to `Normal`. Due to an execution of less than a second for `Mandelbrot` when compiled on Intel's oneApi, `Mandelbrot`'s parameter will be changed from 16.000 to 64.000 which takes the duration of the test case to $\sim$ 14 seconds. This is to avoid a case where the Plug only has a single data point when measuring. For this experiment, `Fannkuch-Redux` will be run for 550 times, while `Mandelbrot` will be run for 222 times, decided based on Table **6**.

Initial Measurements : Following the initial measurements, Cochran's formula was applied, in order to find how many measurements would be required to gain confidence. This can be found in Appendix **E**, where for some measuring instruments more measurements were required, and for others there were enough. In cases where there were not enough measurements, the number from Cochran's formula was used to decide how many measurements each measuring instrument needed respectively. When looking at how many measurements are required for each measuring instrument, the Clamp requires the most measurements and will in one case require 44.106 measurements. Given how much time would be required to do that, a more in depth look was taken into this measuring instrument. In Figure **3** the evolution of the median, percentiles and whiskers can be seen when plotting from $100 - 1000$ measurement. For the evolution of the median, it decreases by 3.38% between the median at 100 measurements and 1000 measurements, and by 1.03% between 900 and 1000 measurements. A pattern can also be observed, where the median decreases as more measurements are made, but given how little the median changes, the argument is made that if Cochran's formula states that more than 1.000 measurements are required, it will be capped at that.

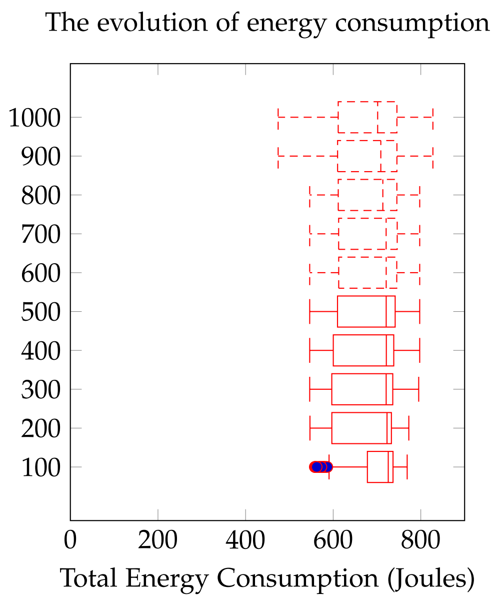

Figure 3: A visual representation of how the energy measurements evolve as more measurements are made by clamp on DUT 2 for test case MB

Results: When analyzing the results, graphs will be shown for the DEC for DUT 1, where additional plots can be found in Appendix **D**.

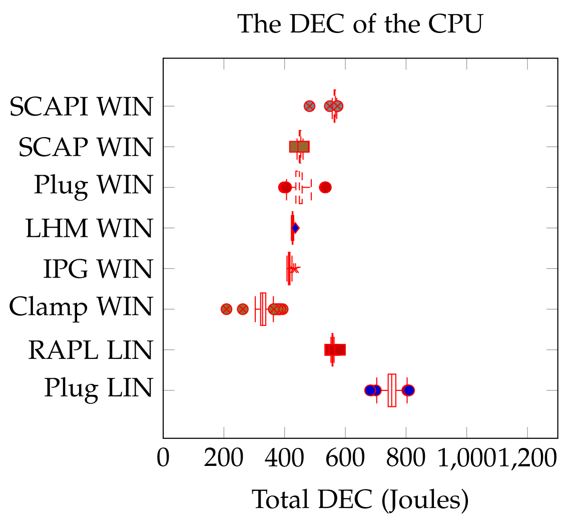

Figure 4: CPU measurements on DUT 1 for test case(s) MB compiled on oneAPI

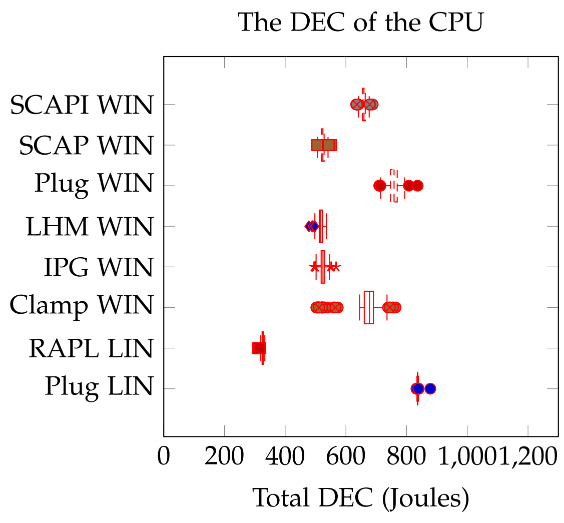

Figure 5: CPU measurements on DUT 1 for test case(s) FR compiled on oneAPI

5.3 Experiment Three

The third experiment will answer RQ3-4. This will done by taking a look at the per-core performance of the two CPU's used in this work. This will be tested using single-core test cases introduced in Section **4.6**, by running each test case on one core at a time, while measuring the energy consumption using IPG and Clamp. This will show how the performance is between P- and E-cores, and how the performance is between cores with the same specifications.

Per-Core Initial Measurements: The first measurements made, will be in order to compare the per-core performance, where 250 measurements will be made for each test case on each core. After 250 measurements, more measurements were made where it was required, as can be found in Appendix **G**, with an upper limit of 1000 measurements.

Per-Core Results: When presenting the results, it will be based on DUT 2 and test case SN, where the other results can be found in Appendix **F**. The CPU in DUT 2 is the one with both P- and E-cores, where the difference in performance can be observed in Figure **6**, Figure **7** and Figure **8** showing the DEC, DEC per second and duration respectively. When comparing between P and E cores, the duration is on average is 76.26% lower on P cores, the energy consumption is 70.44% lower on P cores over the entire duration, while E cores has a 72.88% lower energy consumption per second. When comparing cores of the same type, the largest difference between the best and worst performing core was found on DUT 2, with test case NB, where the performance was 11.61% worse on core 2 than core 7. The lowest difference was found on DUT 2, test case NB on a P core, where the energy consumption was 1.17% higher on core 6 than core 10.

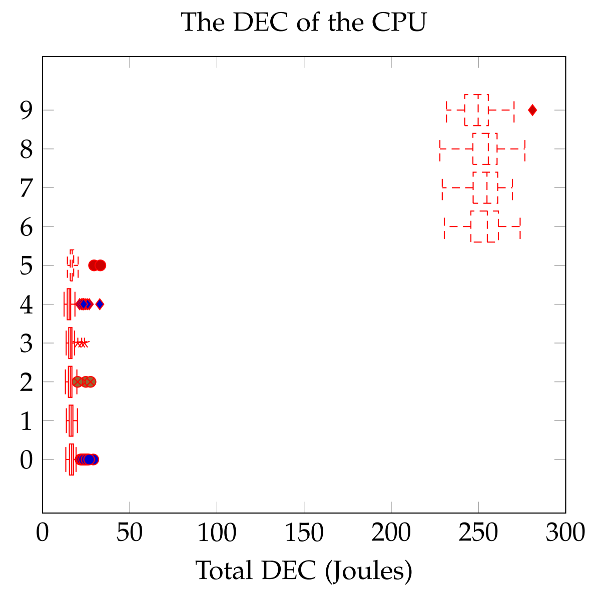

Figure 6: CPU measurements by IPG on DUT 2 for test case(s) SN compiled on oneAPI

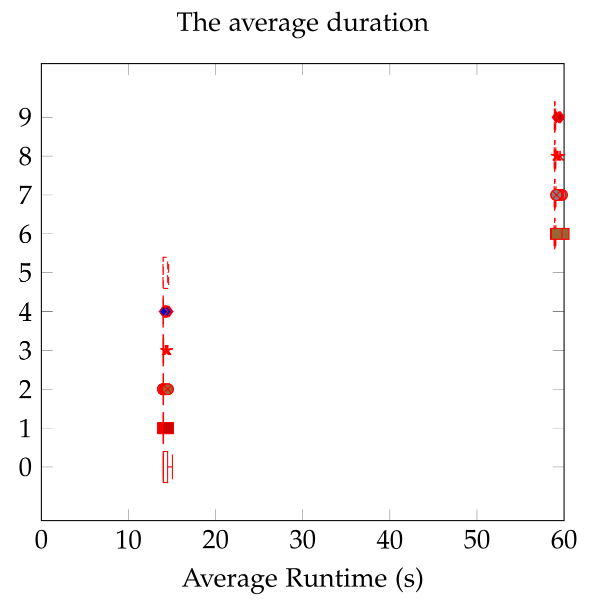

Figure 8: Runtime measurements by IPG on DUT 2 for test case(s) SN compiled on oneAPI

6 Results

7 Discussion

8 Conclusion

Acknowledgements

9 Future Works

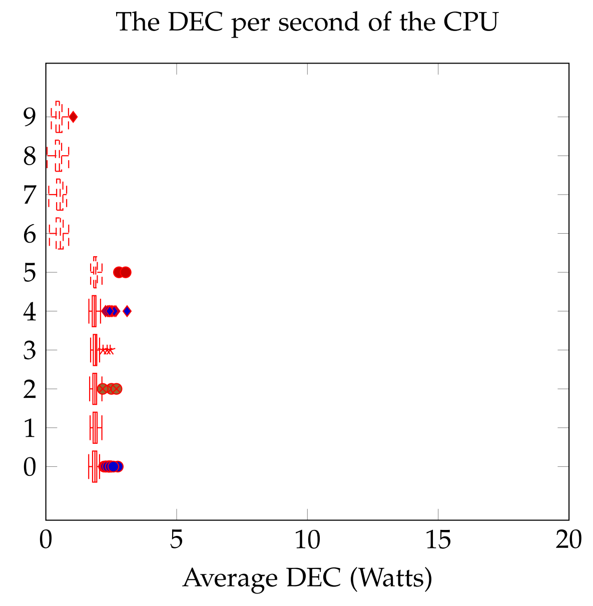

Figure 7: CPU measurements by IPG on DUT 2 for test case(s) SN compiled on oneAPI

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 (Oct. 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. Lima, E., Xavier, T., Faustino, A. & Ruiz, L. *Compiling for performance and power efficiency* in (Sept. 2013), 142–149.
12. Cooper, K. & Waterman, T. Understanding Energy Consumption on the C62x (Jan. 2004).
13. 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.
14. 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.
15. Intel. *https://www.intel.com/content/dam/develop/external/us/en/doc hill-sw-20-185393.pdf* https://www.intel.com/content/dam/develop/external/us/en/documents/green-hill-sw-20-185393.pdf. 2011. 07/03/2023.
16. 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.
17. 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.
18. ARM. *Processing Architecture for Power Efficiency and Performance* https://www.arm.com/technologies/big-little. 09/03/2023.
19. 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.
20. Rotem, E. *et al.* Intel alder lake CPU architectures. *IEEE Micro* **42,** 13–19 (2022).
21. *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.
22. Karl-Bridge-Microsoft. *Scheduling priorities - win32 apps* https://learn.microsoft.com/en-us/windows/win32/procthread/scheduling-priorities. 03/03/2023.
23. Michael Womack, A. W. *Parallel Programming and Performance Optimization With OpenMP* https://passlab.github.io/OpenMPProgrammingBook/cover.html. 03/03/2023.
24. Mozilla. *tools/power/rapl* https://firefox-source-docs.mozilla.org/performance/tools_power_rapl.html. 24/02/2023.

25. source, O. *Libre Hardware Monitor* https : / / github . com / LibreHardwareMonitor / LibreHardwareMonitor. 03/03/2023.
26. CloudFree. *CloudFree EU Smart Plug* https://cloudfree.shop/product/cloudfree-eu-smart-plug/. Accessed: 10/03/2023.
27. Hubblo. *Scaphandre* https://github.com/hubblo-org/scaphandre. 23/02/2023.
28. Hubblo. *windows-rapl-driver* https://github.com/hubblo-org/windows-rapl-driver. 23/02/2023.
29. Fahad, M., Shahid, A., Manumachu, R. R. & Lastovetsky, A. A comparative study of methods for measurement of energy of computing. *Energies* **12,** 2204 (2019).
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. Accessed: 24/03/2023.
42. *Central Limit Theorem* https://sphweb.bumc.bu.edu/otlt/mph-modules/bs/bs704_probability/BS704_Probability12.html. 20/03/2023.

A The Framework

The framework used in this work is an improvement on what was used in [3]. A key difference is how the framework used in this work is a command line tool, supporting all languages, as opposed to in [3] where only C# was supported. 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. BDE was made as a command line tool, based on assumption that most people interested in using it, would belong to a demographic experienced in using the console.

```
.\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 the configuration is chosen, the parameter `--config` specifies the location of a valid json of the format seen in Listing **5**. In Listing **5**, it is possible to specify multiple paths to executable files and assign each executable file with a parameter in `TestCasePaths` and `TestCaseParameter` respectively. Information like the `compiler`, `language`, etc can also be specified about the test case in the configuration. It is also possible to specify the affinity of the test case through `AllocatedCores`, where an empty list represents the use of all cores and the list `1,2` specifies how the test case can only execute on core one and two. When multiple affinities are specified, each test case will be run on both. Limits for the temperature the test cases 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,
    "TestCasePaths": [
        "path/to/one.exe", "path/to/two.exe"
    ],
    "AllocatedCores": [
        [], [1,2]
    ],
    "TestCaseParameters": [
        "one_parameter", "two_parameter",
    ],
    "UploadToDatabase": true,
    "BurnInPeriod": 0,
    "MinimumTemperature": 0,
    "MaximumTemperature": 100,
    "DisableWifi": false,
    "ExperimentNumber": 0,
    "ExperimentName": "testing-phase",
    "ConcurrencyLimit": "multi-thread",
    "TestCaseType": "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 `TestCasePath` and `TestCaseParameter`. This way of using BDE is based on the assumption that when using BDE, the configuration will in most cases be the same.

```
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

Since this work uses both Windows and Linux, BDE should support both. This is supported by introducing the `IDutService` seen in Listing **6**, where all OS dependent operations are located. This includes the ability to enable and disable the WiFi and stop background processes. Furthermore, the measuring instruments chosen in the configuration are validated by checking if that measuring instrument is valid on the current OS. 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.

```
public class MeasuringInstrument
{

    public (TimeSeries, Measurement) GetMeasurement()
    {
        var path = GetPath(_measuringInstrument, fileCreatingTime);
        return ParseData(path);
    }

    public void Start(DateTime fileCreatingTime)
    {
        var path = GetPath(_measuringInstrument, fileCreatingTime);

        StartMeasuringInstruments(path);

        StartTimer();
    }

    public void Stop(DateTime date)
    {
        StopTimer();
        StopMeasuringInstrument();
    }
    internal virtual int GetMilisecondsBetweenSampels()
    {
        return 100;
    }

    internal virtual (TimeSeries, Measurement) ParseData(string path) { }

    internal virtual void StopMeasuringInstrument() { }

    internal virtual void StartMeasuringInstruments(string path) { }

    internal virtual void PerformMeasuring() { }
}
```

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

In addition to multiple OSs, multiple measuring instruments are also supported on BDE. This is supported by inheriting the class `MeasuringInstrument` for all measuring instruments and then overriding the necessary methods. `MeasuringInstrument` implements a start and stop method, and a method to get the data measured between the start and stop. In terms of the virtual methods, each measuring instrument needs to override, these are measuring instruments specific. This includes a start and stop method, and a method to parse the measurement data. In addition to this, it includes a method `PerformMeasuring`, which is called every 100ms by default. The idea behind this method is for measuring instruments like RAPL and LHM to make a measurement each time it is called, whereas for other measuring instruments like IPG, this is done by the measuring instrument itself. The default 100ms interval between samples can be changed by overriding `GetMiliecondsbetweenSamples`.

```
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

Following the introduction to the configuration and the measuring instruments, 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 test cases 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.

B 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 **9**, where the `MeasurementCollection` table defines under which circumstances the measurements were made under. This includes which measuring instrument was used, which test case 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 `TestCase`, 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 test case, while the `Sample` represents samples taken during the execution of the test case. 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 fine-grained representation of the measurement.

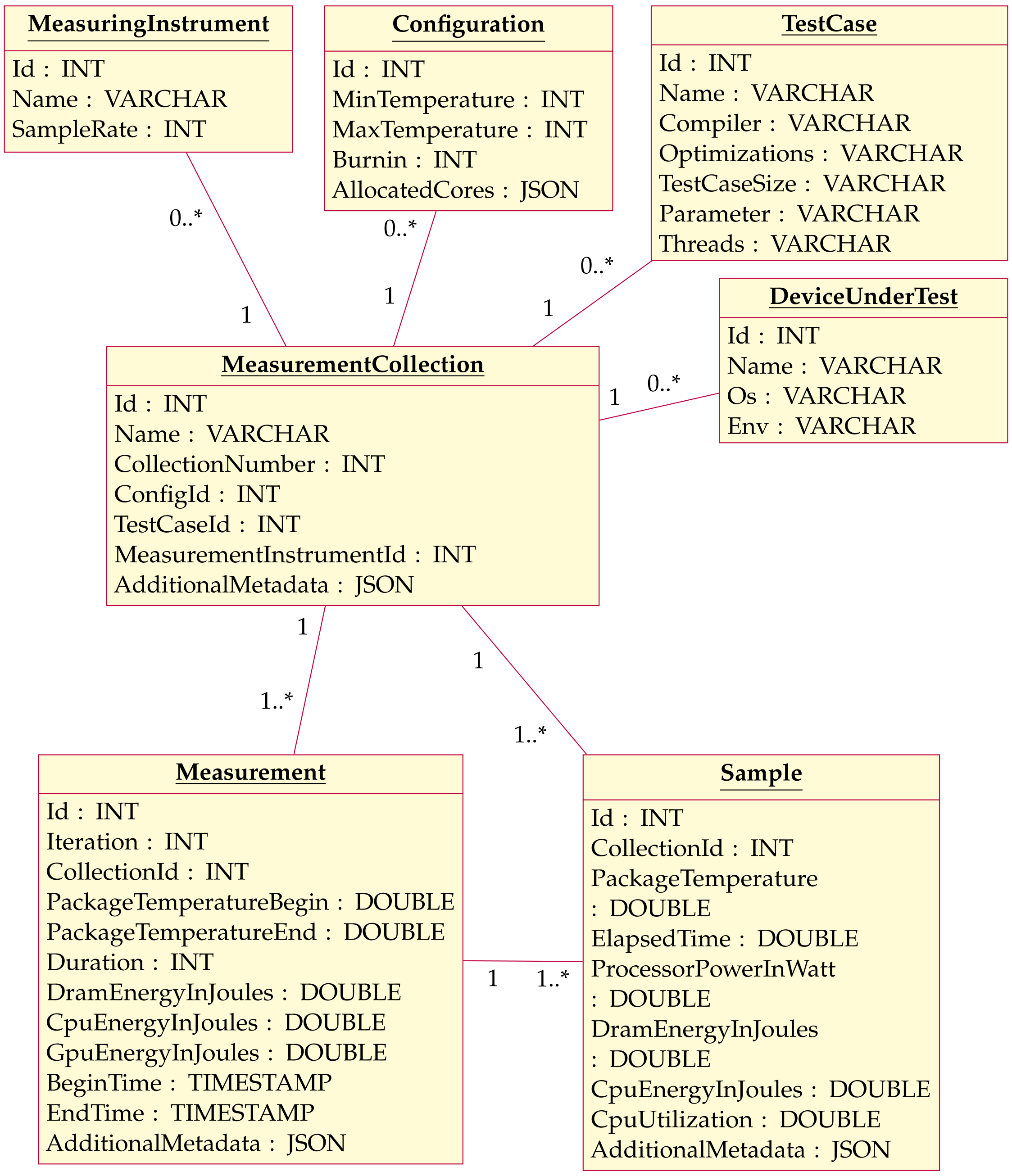

Figure 9: An UML diagram representing the tables in the SQL database

C Experiment One

Measurements made on test case Mandelbrot for the first experiment, found in Section **5.1**.

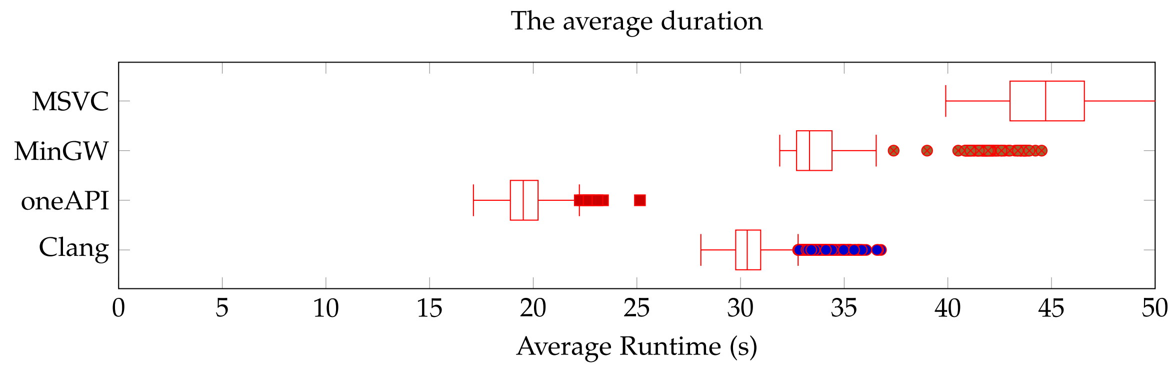

Figure 10: Runtime measurements by IPG on DUT 1 for test case(s) FR

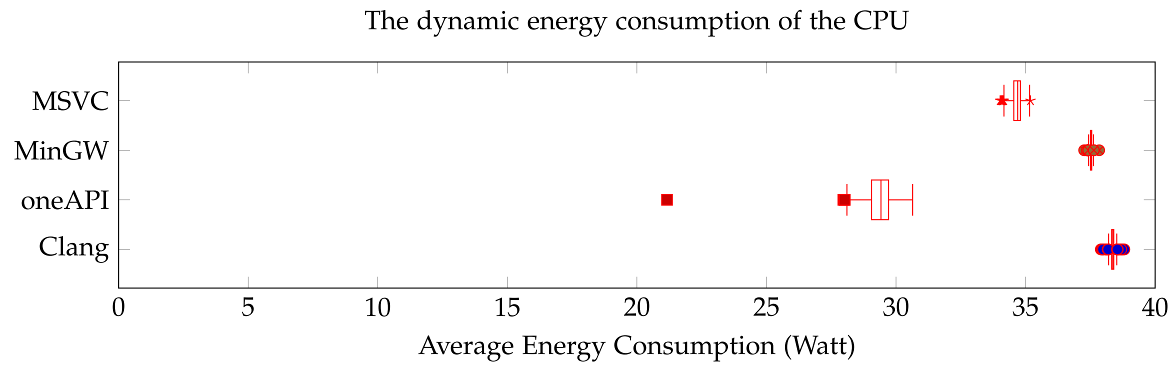

Figure 11: CPU measurements by IPG on DUT 1 for test case(s) MB

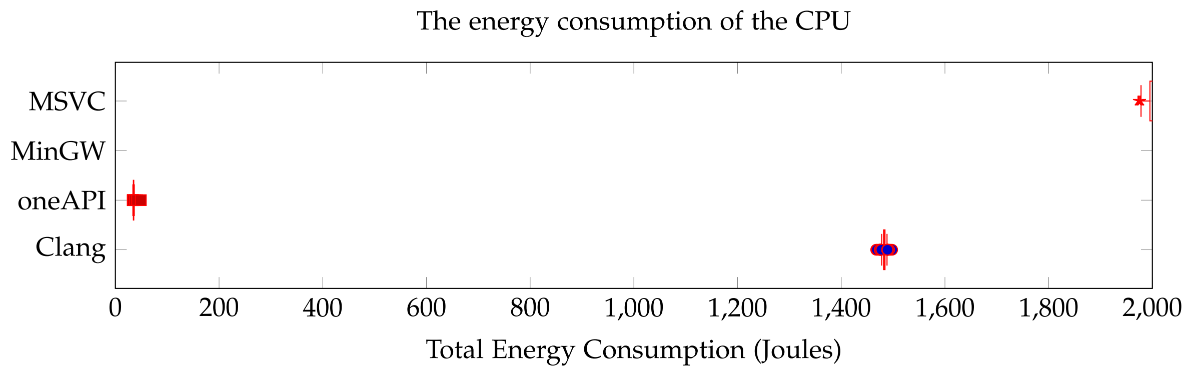

Figure 12: CPU measurements by IPG on DUT 1 for test case(s) MB

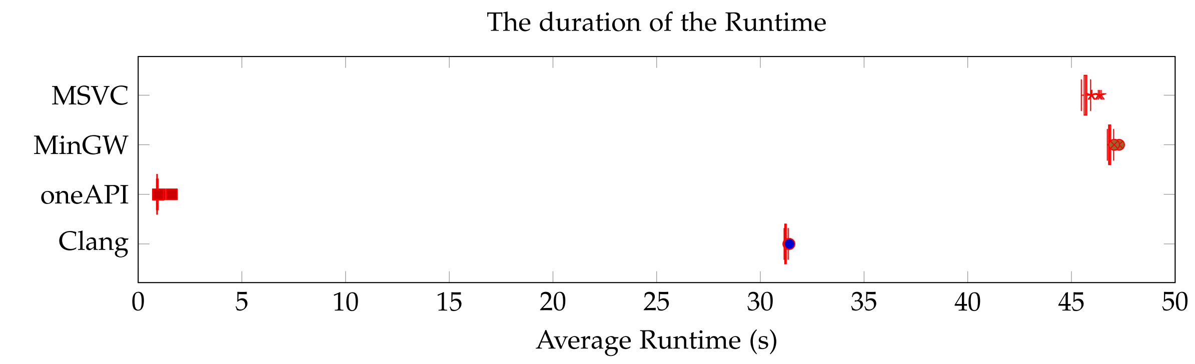

Figure 13: Runtime measurements by IPG on DUT 1 for test case(s) MB

D Experiment Two

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

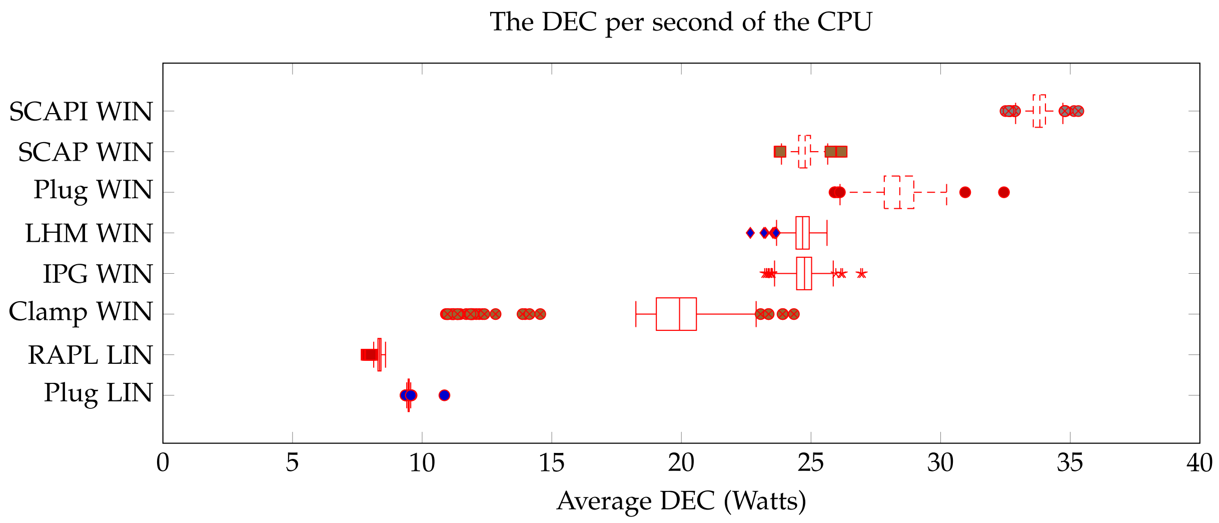

Figure 14: CPU measurements on DUT 1 for test case(s) FR compiled on oneAPI

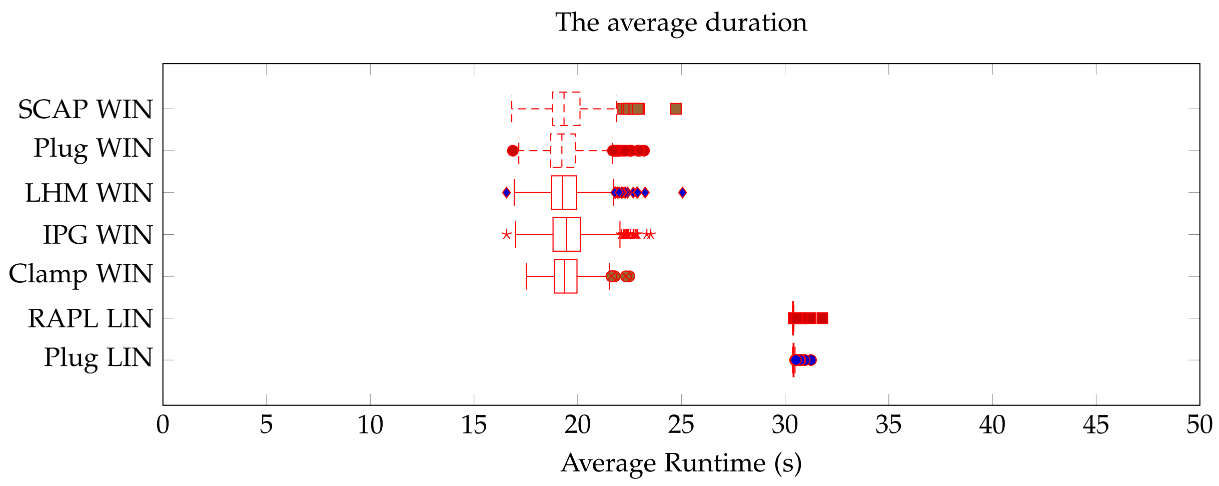

Figure 15: Runtime measurements on DUT 1 for test case(s) FR compiled on oneAPI

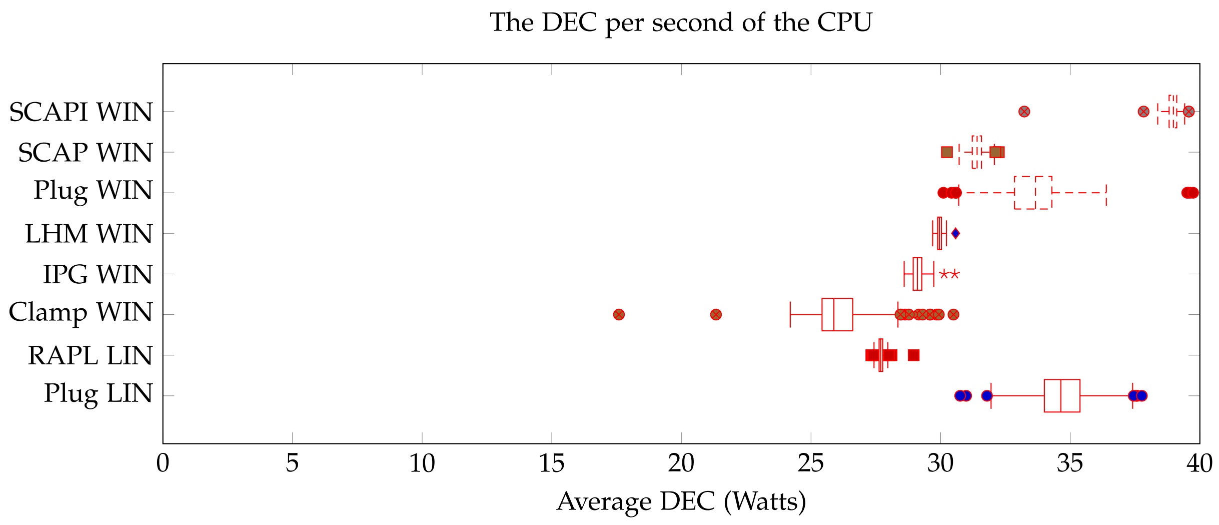

Figure 16: CPU measurements on DUT 1 for test case(s) MB compiled on oneAPI

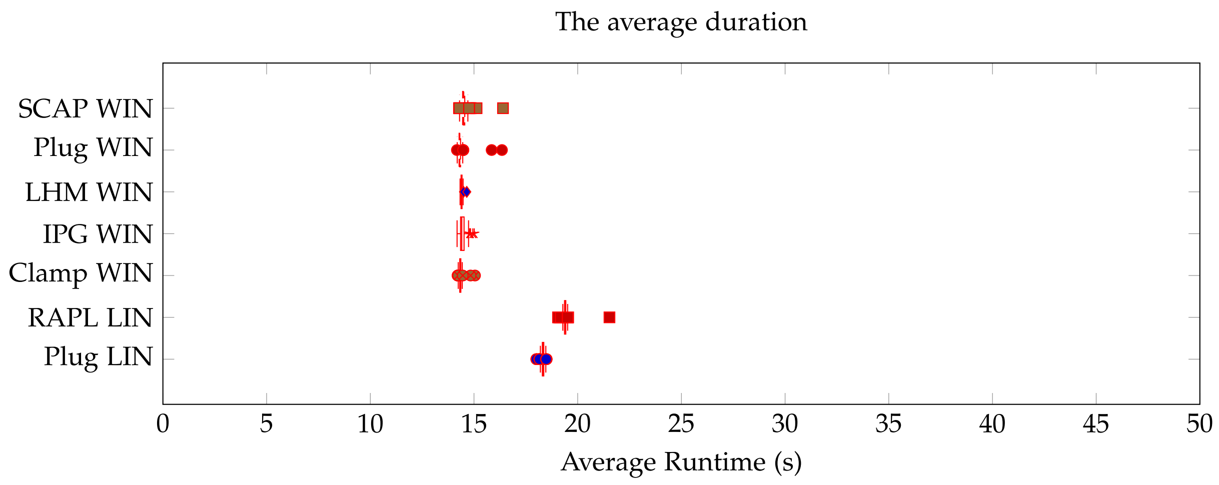

Figure 17: Runtime measurements on DUT 1 for test case(s) MB compiled on oneAPI

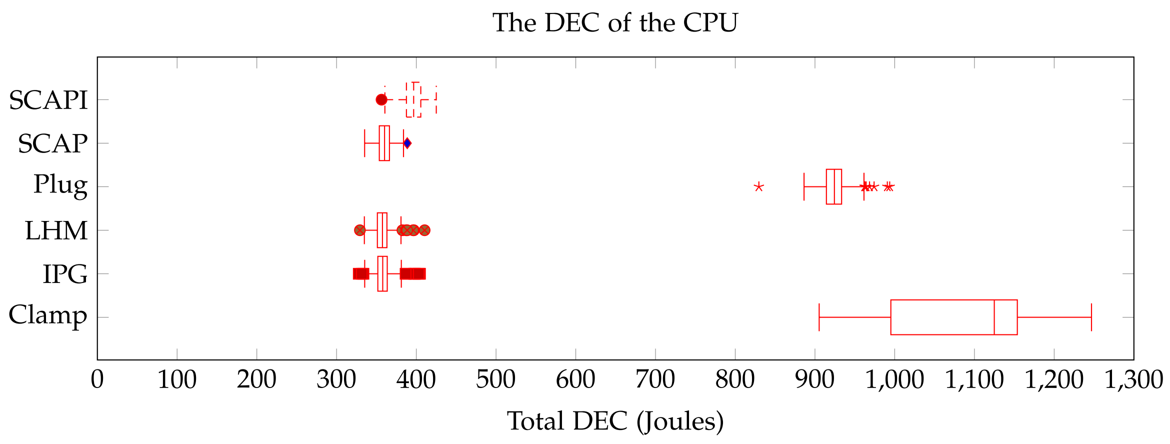

Figure 18: CPU measurements on DUT 2 for test case(s) FR compiled on oneAPI

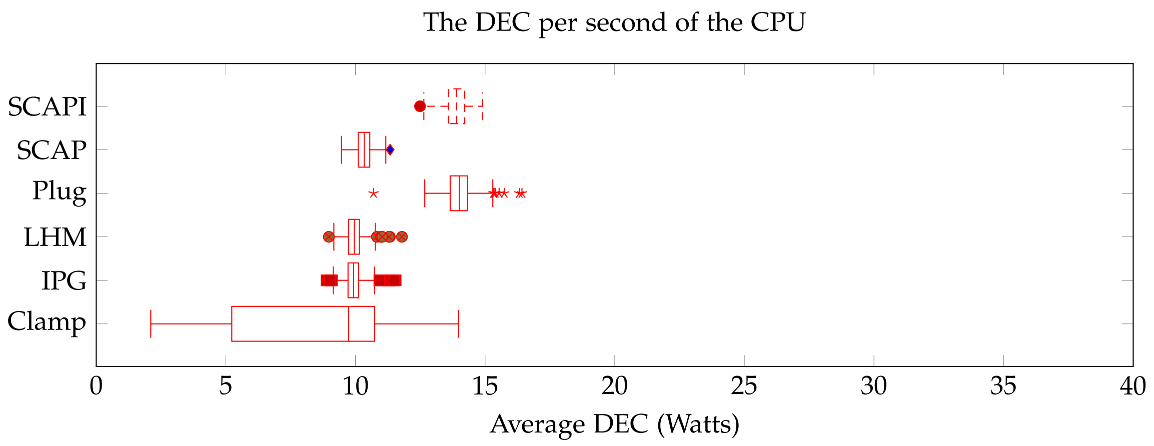

Figure 19: CPU measurements on DUT 2 for test case(s) FR compiled on oneAPI

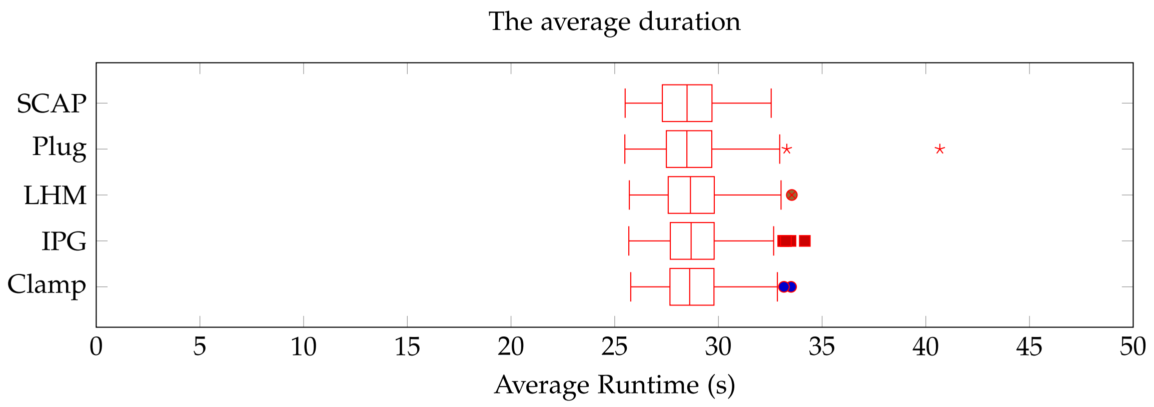

Figure 20: Runtime measurements on DUT 2 for test case(s) FR compiled on oneAPI

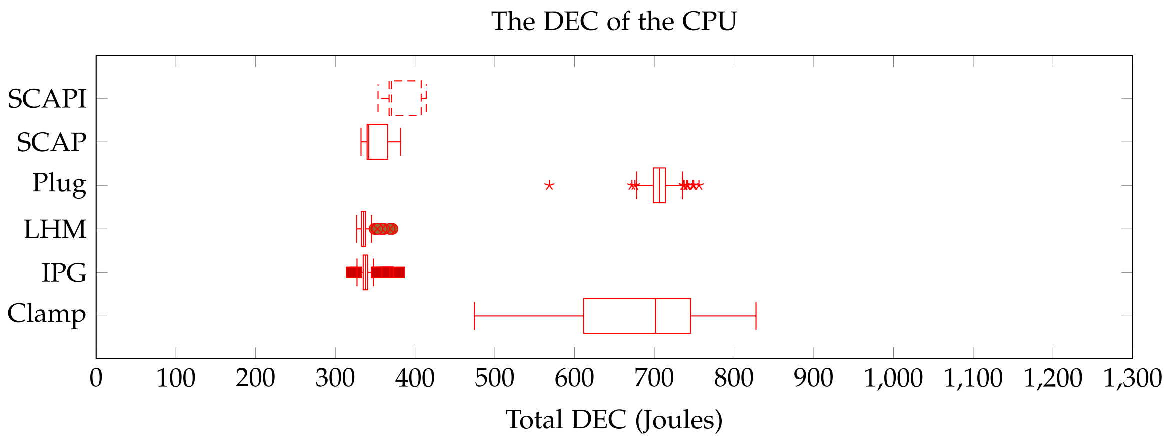

Figure 21: CPU measurements on DUT 2 for test case(s) MB compiled on oneAPI

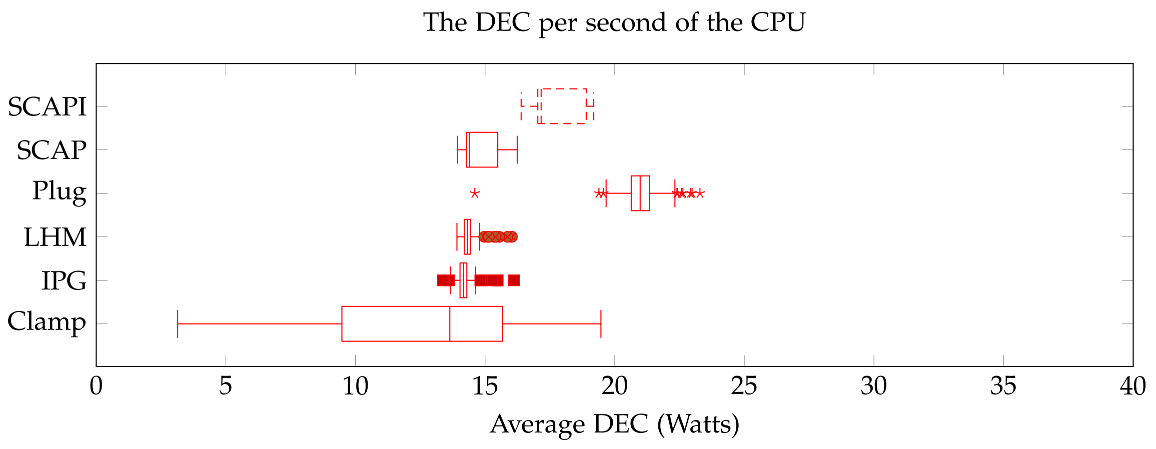

Figure 22: CPU measurements on DUT 2 for test case(s) MB compiled on oneAPI

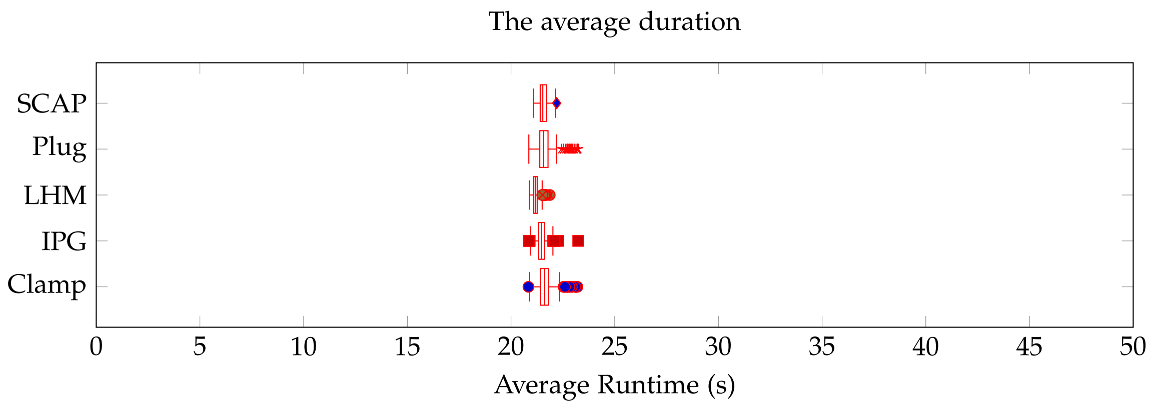

Figure 23: Runtime measurements on DUT 2 for test case(s) MB compiled on oneAPI

E 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 Section **5.2**.

Initial Measurements		
Name	Fannkuch Redux	Mandelbrot
Plug (W)	3.224	1.790
Plug (L)	???	5.889
Clamp (W)	3.396	2855
Clamp (L)	???	???
RAPL	26	53
SCAP	459	74
SCAPI	453	153
IPG	751	216
LHM	615	45

Table 7: The required samples to gain confidence in the measurements made by the different measuring instruments, on both OSs for DUT 1

Initial Measurements		
Name	Fannkuch Redux	Mandelbrot
Plug (W)	916	1.088
Plug (L)	???	???
Clamp (W)	4.942	44.106
Clamp (L)	???	???
RAPL	???	3
SCAP	416	1.628
SCAPI	840	3.796
IPG	379	88
LHM	379	31

Table 8: The required samples to gain confidence in the measurements made by the different measuring instruments, on both OSs for DUT 2

F Experiment Three

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

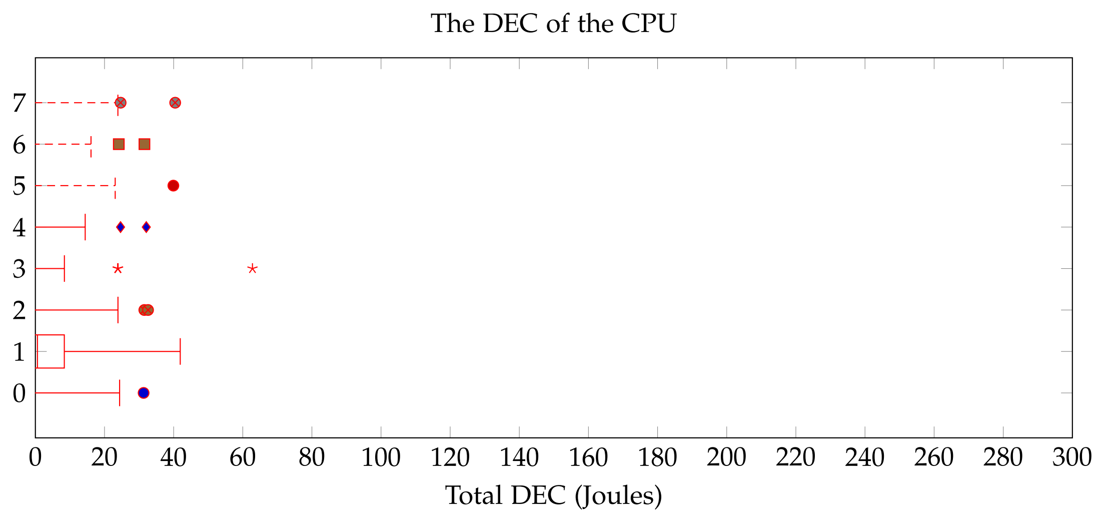

Figure 24: CPU measurements by IPG on DUT 1 for test case(s) NB compiled on oneAPI

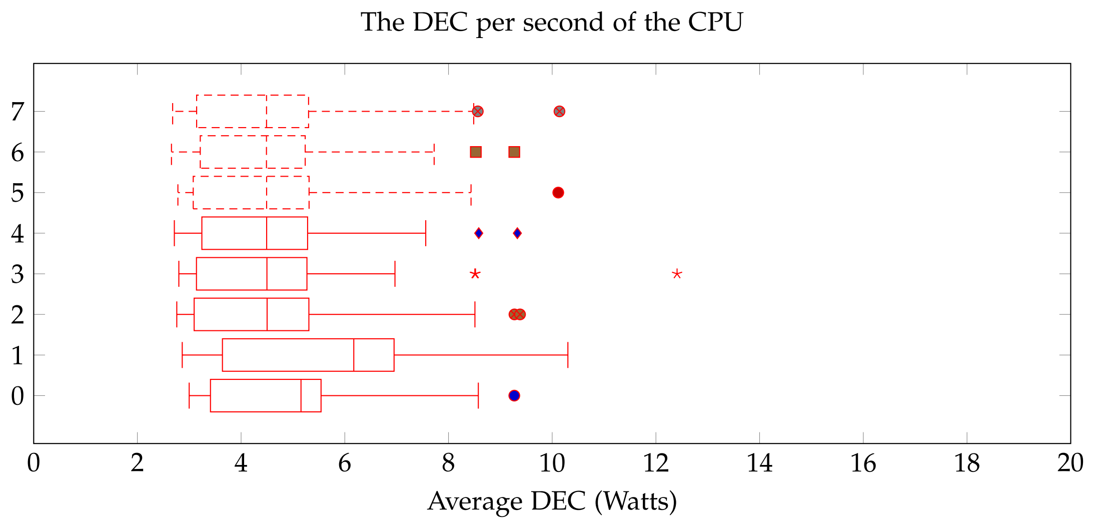

Figure 25: CPU measurements by IPG on DUT 1 for test case(s) NB compiled on oneAPI

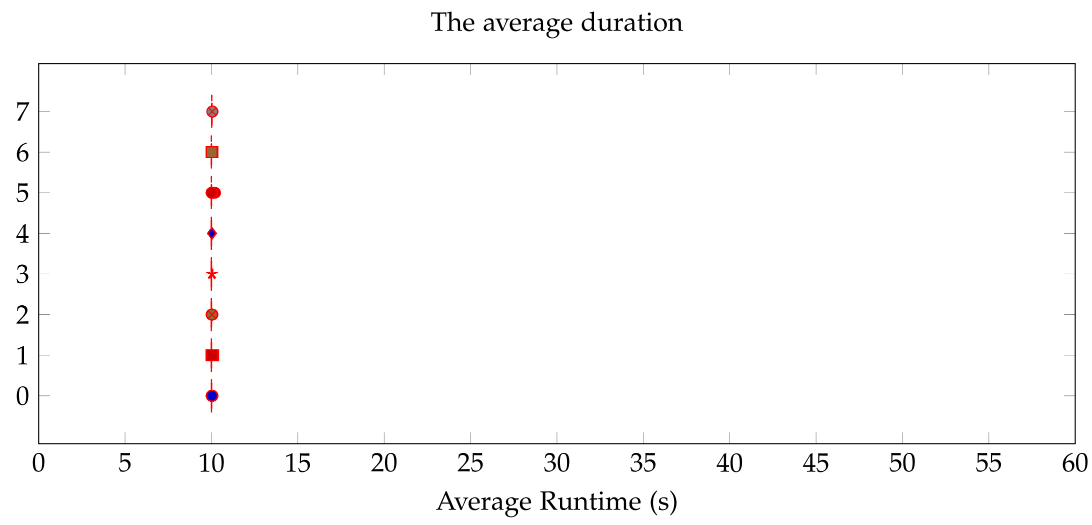

Figure 26: Runtime measurements by IPG on DUT 1 for test case(s) NB compiled on oneAPI

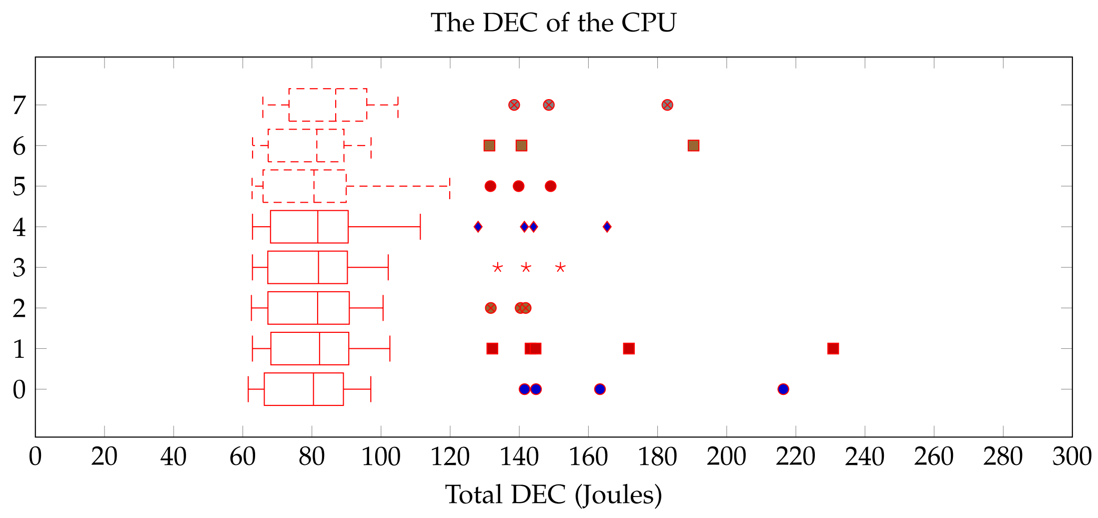

Figure 27: CPU measurements by IPG on DUT 1 for test case(s) SN compiled on oneAPI

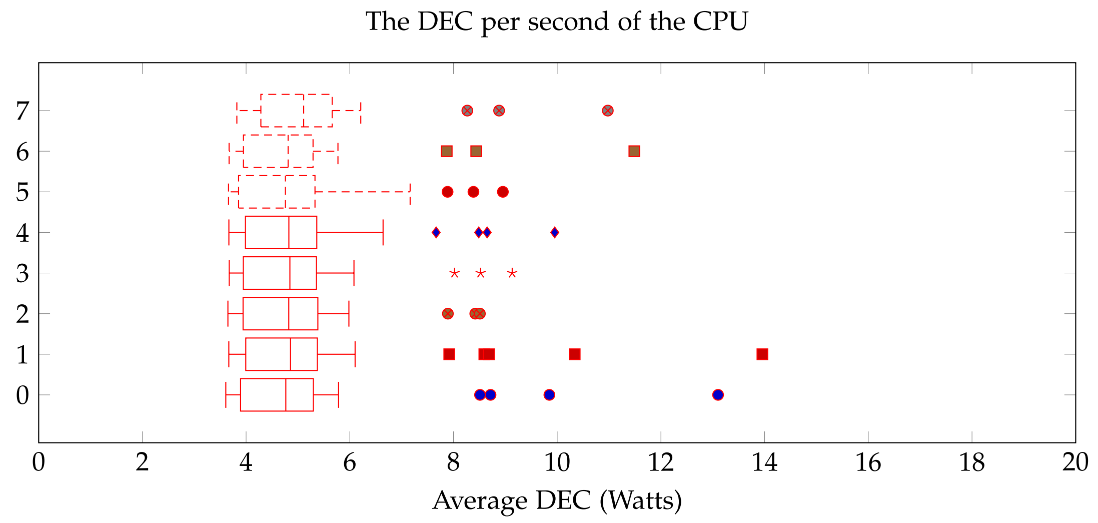

Figure 28: CPU measurements by IPG on DUT 1 for test case(s) SN compiled on oneAPI

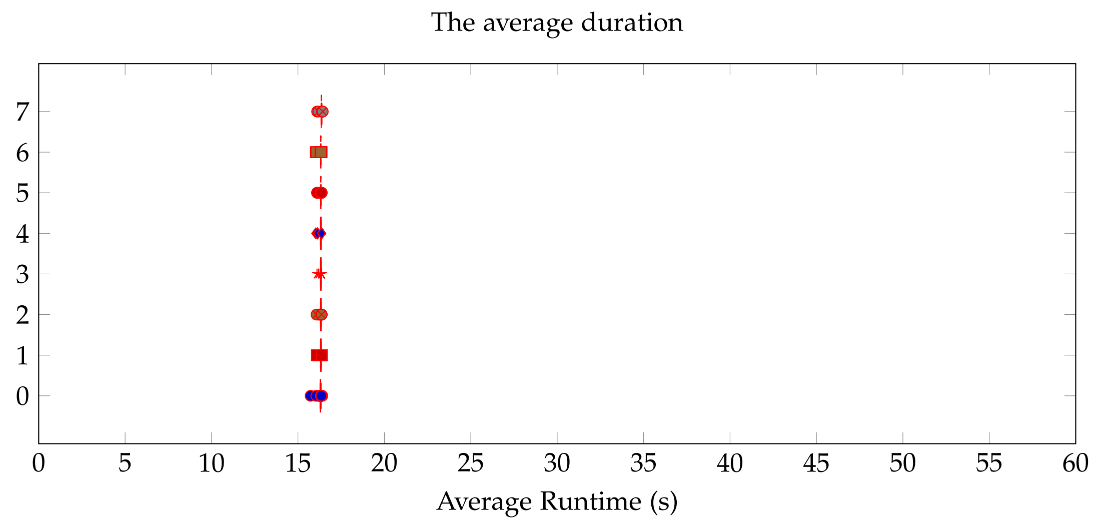

Figure 29: Runtime measurements by IPG on DUT 1 for test case(s) SN compiled on oneAPI

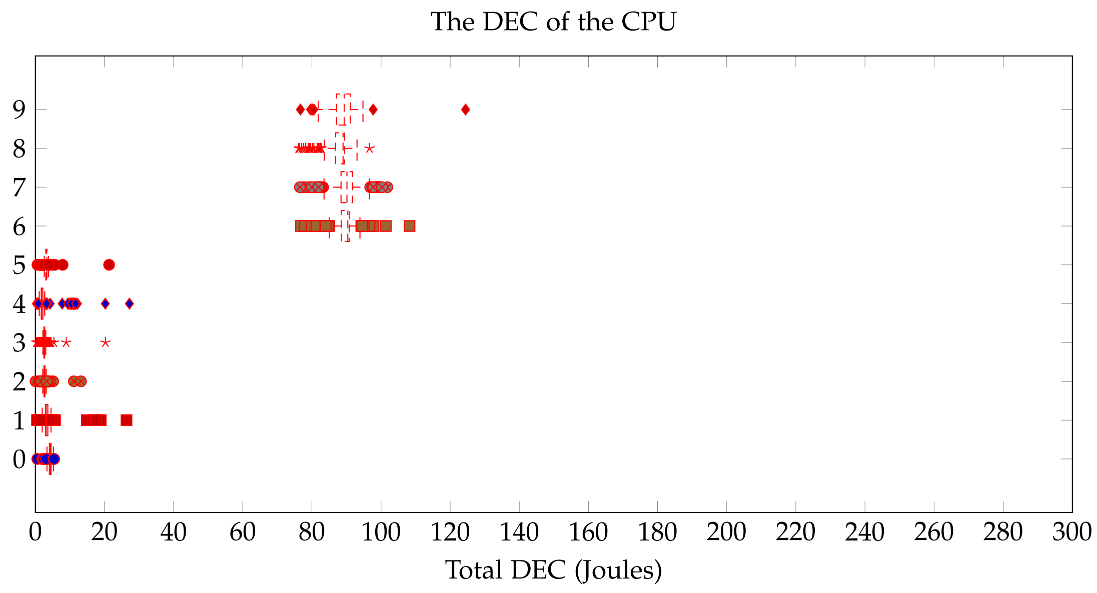

Figure 30: CPU measurements by IPG on DUT 2 for test case(s) NB compiled on oneAPI

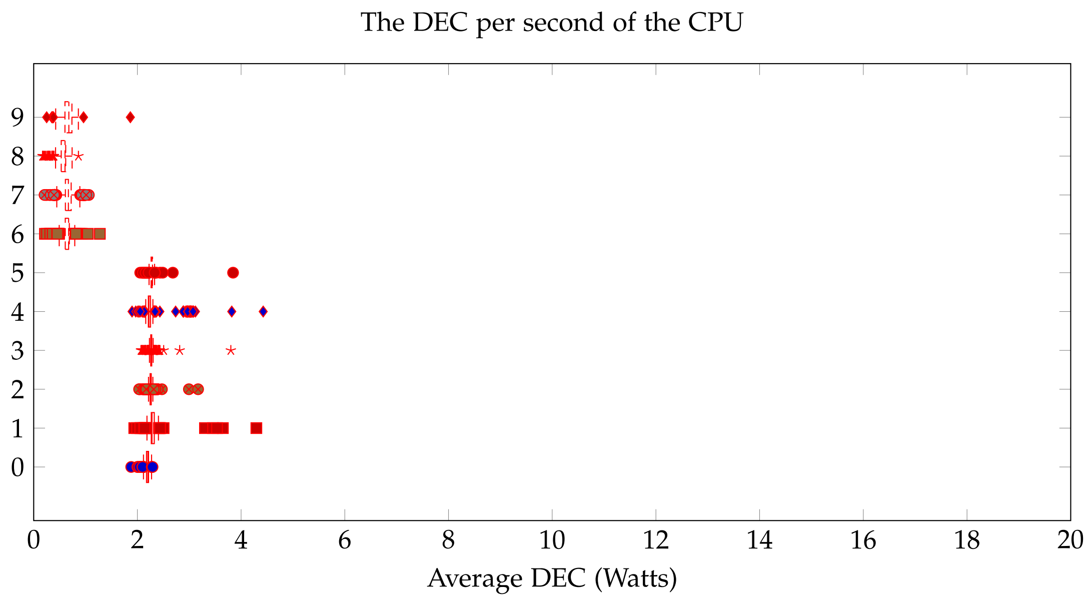

Figure 31: CPU measurements by IPG on DUT 2 for test case(s) NB compiled on oneAPI

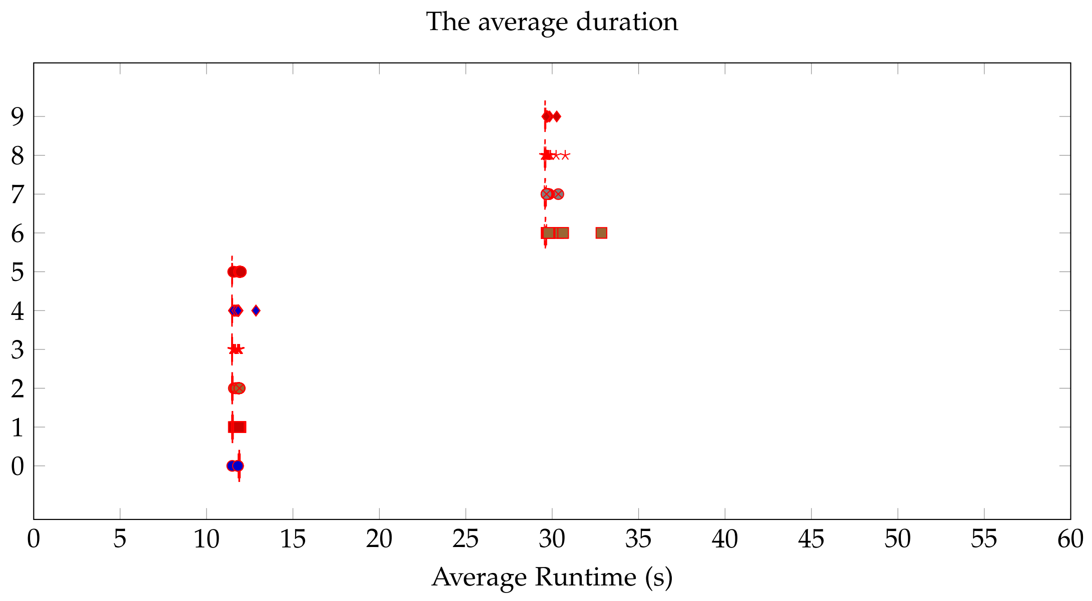

Figure 32: Runtime measurements by IPG on DUT 2 for test case(s) NB compiled on oneAPI

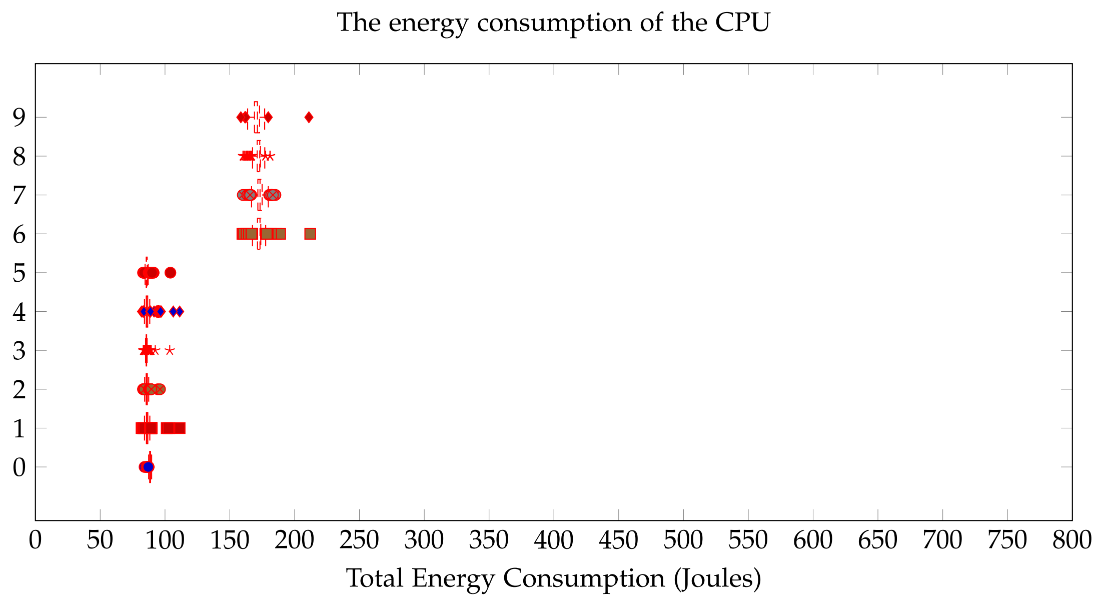

Figure 33: CPU measurements by IPG on DUT 2 for test case(s) NB compiled on oneAPI

G Initial Measurements in Experiment Two

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 Section **5.3**.

Initial Measurements		
Name	NBody	Spectral-Norm
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 9: The required samples to gain confidence in the measurements made by IPG in different cores for DUT 1

Initial Measurements		
Name	NBody	Spectral-Norm
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 10: The required samples to gain confidence in the measurements made by IPG in different cores for DUT 2