Universiteit Leiden
The Netherlands

Opleiding Informatica

Energy Efficiency across Programming Languages Revisited

Emiel Beinema

Supervisors:
Kristian Rietveld & Erik van der Kouwe

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

28/08/2018

Abstract

A recent publication discussed the energy efficiency of programming languages and presented a ranking of programming languages by energy efficiency. The paper makes certain assumptions on the relation between the execution time of a program and its energy consumption and only presents measurements performed on a single system, which we argue warrants further research. Additionally, their investigation into the relation between memory usage and DRAM energy consumption did not yield a clear conclusion. We scrutinized the test setup used in the paper, tested the assumptions and investigated the reproducibility of the measurements across different systems. Additionally, we applied a new strategy to the investigation of the relation between memory usage and DRAM energy consumption.

In this thesis, we present a modification to the test setup used in the original paper, to mitigate a significant impact on the measurements. Using our modified setup, we show that variations occur between measurements on different systems that can potentially affect the ranking. We also present two counter-examples to the assumption that a faster implementation of a certain benchmark in a certain language is always more energy efficient. Finally, we show a possible relation between the number of memory instructions and the DRAM energy consumption and we propose a strategy for future investigation of this relation.

Contents

Chapter 1

Introduction

For long, energy efficiency has not been the first concern in application programming. While processor manufacturers[Cla+02] and developers of low-power devices, like digital watches, pocket calculators or implanted devices[Vit94], have concerned themselves with energy consumption before, it previously was no issue of significance in application programming. Concerns about power consumption were typically considered to be relevant only for lower-level layers such as the compiler, operating system and hardware[Sah+12].

Various factors, however, contribute to an emerging interest. Firstly, general-purpose devices are getting smaller and battery lifetime has become one of the key selling points of many types of consumer electronics. Notebooks, smartphones and even smartwatches compete in form-factor and battery life[Saw18]. As an effect, manufacturers all try to develop more powerful batteries and less energy consuming devices, while still making them thinner, lighter and more feature-packed[PLS01]. Secondly, energy consumption in general has become a major topic in society and academics[PC17; SW15]. The growing feeling of urgency about men's contribution to climate change has moved many to explore this field of research.

In a young field of research like this, a lot of challenges are at play. Processes and methods have not had the time to evolve, data is often scarce and the possible causes of observations are not as clear as in more mature fields[Yad18]. A critical look at early literature is therefore of great importance to the development of the field. In this thesis, we will take a close look at "Energy Efficiency across Programming Languages" by Pereira et al. [Per+17a].

In a series of papers, their group is looking to equip developers with the tools they need for energy-aware development[Cou+15; Per+16; Per+17b] and in this particular paper[Per+17a], they compared the energy efficiency of various popular programming languages, to provide developers with rankings of languages based on (combinations of) the potential constraints execution time, memory use and energy efficiency. They find that they are able to relate execution time and peak memory usage to energy usage and to construct a ranking of the programming languages on the energy, time and peak memory characteristics. However, from their results also follows that a language in which a benchmark executes in a shorter time, is not always more energy efficient.

Carefully reading the paper prompted a few questions, which we will investigate in this thesis:

1. Is the fastest always the "most efficient"?

In their paper, Pereira et al. [Per+17a] study a set of test programs implementing various benchmarks in a number of programming languages. This set comprises at most one implementation in each language for a given benchmark. In the paper, the authors describe how they chose their test programs: for each algorithm in each language they tested, they included the fastest implementation in their comparison, which they call the "most efficient". However, they are ranking the languages by energy consumption so it would seem more logical to use the most energy efficient implementations for their comparison. We will explore whether the difference is substantial.

2. Are the results reproducible across different systems?

The results of the measurements by Pereira et al. [Per+17a] are presented in the paper as a tool for developers to help choose a programming language taking into account the constraints that apply. This only works if the results are reproducible across different systems. Because Pereira et al. only performed measurements on a single system, we will test the reproducibility of these measurements by measuring the programs on different systems and comparing the results.

3. Is there a connection between energy consumption and memory accesses?

Pereira et al. [Per+17a] discuss the correlation between the energy consumed by the memory controller and the peak memory usage. They were surprised to find that "the relationship is almost non-existent"[Per+17a]. We investigate a possible correlation between the energy consumption of the memory controller and the amount of read/write actions on the memory to find out whether the amount of reads and writes in a program is a better indicator for its memory energy consumption.

In the process of investigating these questions, the measurement setup and methodology used by the paper was scrutinized. This raised two further questions:

4. Does unnecessary processing in the scheduling script affect the results?

The scheduling script Pereira et al. [Per+17a] uses, processes the entire output of the measured programs without actually using it. Some of the programs generate a large output in a relatively small execution time and processing this data takes considerable system resources. Because we can only measure the power consumption of all the CPU cores at once (see Section 2.1.1), this additional processing could have a significant impact on

the (power) performance of the measurements. We will compare the energy measurements of the original scheduling script with tests run by an adapted script to determine the impact of the unnecessary processing.

5. Does the order of the experiments affect the results?

In their experiments, Pereira et al. [Per+17a] ran the programs one after another in a fixed order, each ten times. This is a very simple method, but is it a fair one, too? Because of caches, frequency scaling, it is not directly clear that the performance of a program cannot be affected by the program that runs before it. Furthermore, a temporary interference, for example a background process, could have a significant impact on the results if it impacts most of the measurements of a single implementation and (almost) no other. A comparison of different orders could offer some certainty.

1.1 Thesis Overview

In this thesis, we will have a close look at a paper in the field of energy efficiency in application programming. We will first present a background on the relevant concepts in Chapter 2. Subsequently, we will visit the test setup that was used and propose a modification to the scheduling script used for the measurements in Chapter 3. Then we investigate whether the results are reproducible across different systems in Chapter 4. Next we revisit two of the research questions in the original paper: in Chapter 5 we will try another strategy to answer the question "Is a faster program always more energy efficient?" and we will investigate the relation between memory usage and DRAM energy consumption by looking at the memory usage from another angle in Chapter 6. Finally, in Chapter 7, we will take a look back at this thesis to highlight the conclusions and suggest some directions for future research.

This bachelor thesis was constructed under supervision of Dr. Kristian Rietveld and Dr. Erik van der Kouwe at the Leiden Institute of Advanced Computer Science.

Chapter 2

Preliminaries

2.1 Measuring energy usage

To be able to compare the energy efficiency of computer programs, we need to decide what energy consumption we actually want to measure and how we can do that. While there are many components in a modern computer, most of the energy that is attributable to a specific application, is consumed by the CPU and RAM. Because of the many other components and their power consumption, a simple power meter does not offer the data we need. We need more granularity, to only measure the components we are interested in, and a certain time resolution, to ensure we measure exactly during the execution of a test program. Furthermore, measuring the entire system would introduce many additional variables, like the efficiency of the power supply, the power usage of a storage device and any peripherals that may be present. This would make the measurements much more specific to the exact test system, and less applicable to other computer systems.

2.1.1 Intel Running Average Power Limit

In the Sandy Bridge generation of Intel Core processors, Intel introduced RAPL: *Running Average Power Limit*. This feature exposes information about real-time power consumption and allows users to set limits.

The data exposed by RAPL is presented for certain domains. The possible domains are:

- the entire CPU package;
- the cores;
- the DRAM (available on some devices);
- a device outside the cores, often used for the on-chip GPU (available on some devices).

The functionality that RAPL provides, varies per domain, but they all support the energy status. This includes a counter that, in some defined unit, counts the energy used in that domain[16, pp. 31–39]. By reading this counter directly before and after running a program, we can calculate its energy usage. Note that the most specific domain to measure core power consumption of a program encompasses all cores on the chip. It is not possible to measure the energy consumption of an individual core in a multi-core CPU. Because of this limitation, measurements will always be influenced by any other (background) processes that may run at the same time.

It is not entirely clear whether the metrics RAPL provides come from real measurements by hardware sensors or estimates based on a predefined model, but Intel RAPL has been compared with physical measurements of the energy consumption and has been found to be very accurate, but with a constant offset[Häh+12].

To read the measurements from Intel RAPL, Linux provides three options. A user can read out the Model Specific Registers (MSRs) directly, by using the *perf_event* interface that exposes various power events or by using the *powercap* interface that is available from Linux 3.13. Pereira et al. [Per+17a] read out the MSRs directly, but because the specifics differ between architectures, we opted to use the Linux powercap interface to make sure the implementation was correct on all our test systems. The adapted measurement program can be found in Appendix B.

2.2 Memory measurements

For our investigation into the correlation between memory accesses and DRAM energy consumption, we needed counts of the number of memory accesses performed. These are provided by Intel as Performance Monitoring Events[16]. From these events we chose to measure `MEM_INST_RETIRED.ALL_LOADS`, all the memory load instructions that were completely executed, `MEM_INST_RETIRED.ALL_STORES`, all the memory store instructions that were completely executed, and `MEM_LOAD_RETIRED.L3_MISS` (called `MEM_LOAD_UOPS_RETIRED.LLC_MISS` in the Ivy Bridge architecture), that counts the number of completely executed misses at the last CPU cache. These events all count so-called "retired" instructions, meaning that they have been fully executed and the results were used. Because of the *speculative execution* strategies included in modern Intel CPUs, it is possible that a number of (load or store) operations are partially performed, without their results being used[18] but Intel does not provide Performance Monitoring Events that include the speculative execution for these categories of instructions. Because the speculative execution mechanism is intended to improve performance, we assume that it performs only a very small number of unnecessary load and store operations and therefore its impact on our measurements is limited. We read these using the Linux perf_event API in another adaptation of the measurement program by Pereira et al. [Per+17a]. The adapted code is included in Appendix C.

2.3 Computer Language Benchmarks Game

The Computer Language Benchmarks Game (CLBG) is a collection of benchmarks written in a set of programming languages, started by Dough Bagley and currently maintained by Isaac Gouy[Gou]. The implementations are crowd-sourced and constantly added or improved. At the time of writing, the game consists of 10 benchmarks and 27 languages. Not all benchmarks have been implemented in all languages, and of many benchmarks multiple implementations in certain languages are included in the game. Pereira et al. [Per+17a] used CLBG mainly because it provided them with a collection of programs in different languages where the authors tried to solve the same problem. The effective work of the various implementations of a given benchmark is therefore the same. We think that this makes for a relatively fair comparison between languages, for it shows what performance experienced programmers obtain using a given programming language. In this thesis, we use the numbering of the implementations as they are presented in the CLBG. Because sometimes implementations are removed from the set, the numbering of the remaining implementations is not always continuous.

Chapter 3

Test setup

3.1 Introduction

While running the test suite on one of our systems, an unexpected Out-of-Memory situation occurred, failing the tests. Alongside the probable cause for the Out-of-Memory problem, we found that the execution order of the measurements could possibly allow other factors to affect the measurements. This was cause for us to scrutinize the methodology and set up our own methodology to test these hypotheses.

3.2 Test setup

The base test setup, that we used unless otherwise specified, was to use our slightly modified version of the scheduling script used by Pereira et al. [Per+17a] as included in Appendix D to run our tests. The actual measuring was done by our version of the measuring program of Pereira et al. that was modified to use Linux powercap for the RAPL measurements and is included in Appendix B. We tested four benchmarks in three programming languages and used all the available implementations in the Computer Language Benchmarks Game, except `C++ fasta #5`[1]. The languages C++, Go and Python were chosen for their popularity and some variation in paradigm (a garbage-collected and an interpreted language). The four benchmarks, binarytrees, fasta, n-body and mandelbrot, were chosen to test a variation in load: both memory-intensive and compute-intensive. The C++ and Go programs were compiled using a recent compiler or the one already present on the system. The implementations in Python were executed using the version of Python 3 already present on the system. The exact versions of the compilers and interpreter used on all systems are listed in Table A.1. In a run of the test suite, each implementation was measured 10 times. For most of the presented results, we ran the suite four times, collecting a total of 40 measurements for each implementation. Of these measurements, we then computed an average and standard deviation.

[1] `C++ fasta #5` repeatedly crashed our test systems, so we excluded it from further tests.

These measurements were performed on three different systems, all containing an Intel CPU with support for RAPL, of which the details are included in Table A.1. These systems were chosen to represent different categories of common system: from System C as an ordinary office computer to System A representing a more serious home computer and System B a high-end desktop. System A and System C were fully under our control, but System B was a shared system, which could potentially be used by other users during our measurements. To prevent this from affecting our measurements significantly, we spread out the four runs of the measurement suite and tried to schedule our measurements at unpopular times. Because we only base conclusions on the relations between measured values on the same system and we see a relatively small standard deviation in the obtained data, the quality of this data is sufficient for our purposes.

3.3 Program output processing

3.3.1 Introduction

After the Out-of-Memory problems on System C (Table A.1), we looked into the cause for the memory shortage and made an interesting discovery: while the entire system ran out of its 8 GiB of RAM, the fasta benchmark on which it crashed, only uses tens of MBs of RAM. The rest of the system memory was actually used by the scheduling script. The scheduling script (`compile_all.py`) used by Pereira et al. [Per+17a] calls for Python to process the entire output, both `stderr` and `stdout`, of the measured programs. While `stderr` is used in case an error occurs in one of the executed programs, `stdout` is not used at all.

The output of a successful run of, for example, the fasta algorithm amounts to more than 250 MB of data. To pipe this data into Python requires extra computations and buffer management. Because we can only measure energy consumption of all the CPU cores together (subsection 2.1.1) and this processing happens during the execution, it is included in the measurements and we suspect it has a considerable effect on the measurements of energy and time of the programs.

3.3.2 Method

To test this hypothesis, we focused on the algorithm with the largest output in our study. Between the algorithms we consider, `fasta` has the largest output for a successful run: 254 MB. The largest output presumably requires the most processing and certainly the most buffer management, and is therefore expected to have the largest effect on energy usage and execution time.

We test the impact by adapting the original scheduling script (Appendix D) to ignore `stdout` and run the tests on only the `fasta`-programs multiple times, alternating between the original and the adapted scheduling script.

In total, four runs of each of the scheduling scripts were performed on System A (Table A.1), amounting to 40 executions of each program with the original scheduling script and as many with the adapted script.

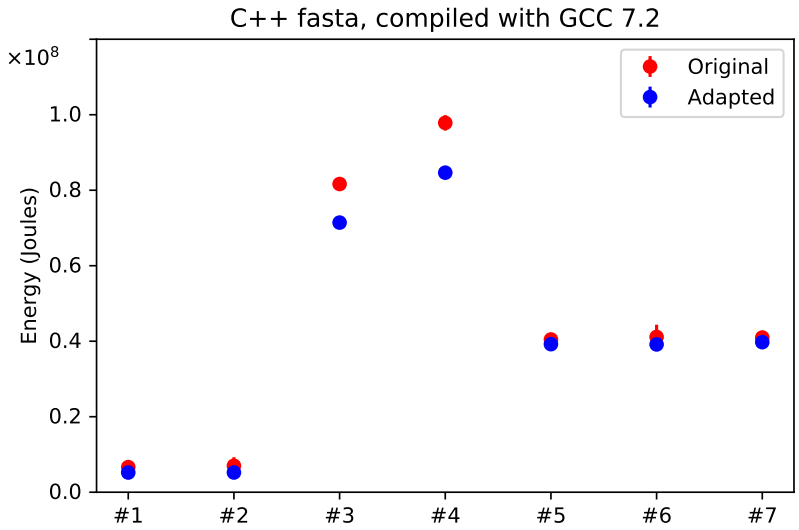


Figure 3.1: The energy usage of the C++ implementations of fasta using the original vs. the adapted scheduling script.

3.3.3 Results

Figure 3.1, Figure 3.2 and Figure 3.3 show the energy usage of the various programs grouped by programming language. Clear is that for all the programs, the measurements with the original scheduling script used more energy than the same program with the adapted script. In most cases, this difference is greater than the sum of the standard deviations of the measurements (figs. 3.1 to 3.3). However, the difference between the two varies wildly. The constant offset that one could intuitively expect for processing identical amounts of output data is certainly not supported by the data. Figure 3.4 shows the percentage increase in energy consumption of the original script over the adapted script. The increase measured is up to 30% and, also considering the errors in the measurement, clearly not constant.

3.3.4 Discussion

Pereira et al. [Per+17a] performed their measurements with a scheduling script that performed unnecessary processing on the `stdout` of the measured programs. According to our comparison, this processing added a significant penalty to the measured energy usage. As can be seen in the results, this penalty is not constant or proportional to the energy usage of the actual program. Therefore, it could affect the ranking of the programs. After all: if the difference in energy consumption of two programs is small, one of the programs could get a larger penalty of the scheduling script than the other, which could change their ranking. Although in our limited set of languages the ranking is not directly affected, we believe a deviation is very likely to appear when the complete set of languages and benchmarks is considered. To prevent this unnecessary processing from influencing our measurements, we will use the modified scheduling script in Appendix D for our other measurements.

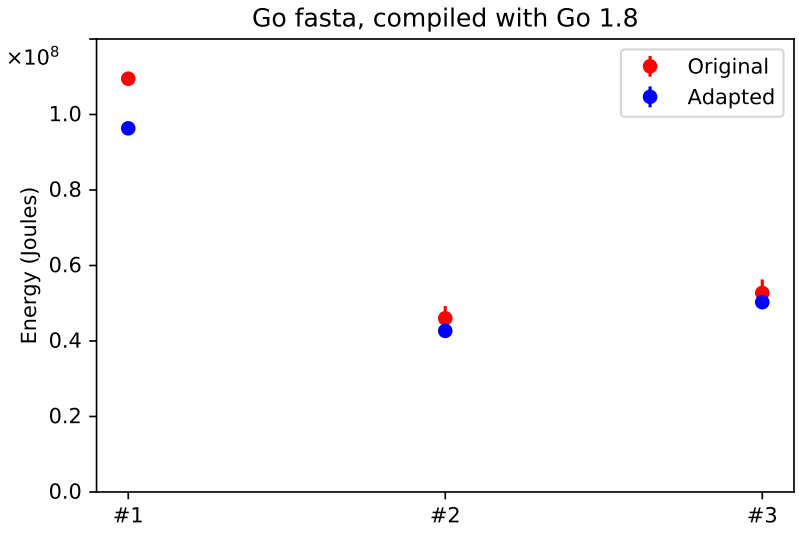


Figure 3.2: The energy usage of the Go implementations of fasta using the original vs. the adapted scheduling script.

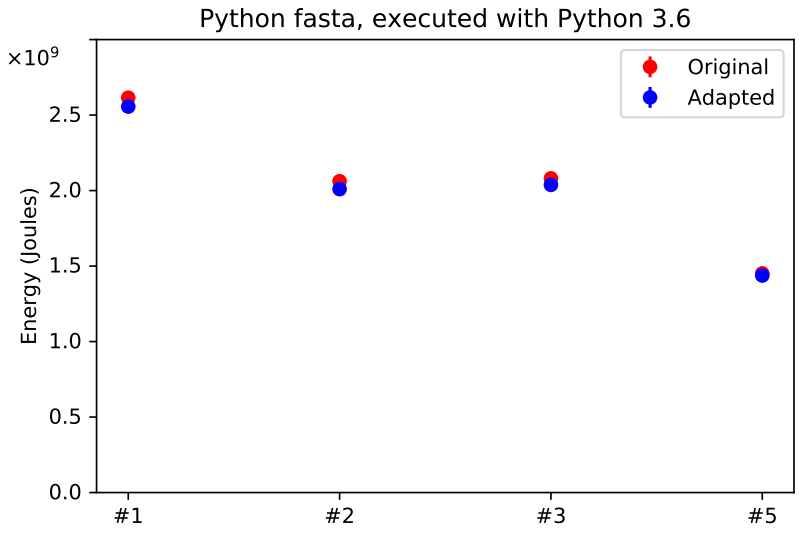
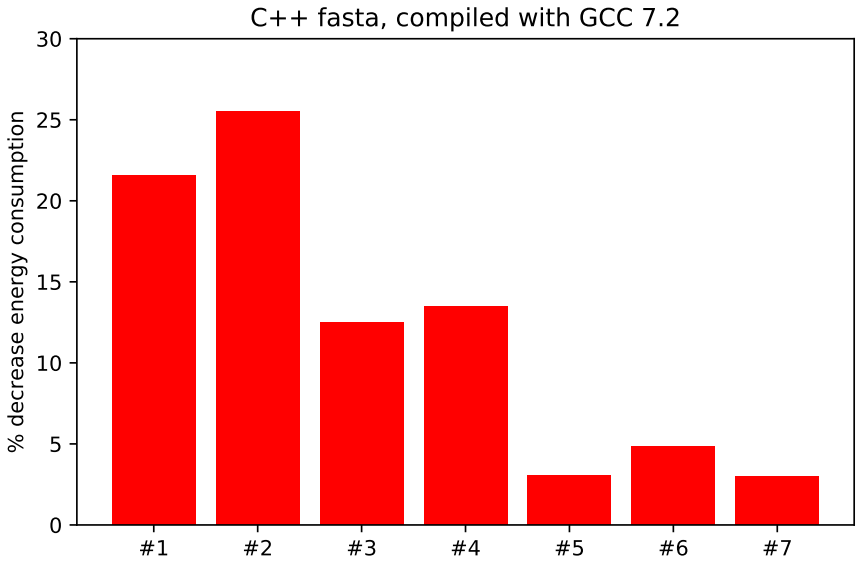
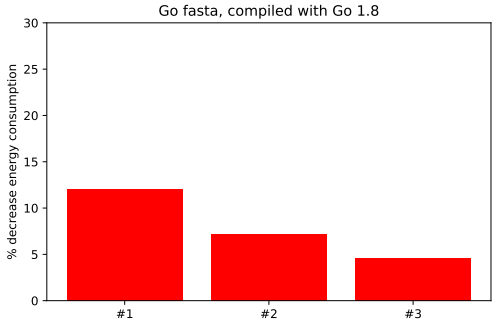


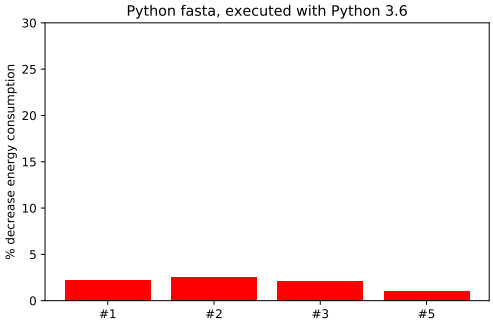
Figure 3.3: The energy usage of the Python implementations of fasta using the original vs. the adapted scheduling script.



(a) The percentage of decrease in measured energy consumption of the C++ implementations when using the adapted scheduling script with respect to the original scheduling script.



(b) The percentage of decrease in measured energy consumption of the Go implementations when using the adapted scheduling script with respect to the original scheduling script.



(c) The percentage of decrease in measured energy consumption of the Python implementations when using the adapted scheduling script with respect to the original scheduling script.

Figure 3.4: The percentage of decrease in measured energy consumption on System A, grouped by programming language.

3.4 Test program ordering

3.4.1 Introduction

On close inspection of the scheduling script after the memory problem, something else came to our attention: the scheduling script used by Pereira et al. [Per+17a] performs all 10 measurements of the same implementation in a row and this could have unintended consequences for the measurements.

That is because modern computers are complex systems. The clock speed of the CPU is not fixed, but can be dynamically adjusted for various reasons, including to optimize for a short, intensive workload, to manage the temperature of the chip or to limit the power usage of the system[Sae04; CSP04; CL95]. These features are very useful in many applications, but could influence our measurements, introducing an extra variable into our measurements. For example, if a certain compute-intensive program A always precedes another, short program B, then the performance of the system during execution of program B could be limited by throttling that program A caused. To determine whether this theoretical issue is really at play, we compare the deterministic order used by Pereira et al.[Per+17a] with a more random scheduling.

3.4.2 Method

We have tested this hypothesis by adapting the scheduling script introduced in section 3.3 (Appendix D) to create a random order for all the tests, ten for each program, using Python's `random.shuffle()` method. Because of this, the predecessor and successor of a program will probably be different at each measurement. We opted to establish the randomized order beforehand to make sure we do test each of the programs an equal number of times. The scheduling script in full is included in Appendix E. Three full runs of both of the scheduling scripts are performed on System A (Table A.1), for a total of 30 measurements for each program using each scheduling script, after which the results can be compared.

3.4.3 Results

If we compare the average energy usage of each program in the original scheduling with the random scheduling, we get the ratios plotted with their standard deviation in Figure 3.5, Figure 3.6 and Figure 3.7 for C++, Go and Python respectively. The values are all reasonably small. As the sample standard deviation of these measurements is typically a few percent, these ratios suggests that the difference in energy usage between the two scheduling strategies is negligible, if there is a difference at all.

3.4.4 Discussion

Pereira et al. [Per+17a] performed their measurements in a fixed order: each program was preceded and succeeded by the same program every time. We compared that strategy to a random order, in which the

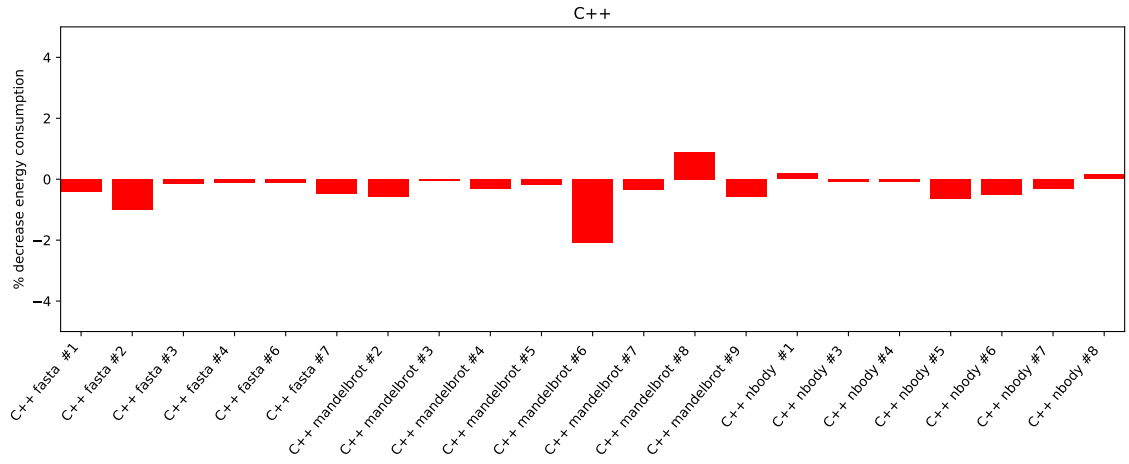


Figure 3.5: The ratio of the difference in energy usage of the C++ implementations between the original scheduling with respect to the random scheduling.

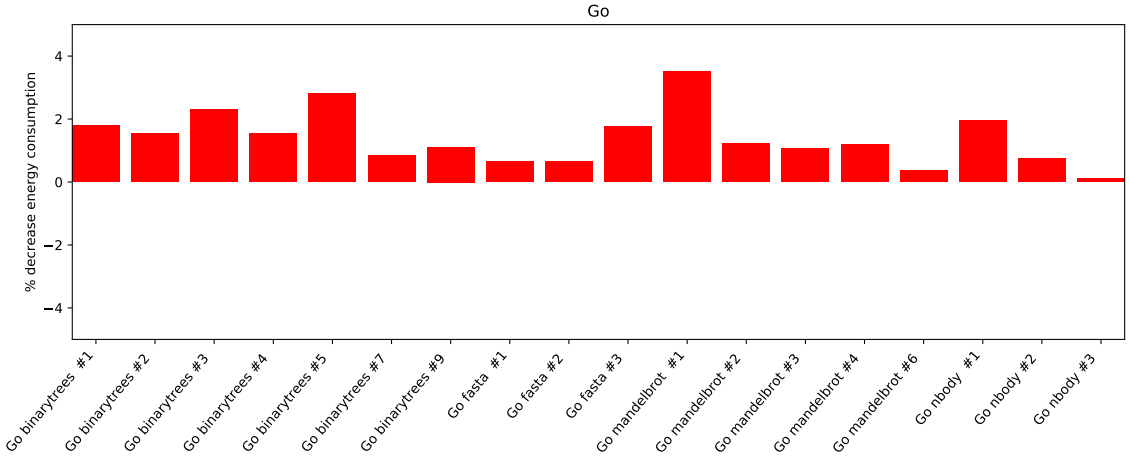


Figure 3.6: The ratio of the difference in energy usage of the Go implementations between the original scheduling with respect to the random scheduling.

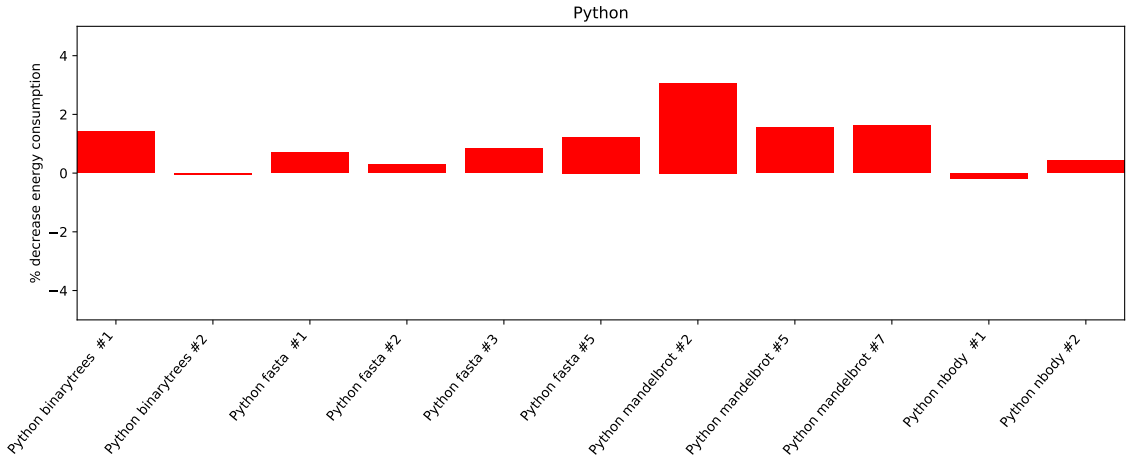


Figure 3.7: The ratio of the difference in energy usage of the Python implementations between the original scheduling with respect to the random scheduling.

preceding and succeeding program would differ between measurements. However, our measurements show no significant difference in measured energy consumption between these two methods. If there are differences, they are smaller than the usable resolution of our measurements, and therefore cannot have an effect on the ranking. The sequential scheduling therefore appears to be usable.

3.5 Conclusion

We have tested two hypotheses with respect to the measuring script. The hypothesis that directly concerned our memory problem has been confirmed. The unnecessary processing of all the output data of the programs does add a measurable and variable energy consumption, thereby potentially affecting the final ranking of the energy efficiency of programming languages. Our other hypothesis, however, that the sequential ordering of the measurements would negatively impact the quality of the measurements, is not supported by our results. This means that we will continue to use our adapted scheduling script that prevents processing the program output, but keeps the sequential ordering, in the rest of our measurements.

Chapter 4

Reproducibility

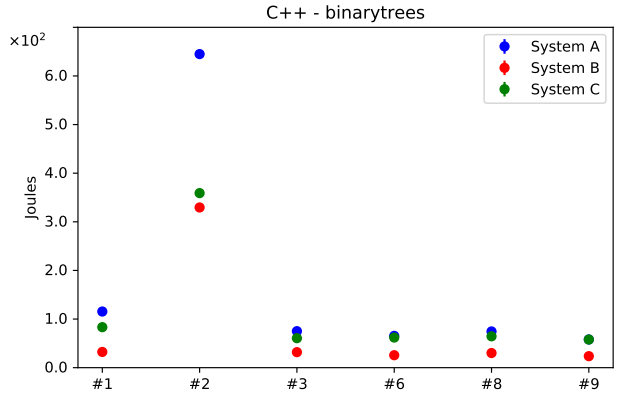
4.1 Introduction

Reproducibility and replicability are considered cornerstones of modern science. Especially in recent years, many scientific communities have been worried about the reproducibility of the research in their field[Err+14; CT14; GFI16] and not all results survive a closer look[Col15; Bak16]. In this chapter, we will investigate whether the results obtained by the methodology that was used for the paper of Pereira et al.[Per+17a] are reproducible on another computer system, or are very specific to the exact testing environment.

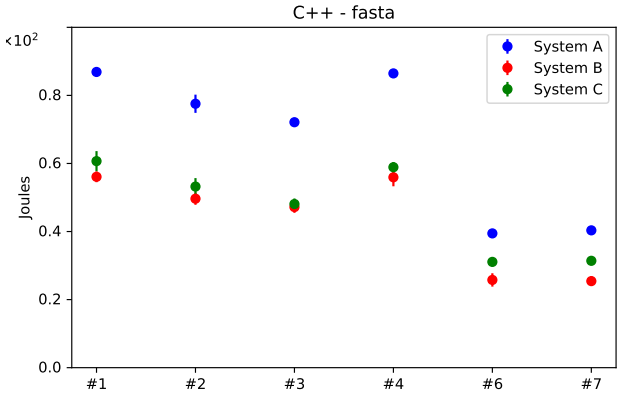
In their paper, Pereira et al. presented various rankings for programming languages including based on energy use. These rankings would only be useful if it is clear for which systems they are valid. In their conclusions, they do not mention reservations about the systems for which their rankings are valid, which suggests they should be valid for most realistic computer systems. We compare the energy efficiency of our benchmark implementations across three computer systems to verify whether the behaviour of the implementations is indeed similar.

4.2 Method

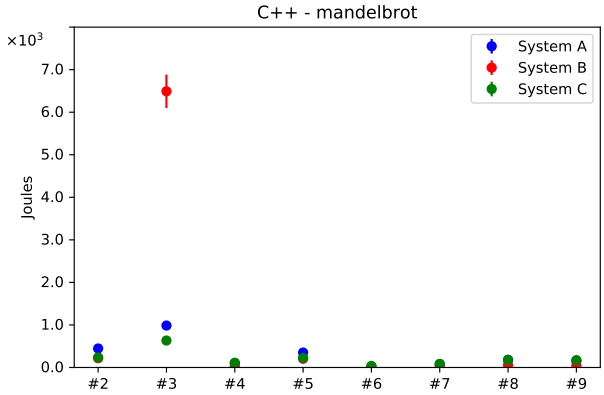
We tested the reproducibility of the results obtained in a certain environment by measuring the energy consumption of all programs on three different systems. The specification of these systems are listed in Table A.1. We performed four runs of the entire test suite on all our systems for a total of 40 measurements of each program on each system. From these measurements, we then calculated the average and standard deviation. Any outliers are compensated by the number of measurements and the inclusion of the standard deviation in our results. Although it is to be expected that the measurements are not exactly the same, we need them to display similar trends to call the results reproducible.



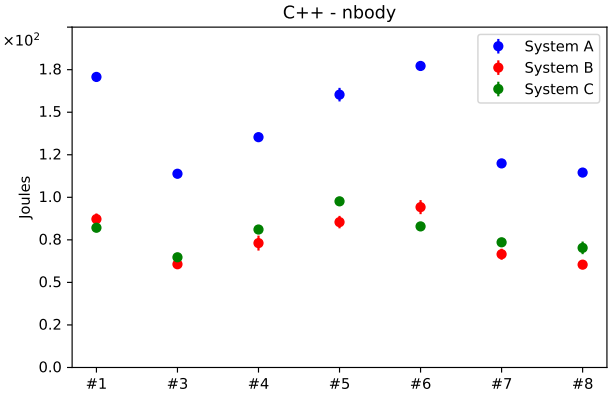
(a) The C++-implementations of the binarytrees benchmark.



(b) The C++-implementations of the fasta benchmark.



(c) The C++-implementations of the mandelbrot benchmark.



(d) The C++-implementations of the nbody benchmark.

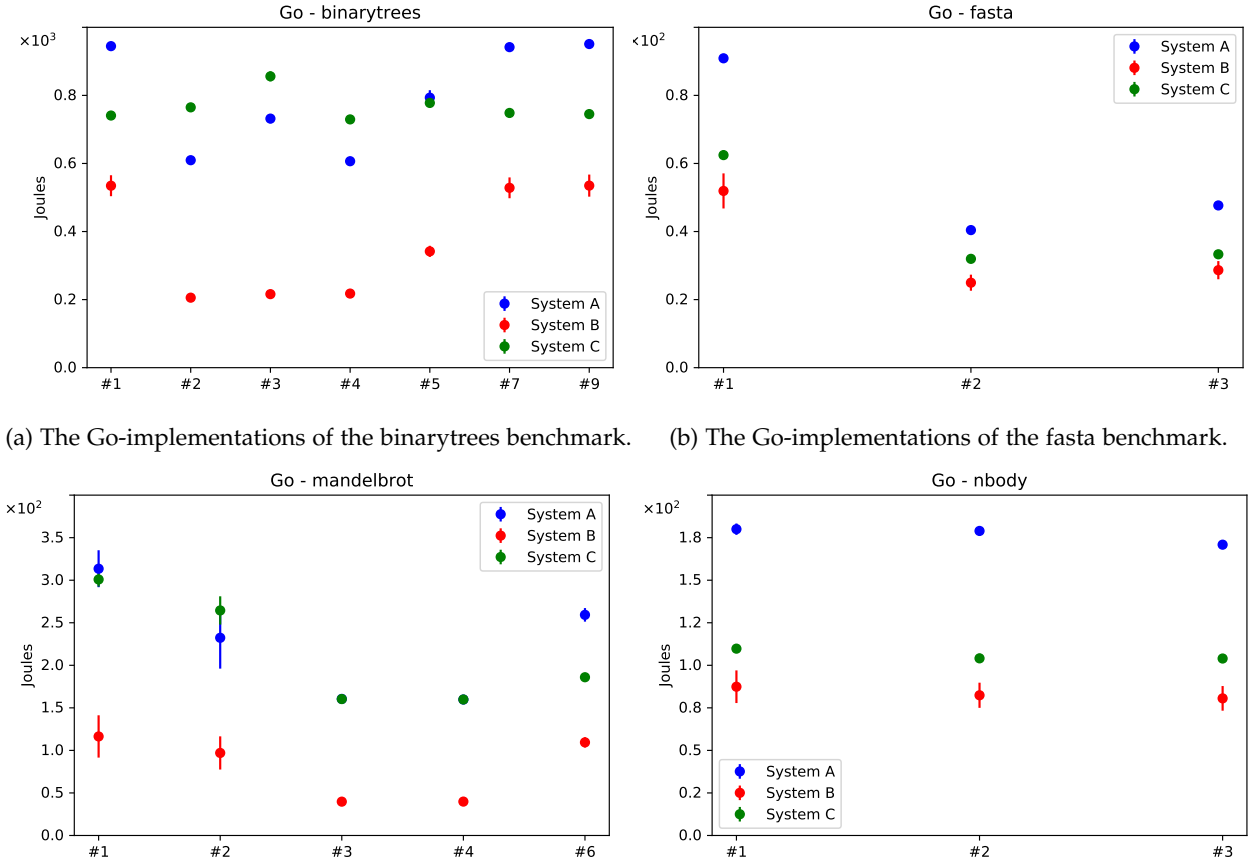
Figure 4.1: The energy use of the C++-implementations on all three systems.

4.3 Results

In the C++-implementations in Figure 4.1, we observe that across most of the benchmarks, the three systems show the same behaviour: the implementations that show a high energy consumption on one system, generally do so on the other systems too and vice versa.

Two exceptions are visible. First, in Figure 4.1d System C shows a lower energy use for benchmark `C++ nbody #6` than both System A and System B and secondly, what really stands out is the measurement of `C++ mandelbrot #3` for System B. While System B generally shows the lowest energy use of all three systems, for `C++ mandelbrot #3` (Figure 4.1c) its measured energy use is almost an entire order of magnitude greater than the other systems and the other mandelbrot benchmarks on System B. Further inspection indicates that an executable built by another compiler shows a similar energy consumption on System B as on System A and C. This establishes that the outlier is caused by a sub-optimal sequence of instructions generated by the GCC compiler for this processor.

In the Go-implementations in Figure 4.2, there are more exceptions to the similarity. In particular the implementations of the binarytrees benchmark (Figure 4.2a) show big differences in behaviour across implementations between System A and System C. Implementation `Go binarytrees #2` would be the most energy-efficient



(a) The Go-implementations of the binarytrees benchmark.

(b) The Go-implementations of the fasta benchmark.

(c) The Go-implementations of the mandelbrot benchmark.

(d) The Go-implementations of the nbody benchmark.

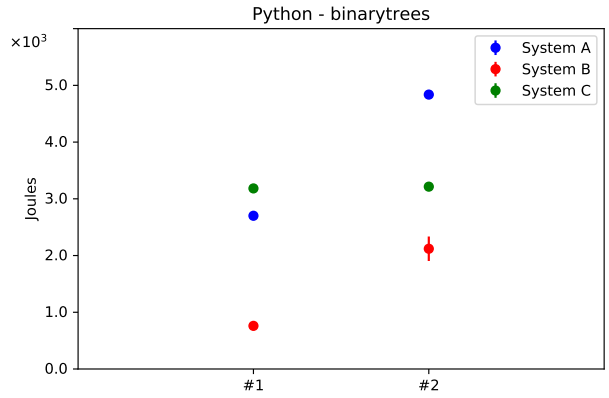
Figure 4.2: The energy use of the Go-implementations on all three systems.

implementation on System A and System B, but on System C, multiple implementations use less energy. For mandelbrot (Figure 4.2c), implementation `Go mandelbrot #6` uses only a little more energy than `Go mandelbrot #3` and `Go mandelbrot #4` on System C, but on System A and System B, its energy use is relatively higher: between that of `Go mandelbrot #1` and `Go mandelbrot #2`.

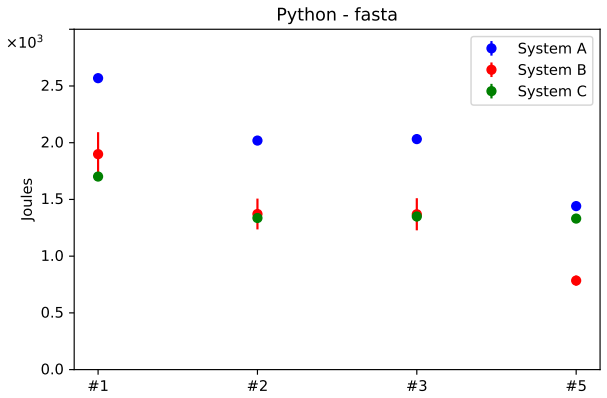
In Python, we have a smaller number of implementations to look at (Figure 4.3). Most of them show the expected similarities between the systems. Only the binarytrees benchmarks stand out a bit (Figure 4.3a). While on System A and System B the `Python binarytrees #2` implementation uses significantly more energy than the `Python binarytrees #1` implementation, on System C, the difference is quite small.

4.4 Discussion

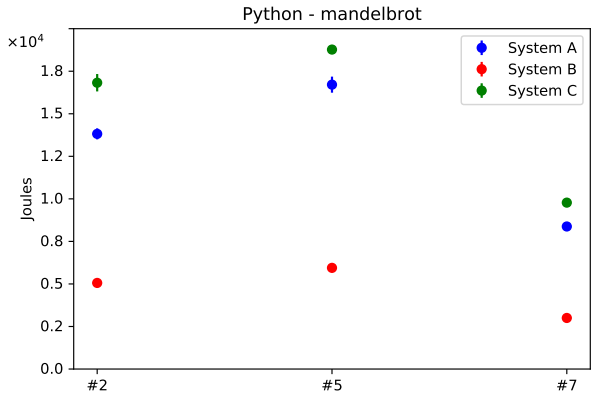
If we want to have a useful ranking of programming languages on metrics including energy efficiency, we have to be sure that this ranking is valid for a clear set of systems, so users can determine whether they can make use of it for their situation. Pereira et al. [Per+17a] present this ranking without such reservations, which poses the question: is their ranking valid in general for systems with a comparable CPU microarchitecture?



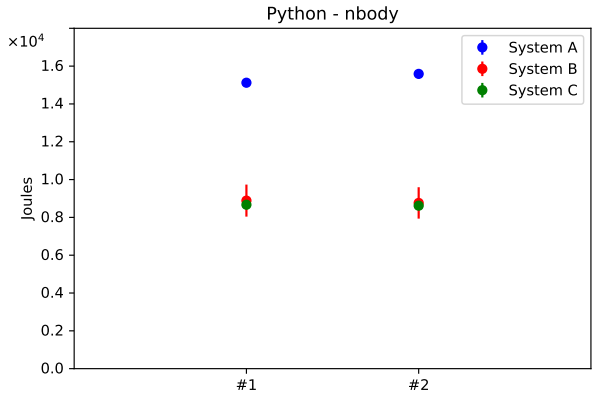
(a) The Python-implementations of the binarytrees benchmark.



(b) The Python implementations of the fasta benchmark.



(c) The Python implementations of the mandelbrot benchmark.



(d) The Python implementations of the nbody benchmark.

Figure 4.3: The energy use of the Python implementations on all three systems.

Our investigation does not result in a conclusive answer to this question, due to its limited sample size in programming languages, benchmarks and systems, but it does confirm the relevance of the question. The assumption that the same implementations will be fastest on different systems holds in many cases, but after testing less than 20 implementations of only four benchmarks in three languages on three systems, we already found a number of exceptions. So much so, that the most energy-efficient implementation can vary between systems, as it does in Figure 4.2a. The paper by Pereira et al. [Per+17a] only considered one implementation for each combination of benchmark and language but as the most energy efficient implementation varies between systems, this creates another possibility for languages to take a different place in the ranking on another system. This does not occur in our limited set of languages, benchmarks and systems, but further investigation measuring more benchmarks in more languages across more systems would help to determine how serious this problem is.

Furthermore, even if a ranking is found that is valid for most systems, we have seen in Figure 4.1c that outliers can occur in specific situations that have an substantial impact on the performance of an application. That being said, the results do indicate a coarse indication can be given as to which implementations are most likely to demonstrate the minimal, or at least a relatively small, energy consumption.

Chapter 5

Time vs Energy

5.1 Introduction

In "Energy Efficiency of Programming Languages", the authors explain very briefly how they chose their test programs: "We (...) gathered the most efficient (i.e. fastest) version of the source code in each of the (...) 10 benchmark problems, for all of the 27 considered programming languages"[Per+17a]. As the paper is about energy efficiency, the easy assumption would be that they are talking about energy when choosing the "most efficient" implementation for each benchmark in each language. In fact, in the paper the authors equate the efficiency with speed, while the conclusion they present to one of their research questions is that a faster language is not always more energy efficient. As this assumption and conclusion appear to contradict each other, we decided to test not just one implementation for each benchmark in each language, but all available. This can show whether the fastest implementation is indeed the most (energy) efficient.

If we find examples of benchmarks where a slower implementation is more energy-efficient, that could help us understand how to achieve this efficiency by comparing the functionally identical programs for the differences that make the energy conservation possible.

5.2 Method

For this analysis, four runs of the test suite were performed on each of the three systems (Table A.1), adding up to a total of 40 measurements for each implementation in each language on each test system. The averages of these measurements are reported.

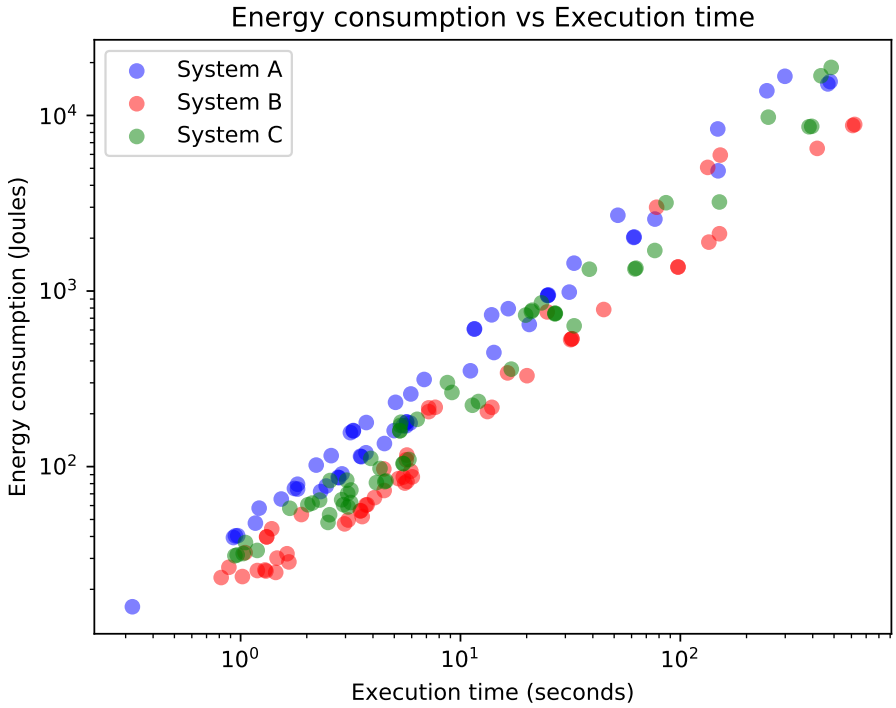


Figure 5.1: The energy consumption plotted against the execution time for all the implementations in our test set. The different test systems are color-coded.

5.3 Results

If we compare all implementations of all benchmarks, we get another interesting view. As is displayed in Figure 5.1, there appears to be a strong, near-linear correlation between the execution time of a given implementation and its energy consumption, regardless of the benchmark or programming language. There is some variation, but most of the energy consumption is determined by the execution time.

Looking closer, Figure 5.2 shows the energy consumption and execution time of the binarytree implementations in Go. Between most of the implementations, we can see that the faster benchmark also consumes less energy, but `Go binarytrees #1` instead consumes less energy than `Go binarytrees #2` while showing a longer execution time. Figure 5.3a shows a similar exception for `C++ nbody #6` and `C++ nbody #5` respectively. Considering the differences between the systems: if we compare the test measurements of the nbody benchmark on System C to System A (Figure 5.3b) and System B (Figure 5.3c), we observe that implementation `C++ nbody #6` consumes relatively less energy on system C, whereas the execution time is not that much lower than on the other systems. Because of this, the ranking of these implementations by energy consumption and by execution time diverge.

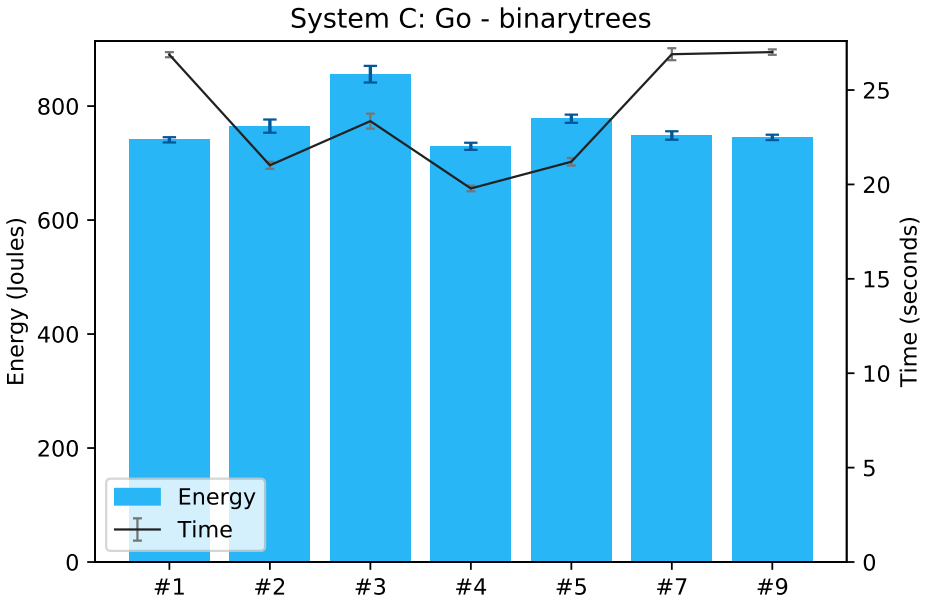


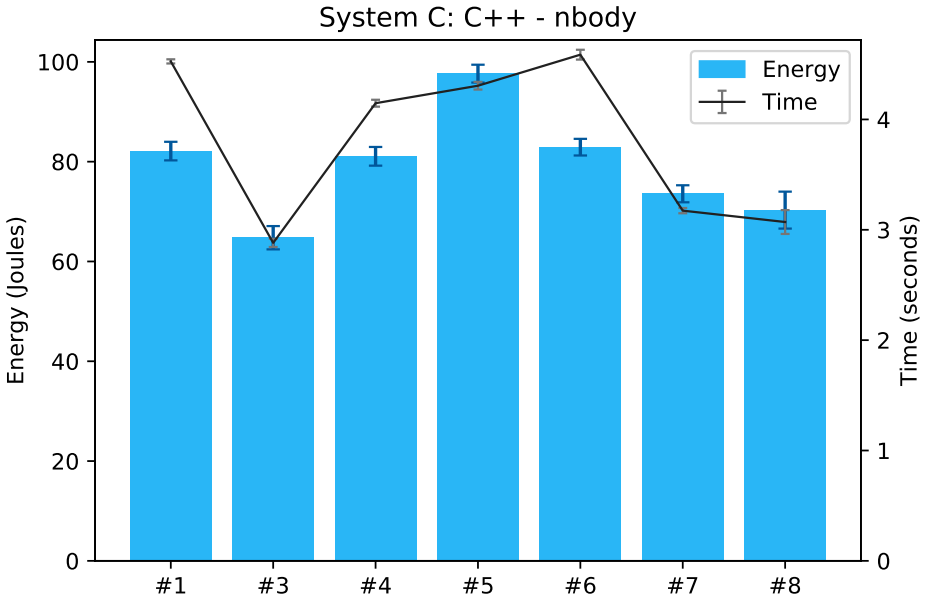
Figure 5.2: The energy consumption and execution time of the available Go implementations of the binarytrees benchmark tested on System C.

5.4 Discussion

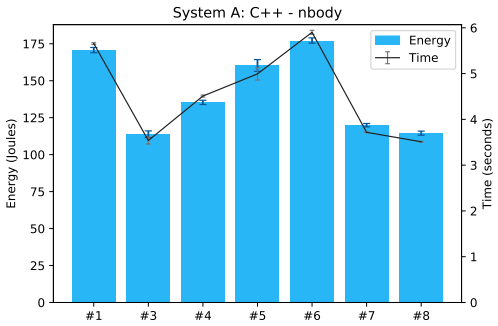
In this experiment, we investigated the relationship between time and energy of different benchmarks. Examples of benchmarks where a slower implementation is still more energy efficient would unambiguously contradict the notion that a faster program is always a more energy efficient one. Furthermore, these could provide insights into the cause of the difference in efficiency, enabling the design of more generally applicable optimizations. However, the results show quite a strong correlation between execution time and energy consumption of a program. Not only within a single benchmark or language, but across the entire set of implementations we tested.

We have encountered two exceptions to the correlation between execution time and energy consumption, where a slower implementation was more energy efficient. These examples demonstrate that there is no fixed relation between execution time and energy consumption of a program and that sometimes a slower program can be more energy efficient. In this small sample set, however, no definitive conclusion can be drawn about the effect on the ranking. Testing a larger sample set is therefore necessary to confirm or reject the validity of the assumption of Pereira et al. [Per+17a].

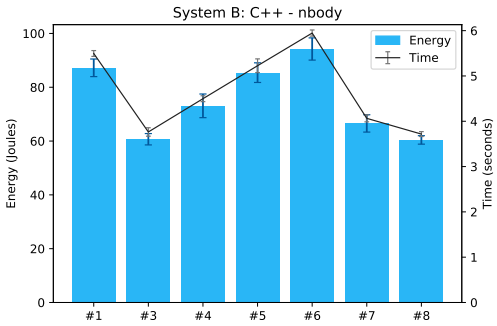
Additionally, these counter-examples were only encountered on System C. While investigating these further could potentially offer valuable information about the energy model of that specific CPU, they are unlikely to offer an easy one-size-fits-all solution for energy saving. This does, however, strengthen the case that the reproducibility of this research across different systems is a real issue and that presenting rankings based on experiments conducted on a single system, as done by the authors of the paper, is premature.



(a) The energy consumption and execution time of the available C++ implementations of the nbody benchmark.



(b) The energy consumption and execution time of the available C++ implementations of the nbody benchmark.



(c) The energy consumption and execution time of the available C++ implementations of the nbody benchmark.

Figure 5.3: The energy consumption and execution time of the available C++ implementations of the nbody benchmark on all three systems.

Chapter 6

Memory

6.1 Introduction

In "Energy Efficiency across Programming Languages", Pereira et al. [Per+17a] also try to answer their research question "How does memory usage relate to energy consumption?". A clear answer to this question could help developers determine how to manage the memory use of their programs if energy consumption is a concern. The authors consider two possible factors that could influence energy consumption by memory: 'continuous memory usage' (that is, the "total amount of memory used over time"[Per+17a]) and 'peak memory usage' (the largest amount of memory in use by the program at any given time during its execution). They investigate the latter, and conclude that, against their expectations, they find no significant relationship between the peak memory usage and the DRAM energy consumption. Finally they do note that the energy consumption might have a stronger relation with how the memory is used, rather than how much of it is used at its maximum.

In this chapter, we will investigate that hypothesis. We suspect the DRAM energy consumption may increase with the number of memory accesses (loads and stores) actually performed on the DRAM, that is, the last level cache misses on a single processor system[DPW16]. To test this hypothesis, we measure the number of load and store instructions performed and the number of misses of the last level cache on the CPU and try to correlate them with the DRAM energy consumption reported by RAPL.

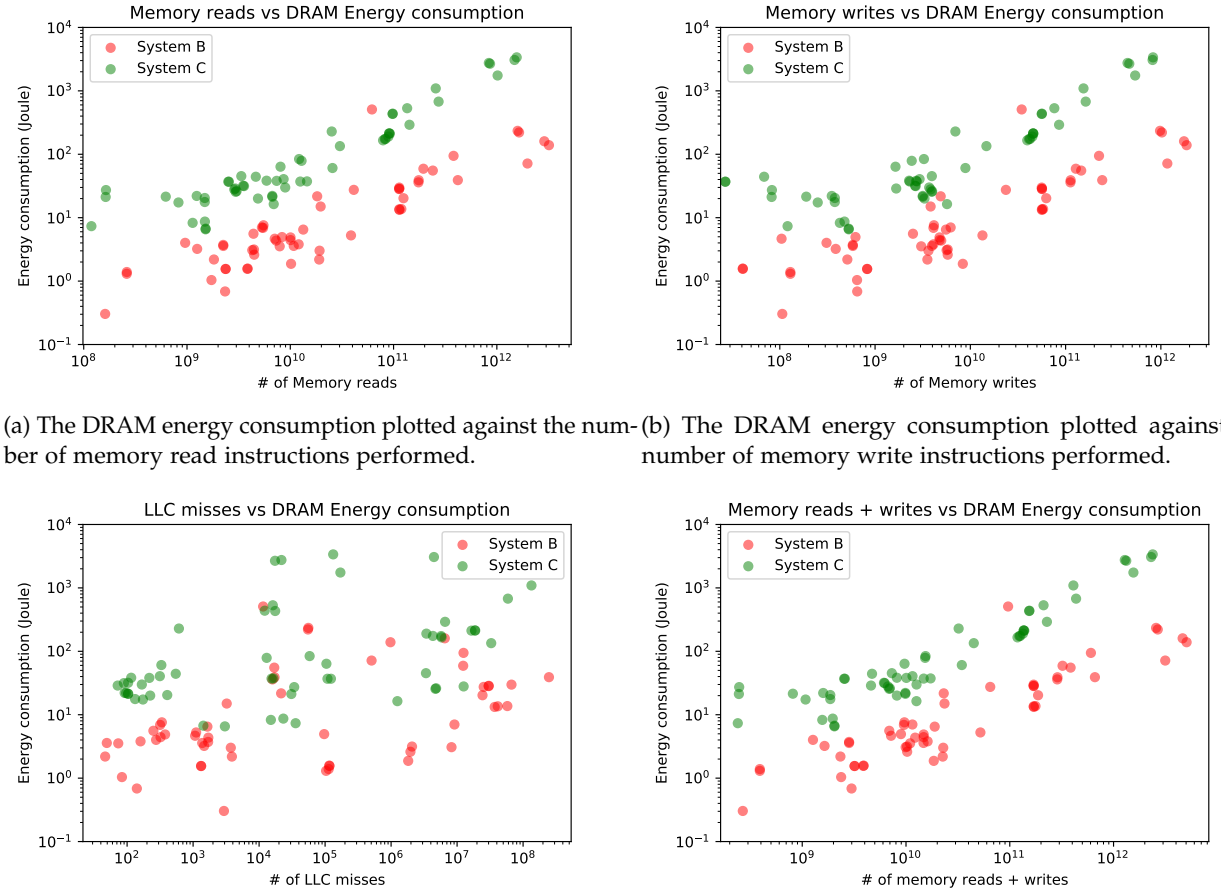
6.2 Method

For this experiment, we need the number of load and store instructions performed and the number of misses of the last level cache. We determine these using the Performance Monitoring Events provided by the Intel processors using the *perf_event* API in Linux (see Section 2.2). For the memory measurements, we run the entire test suite twice on System B and System C for a total of 20 measurements for each implementation on each of the two systems. We do not use System A for this experiment, because the CPU in System A does not

support the RAPL DRAM domain (see subsection 2.1.1) and we therefore cannot measure the DRAM energy consumption on this system.

To determine the energy consumption, we run the entire test suite four times, for a total of 40 measurements for each implementation on each system. The averages are computed as in the other experiments.

6.3 Results



(a) The DRAM energy consumption plotted against the number of memory read instructions performed.

(b) The DRAM energy consumption plotted against the number of memory write instructions performed.

(c) The DRAM energy consumption plotted against the number of last level cache misses encountered.

(d) The DRAM energy consumption plotted against the number of memory read and write instructions performed.

Figure 6.1: The DRAM energy consumption plotted against the measured memory metrics for System B and System C.

In Figure 6.1, the correlation between the measured memory metrics and the energy consumption of the programs is shown. The number of memory read and write instructions, both separate (figs. 6.1a and 6.1b) and combined (fig. 6.1d), show a clear correlation to the DRAM energy consumption. One notable exception is `C++ mandelbrot #3`, visible as an outlier on System B, using a remarkable amount of energy (see Section 4.3). The number of last level cache misses (Figure 6.1c) however, seems to have barely any correlation with the energy consumption, contrary to our expectations.

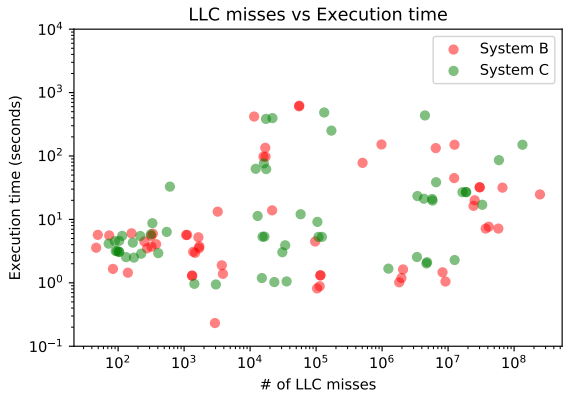
To verify the relevance of the correlation between the number of memory read and write instructions and the energy consumption of the program, we also look at the correlation between these memory metrics and the

execution time. We already know that the execution time is a large factor in the energy consumption of a program and we suspect that the number of read and write instructions has a strong relation to the execution time. If that is the case, then that can explain the correlation between the number of memory instructions and the DRAM energy consumption. It would then provide no meaningful information about the energy cost of memory instructions.

In Figure 6.2 we can see that this is, in fact, the case. Comparing the graphs Figure 6.2a, Figure 6.2b, Figure 6.2c and Figure 6.2d with their counterparts in Figure 6.1, it is directly clear that the relation between memory instructions and time is very similar to the correlation between memory instructions and energy consumption. Here we also see that the outlier on System B in Figure 6.1 not only uses a remarkable amount of energy, but also of time, keeping the similarity intact.

(a) The execution time plotted against the number of memory read instructions performed.

(b) The execution time plotted against the number of memory write instructions performed.



(c) The execution time plotted against the number of last level cache misses encountered.

(d) The execution time plotted against the number of memory read and write instructions performed.

Figure 6.2: The execution time plotted against the measured memory metrics for System B and System C.

6.4 Discussion

We investigated the relation between the number of load and store instructions performed or the number of misses of the last level cache and the DRAM energy consumption of programs. Expecting to find a relation

between the number of misses of the last level cache and the DRAM energy consumption, we were surprised to find a lack of correlation between the two. A possible explanation is that the timing of the cache misses forms a larger factor than the number of misses because of the power states of the DRAM controller. If the cache misses are all encountered in a short period of time, followed by a longer period without cache misses, the DRAM controller can switch to a lower power state, thereby reducing its energy consumption. However, if the cache misses are distributed more evenly over time, an active power state is required for a larger amount of time, leading to a greater energy consumption[FEL01]. Further research could investigate the interaction of these power states with high-level programs.

The number of memory load and store instructions do show a clear correlation with the DRAM energy consumption, but a similar relation with the execution time. Because we know that the execution time is a large factor in the energy consumption, a relation between the number of memory instructions and the execution time can very well explain the correlation with energy consumption. We therefore cannot conclusively demonstrate a relation between the number of memory instructions and the DRAM energy consumption with this collection of programs. Neither can we, however, rule out such a relation, like Pereira et al. [Per+17a] have ruled out any significant relationship between DRAM energy consumption and peak memory usage of a program. A future investigation using (micro)benchmarks that show a great difference in memory accesses and cache misses but a similar execution time and vice versa could provide a definitive conclusion about this potential relation.

Chapter 7

Conclusions

In this thesis, we have had a close look at the paper "Energy Efficiency across Programming Languages"[Per+17a]. Firstly, we scrutinized the measurement methodology. We found a measurable impact of the scheduling script on the measurements that could influence the rankings and proposed a modification to the test setup to mitigate this impact and increases the reliability of the measurements. An investigation into the scheduling of the measurements was also performed, but we did not find any influence of the order of the programs on the measurements.

Using the adapted test setup, we could test the reproducibility of the measurements. In experiments across three different systems, we found that the ranking that is presented in Pereira et al. [Per+17a] is not perfect and that variations could occur between different systems. On top of that, specific implementations can see substantial deviations on certain systems, rendering the use of a ranking in general uncertain.

We then revisited two of the research questions in the paper by Pereira et al. [Per+17a] First, in investigating the question whether a faster language always is a more energy efficient one, the authors made the assumption that a faster implementation of a benchmark within a given language is always more energy efficient than an implementation that has a higher execution time. Because Pereira et al. based their selection of sample programs on this assumption, we tested it and found two counter-examples. However, only one of the three systems we tested on demonstrated these counter-examples, so a larger collection of benchmarks, languages and test systems should be examined to provide more insight into the extent to which a ranking based on these measurements is actually applicable.

Secondly, we used a different strategy in an attempt to answer the question how memory and energy consumption relate. Comparing the memory instructions and activity with the DRAM energy consumption of our sample programs, we find a remarkable lack of correlation between the number of last level cache misses and the DRAM energy consumption, possibly because of the power saving strategy of the DRAM controller. We do find a correlation between the number of memory instructions and the energy consumption, but because that correlation can be explained by the relation of both with the execution time, we cannot conclusively show a direct relation between the number of memory instructions and the DRAM energy consumption.

Overall, our investigations suggest that while Pereira et al. [Per+17a] have found very interesting research questions, their conclusions are insufficiently substantiated. The authors claim to have shown which programming languages are the most energy efficient across 10 different benchmark problems, how execution time and energy consumption of programs relate and they present a collection of rankings of programming languages measured by the constraints energy, time, and peak memory usage. However, in our measurements of more implementations of the same benchmarks and across multiple systems, their assumptions and conclusions do not hold up. Further research across more sample programs and systems is required to provide definitive answers to the research questions.

7.1 Future work

Pereira et al. [Per+17a] have constructed a solid starting point, but there are plenty of opportunities to build on their work. Many of the experiments in this thesis could provide much more comprehensive results if the collection of sample programs would be larger. Performing these experiments on the entire collection of programs in the Computer Language Benchmarks Game could yield more examples of exceptions to the assumption that faster programs are always more energy efficient. Investigating these examples would reveal why these slower programs demonstrate a lower energy consumption and provide techniques for general energy optimization.

Another issue of scale is that of different test systems. By testing on three systems, we already saw differences in results, but a wider variety of systems could undoubtedly provide valuable insight into the interaction between the high-level code and the system properties. Other factors that have not yet been integrated into these tests are the operating system, and the compiler. Comparisons across different operating systems or different compilers would show the impact these have on the (power) performance of programs.

Concerning the reliability of the measurements: in scrutinizing the test setup of Pereira et al. [Per+17a], we found that the scheduling script had a significant impact. While we have presented modifications to the test setup to mitigate that impact, background processes and measurement software still pollute the measurements. A possible solution would be to use a dual-CPU system in which a single CPU is reserved for just the measured program and all other processing can be handled by the second CPU. Because Intel RAPL allows us to measure the energy consumption of a single CPU, this would make it possible to determine with great precision the energy consumption by just the measured program.

Finally, because of the use of Intel RAPL, all of the tested systems are based on an Intel CPU. Future research could reveal whether these results are comparable to measurements on systems based on other CPUs and if it is at all possible to similarly optimize for power across all common system configurations, or that different optimization strategies are required for other system configurations.

Bibliography

[16] *Intel64 and IA-32 Architectures Software Developer's Manual*. Vol. 3B.2: *System Programming Guide, Part 2*. Sept. 2016. URL: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf.

[18] *Instructions Retired Event*. Intel Corporation. May 28, 2018. URL: https://software.intel.com/en-us/vtune-amplifier-help-instructions-retired-event (visited on 08/27/2018).

[Bak16] Monya Baker. "Is there a reproducibility crisis? A Nature survey lifts the lid on how researchers view the'crisis rocking science and what they think will help". In: *Nature* 533.7604 (2016), pp. 452–455.

[CL95] Peng-Cheng Chen and Terng-Huei Lai. *Temperature control for a variable frequency CPU*. US Patent 5,422,806. June 6, 1995.

[Cla+02] Lawrence T Clark et al. *Method and apparatus for dynamic power control of a low power processor*. US Patent 6,425,086. July 2002.

[Col15] Open Science Collaboration. "Estimating the reproducibility of psychological science". In: *Science* 349.6251 (2015). ISSN: 0036-8075. DOI: 10.1126/science.aac4716. eprint: http://science.sciencemag.org/content/349/6251/aac4716.full.pdf. URL: http://science.sciencemag.org/content/349/6251/aac4716.

[Cou+15] M. Couto et al. "GreenDroid: A tool for analysing power consumption in the android ecosystem". In: *2015 IEEE 13th International Scientific Conference on Informatics*. Nov. 2015, pp. 73–78. DOI: 10.1109/Informatics.2015.7377811.

[CSP04] Kihwan Choi, Ramakrishna Soma, and Massoud Pedram. "Dynamic Voltage and Frequency Scaling Based on Workload Decomposition". In: *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*. ISLPED '04. Newport Beach, California, USA: ACM, 2004, pp. 174–179. ISBN: 1-58113-929-2. DOI: 10.1145/1013235.1013282. URL: http://doi.acm.org/10.1145/1013235.1013282.

[CT14] Francis S Collins and Lawrence A Tabak. "NIH plans to enhance reproducibility". In: *Nature* 505.7485 (2014), pp. 612–613.

[DPW16] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. "A Validation of DRAM RAPL Power Measurements". In: *Proceedings of the Second International Symposium on Memory Systems*. MEMSYS

'16. Alexandria, VA, USA: ACM, 2016, pp. 455–470. ISBN: 978-1-4503-4305-3. DOI: 10.1145/2989081.2989088. URL: http://doi.acm.org/10.1145/2989081.2989088.

[Err+14] Timothy M Errington et al. "Science forum: An open investigation of the reproducibility of cancer biology research". In: *Elife* 3 (2014), e04333.

[FEL01] Xiaobo Fan, Carla Ellis, and Alvin Lebeck. "Memory Controller Policies for DRAM Power Management". In: *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*. ISLPED '01. Huntington Beach, California, USA: ACM, 2001, pp. 129–134. ISBN: 1-58113-371-5. DOI: 10.1145/383082.383118. URL: http://doi.acm.org/10.1145/383082.383118.

[GFI16] Steven N. Goodman, Daniele Fanelli, and John P. A. Ioannidis. "What does research reproducibility mean?" In: *Science Translational Medicine* 8.341 (2016), 341ps12–341ps12. ISSN: 1946-6234. DOI: 10.1126/scitranslmed.aaf5027. eprint: http://stm.sciencemag.org/content/8/341/341ps12.full.pdf. URL: http://stm.sciencemag.org/content/8/341/341ps12.

[Gou] Isaac Gouy. *The computer language benchmarks game*. URL: URL%20https://benchmarksgame-team.pages.debian.net/benchmarksgame (visited on 03/11/2018).

[Häh+12] Marcus Hähnel et al. "Measuring Energy Consumption for Short Code Paths Using RAPL". In: *SIGMETRICS Perform. Eval. Rev.* 40.3 (Jan. 2012), pp. 13–17. ISSN: 0163-5999. DOI: 10.1145/2425248.2425252. URL: http://doi.acm.org/10.1145/2425248.2425252.

[PC17] Gustavo Pinto and Fernando Castor. "Energy Efficiency: A New Concern for Application Software Developers". In: *Commun. ACM* 60.12 (Nov. 2017), pp. 68–75. ISSN: 0001-0782. DOI: 10.1145/3154384. URL: http://doi.acm.org/10.1145/3154384.

[Per+16] Rui Pereira et al. "The Influence of the Java Collection Framework on Overall Energy Consumption". In: *Proceedings of the 5th International Workshop on Green and Sustainable Software*. GREENS '16. Austin, Texas: ACM, 2016, pp. 15–21. ISBN: 978-1-4503-4161-5. DOI: 10.1145/2896967.2896968. URL: http://doi.acm.org/10.1145/2896967.2896968.

[Per+17a] Rui Pereira et al. "Energy Efficiency across Programming Languages: How Do Energy, Time, and Memory Relate?" In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2017. Vancouver, BC, Canada: ACM, 2017, pp. 256–267. ISBN: 978-1-4503-5525-4. DOI: 10.1145/3136014.3136031. URL: http://doi.acm.org/10.1145/3136014.3136031.

[Per+17b] Rui Pereira et al. "Helping Programmers Improve the Energy Efficiency of Source Code". In: *Proceedings of the 39th International Conference on Software Engineering Companion*. ICSE-C '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 238–240. ISBN: 978-1-5386-1589-8. DOI: 10.1109/ICSE-C.2017.80. URL: https://doi.org/10.1109/ICSE-C.2017.80.

[Per+18a] Rui Pereira et al. *Energy-Languages/compile_all.py*. Aug. 18, 2018. URL: https://github.com/greensoftwarelab/Energy-Languages/blob/321358b642174c2a77a8966d21f7ff4a83fdabd4/compile_all.py.

[Per+18b] Rui Pereira et al. *Energy-Languages/Energy Measurement Framework*. Aug. 18, 2018. URL: https://github.com/greensoftwarelab/Energy-Languages/tree/321358b642174c2a77a8966d21f7ff4a83fdabd4/RAPL.

[PLS01] Johan Pouwelse, Koen Langendoen, and Henk Sips. "Dynamic Voltage Scaling on a Low-power Microprocessor". In: *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*. MobiCom '01. Rome, Italy: ACM, 2001, pp. 251–259. ISBN: 1-58113-422-3. DOI: 10.1145/381677.381701. URL: http://doi.acm.org/10.1145/381677.381701.

[Sae04] Ali Saeed. *Modulating CPU frequency and voltage in a multi-core CPU architecture*. US Patent 6,711,447. Mar. 23, 2004.

[Sah+12] C. Sahin et al. "Initial explorations on design pattern energy usage". In: *2012 First International Workshop on Green and Sustainable Software (GREENS)*. June 2012, pp. 55–61. DOI: 10.1109/GREENS.2012.6224257.

[Saw18] Michael Sawh. *Best smartwatch guide: The top smartwatches to buy in 2018*. Wareable, July 4, 2018. URL: https://www.wareable.com/smartwatches/what-is-the-best-smartwatch-2018 (visited on 08/07/2018).

[SW15] Stephan Schmidt and Hannes Weigt. "Interdisciplinary energy research and energy consumption: What, why, and how?" In: *Energy Research & Social Science* 10 (2015), pp. 206–219. ISSN: 2214-6296. DOI: https://doi.org/10.1016/j.erss.2015.08.001. URL: http://www.sciencedirect.com/science/article/pii/S2214629615300244.

[Vit94] Eric A Vittoz. *Micropower techniques*. 1994.

[Yad18] Manjit S. Yadav. "Making emerging phenomena a research priority". In: *Journal of the Academy of Marketing Science* 46.3 (May 2018), pp. 361–365. ISSN: 1552-7824. DOI: 10.1007/s11747-017-0575-0. URL: https://doi.org/10.1007/s11747-017-0575-0.

Appendix A

System details

	System A	System B	System C
CPU	Intel Core i7-3770 @ 3.40 GHz	Intel Core i7-6950X @ 3.00 GHz	Intel Core i3-4350 @ 3.60 GHz
Cores	4	10	2
RAM	8 GiB	64 GiB	8 GiB
Linux	4.13.0-32-generic	4.4.0-109-generic	4.13.0-46-generic
GCC	7.2.0 (7.2.0-8ubuntu3)	5.4.0 (5.4.0-6ubuntu1~16.04.9)	7.2.0 (7.2.0-8ubuntu3.2)
Python	3.6.3	3.5.2	3.6.3
Go	1.8	1.8	1.10.2

Table A.1: The details of the systems we used for testing.

Appendix B

RAPL measurements

`main.c` is an adaption of the original measuring program[Per+18b] by Pereira et al. [Per+17a] to use the powercap interface for RAPL measurements instead of reading out the MSRs directly.

```
1  #include <stdio.h>
2  #include <time.h>
3  #include <math.h>
4  #include <string.h>
5  #include <inttypes.h>
6  #include <sys/time.h>
7
8  #include <powercap/powercap-rapl.h>
9
10 #include <sys/syscall.h>
11 #include <stdlib.h>
12
13 #define RUNTIME 1
14 #define NUM_ZONES 4
15 powercap_rapl_zone rapl_zones[] = {
16     POWERCAP_RAPL_ZONE_PACKAGE,
17     POWERCAP_RAPL_ZONE_CORE,
18     POWERCAP_RAPL_ZONE_UNCORE,
19     POWERCAP_RAPL_ZONE_DRAM
20 };
21
22 uint64_t calculate_energy_used(uint64_t energy_start[], uint64_t energy_end[], int
      zone_supported[],
23                                uint64_t max_energy[], int zone) {
24   if (zone_supported[zone] != 1)
25     return 0;
26   // Correct for overflow of max energy range (at most once)
27   if (energy_start[zone] > energy_end[zone]) {
28     energy_end[zone] += max_energy[zone];
29   }
```

```
30    return energy_end[zone] - energy_start[zone];
31  }
32
33
34  int main (int argc, char **argv)
35  { char command[500]="",language[500]="", test[500]="", path[500]="";
36    int  ntimes = 10;
37    int  i=0;
38
39  #ifdef RUNTIME
40    //clock_t begin, end;
41    double time_spent;
42    struct timeval tvb,tva;
43  #endif
44
45    uint32_t num_rapl_packages = powercap_rapl_get_num_packages();
46    printf("RAPL domains: %i\n", num_rapl_packages);
47
48    // Initialize first RAPL domain
49    powercap_rapl_pkg pkg = { 0 };
50    powercap_rapl_init(0, &pkg, 1);
51    int zone_enabled[NUM_ZONES];
52    int zone_supported[NUM_ZONES];
53    uint64_t max_energy_range[NUM_ZONES];
54    uint64_t zone_energy_start[NUM_ZONES];
55    uint64_t zone_energy_end[NUM_ZONES];
56
57    system(command);
58
59    FILE * fp;
60
61    //Run command
62  //  strcpy(command, "./" );
63    strcat(command,argv[1]);
64    //Language name
65    strcpy(path,"../");
66
67
68    strcpy(language,argv[2]);
69    strcat(language,".csv");
70    strcat(path,language);
71    //Test name
72    strcpy(test,argv[3]);
73
74    if (argc > 4) {
75      ntimes = atoi(argv[4]);
76    }
77
78    //ntimes = atoi (argv[2]);
```

```
79
80
81    fp = fopen(path,"a");
82
83    //fprintf(fp," , Package , CPU , GPU , DRAM? , Time (sec) \n");
84
85    for (i = 0 ; i < ntimes ; i++)
86      {
87
88          fprintf(fp,"%s , ",test);
89
90
91                  #ifdef RUNTIME
92                          //begin = clock();
93                                  gettimeofday(&tvb,0);
94                  #endif
95
96      for (int i = 0; i < NUM_ZONES; i++) {
97        zone_enabled[i] = powercap_rapl_is_enabled(&pkg, rapl_zones[i]);
98        zone_supported[i] = powercap_rapl_is_zone_supported(&pkg, rapl_zones[i]);
99        powercap_rapl_get_max_energy_range_uj(&pkg, rapl_zones[i], &(max_energy_range[i]));
100 #ifdef DEBUG
101       char name[20];
102       ssize_t len = powercap_rapl_get_name(&pkg, rapl_zones[i], &name, 20);
103       printf("Domain %s |", name);
104       int supported = powercap_rapl_is_zone_supported(&pkg, rapl_zones[i]);
105       printf(" %s |", supported ? "supported" : "unsupported");
106       if (zone_enabled[i] == 0 && supported) {
107         powercap_rapl_set_enabled(&pkg, rapl_zones[i], 1);
108       }
109       printf(" %s |", zone_enabled[i] ? "enabled" : "disabled");
110       uint64_t max_range;
111       powercap_rapl_get_max_energy_range_uj(&pkg, rapl_zones[i], &max_range);
112       printf(" max (uJ): %" PRIu64 " \n", max_range);
113 #endif
114       if (zone_enabled[i] == 0 && zone_supported[i] == 1) {
115         powercap_rapl_set_enabled(&pkg, rapl_zones[i], 1);
116       }
117       if (zone_supported[i])
118         powercap_rapl_get_energy_uj(&pkg, rapl_zones[i], &(zone_energy_start[i]));
119 #ifdef DEBUG
120       char name[20];
121       powercap_rapl_get_name(&pkg, rapl_zones[i], &name, 20);
122       printf("%s | now: %" PRIu64 " | max: %" PRIu64 "\n", name, zone_energy_start[i],
              max_energy_range[i]);
123 #endif
124     }
125
126         system(command);
```

```
127
128     for (int i = 0; i < NUM_ZONES; i++) {
129       if (zone_supported[i])
130         powercap_rapl_get_energy_uj(&pkg, rapl_zones[i], &(zone_energy_end[i]));
131       // Enable/disable zones to restore initial state
132       if (zone_enabled[i] == 0) {
133         powercap_rapl_set_enabled(&pkg, rapl_zones[i], zone_enabled[i]);
134       }
135     }
136
137     fprintf(fp, "%" PRIu64 ", %" PRIu64 ", %" PRIu64 ", %" PRIu64 ", ",
138             calculate_energy_used(zone_energy_start, zone_energy_end, zone_supported,
                    max_energy_range, 0),
139             calculate_energy_used(zone_energy_start, zone_energy_end, zone_supported,
                    max_energy_range, 1),
140             calculate_energy_used(zone_energy_start, zone_energy_end, zone_supported,
                    max_energy_range, 2),
141             calculate_energy_used(zone_energy_start, zone_energy_end, zone_supported,
                    max_energy_range, 3)
142     );
143
144                 #ifdef RUNTIME
145                         //end = clock();
146                         //time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
147                         gettimeofday(&tva,0);
148                         time_spent = (tva.tv_sec-tvb.tv_sec)*1000000 + tva.tv_usec-tvb.tv_usec
                                ;
149                         time_spent = time_spent / 1000;
150                 #endif
151
152
153                 #ifdef RUNTIME
154                         fprintf(fp," %G \n",time_spent);
155                 #endif
156     }
157
158
159   fclose(fp);
160   fflush(stdout);
161
162   return 0;
163 }
```

Appendix C

Memory measurements

main.c is an adaption of the original measuring program[Per+18b] by Pereira et al. [Per+17a] to use the perf_events interface for measurements of memory-related Performance Monitoring Events.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <inttypes.h>
4  #include <sys/time.h>
5
6  #include <powercap/powercap-rapl.h>
7  #include <papi.h>
8  #include <linux/perf_event.h>
9
10 #include <stdlib.h>
11 #include <syscall.h>
12 #include <sys/ioctl.h>
13
14 #define RUNTIME 1
15
16 // Source: http://man7.org/linux/man-pages/man2/perf_event_open.2.html
17 static int
18 perf_event_open(struct perf_event_attr *hw_event, pid_t pid,
19                 int cpu, int group_fd, unsigned long flags) {
20   int ret;
21
22   ret = (int) syscall(__NR_perf_event_open, hw_event, pid, cpu,
23                       group_fd, flags);
24   return ret;
25 }
26
27 int main(int argc, char **argv) {
28   char command[500] = "", language[500] = "", test[500] = "", path[500] = "";
29   int ntimes = 10;
30   int i = 0;
```

```
31
32 #ifdef RUNTIME
33   double time_spent;
34   struct timeval tvb, tva;
35 #endif
36
37   struct perf_event_attr pe_loads, pe_stores, pe_misses;
38   long long count_loads, count_stores, count_misses, before_loads, before_stores,
       before_misses;
39   int fd_loads, fd_stores, fd_misses;
40
41   /*
42    * Skylake / Kaby Lake
43    * MEM_INST_RETIRED.ALL_LOADS 81d0
44    * MEM_INST_RETIRED.ALL_STORES 82d0
45    * MEM_LOAD_RETIRED.L3_MISS 20d1
46    *
47    * Ivy Bridge
48    * MEM_UOPS_RETIRED.ALL_LOADS 81d0
49    * MEM_UOPS_RETIRED.ALL_STORES 82d0
50    * MEM_LOAD_UOPS_RETIRED.LLC_MISS 20d1
51    *
52    * Broadwell
53    * MEM_UOPS_RETIRED.ALL_LOADS 81d0
54    * MEM_UOPS_RETIRED.ALL_STORES 82d0
55    * MEM_LOAD_UOPS_RETIRED.L3_MISS 20d1
56    *
57    * Haswell
58    * MEM_UOPS_RETIRED.ALL_LOADS 81d0
59    * MEM_UOPS_RETIRED.ALL_STORES 82d0
60    * MEM_LOAD_UOPS_RETIRED.L3_MISS 20d1
61    *
62    *
63    */
64
65   memset(&pe_loads, 0, sizeof(struct perf_event_attr));
66   pe_loads.type = PERF_TYPE_RAW;
67   pe_loads.size = sizeof(struct perf_event_attr);
68   pe_loads.config = 0x81d0;
69   pe_loads.disabled = 1;
70   pe_loads.inherit = 1;
71   pe_loads.exclude_kernel = 1;
72   pe_loads.exclude_hv = 1;
73
74   memcpy(&pe_stores, &pe_loads, sizeof(struct perf_event_attr));
75   pe_stores.config = 0x82d0;
76
77   memcpy(&pe_misses, &pe_loads, sizeof(struct perf_event_attr));
78   pe_misses.config = 0x20d1;
```

```
79
80    fd_loads = perf_event_open(&pe_loads, 0, -1, -1, 0);
81    if (fd_loads == -1) {
82      fprintf(stderr, "Error opening leader %llx\n", pe_loads.config);
83      exit(EXIT_FAILURE);
84    }
85
86    fd_stores = perf_event_open(&pe_stores, 0, -1, -1, 0);
87    if (fd_stores == -1) {
88      fprintf(stderr, "Error opening leader %llx\n", pe_stores.config);
89      exit(EXIT_FAILURE);
90    }
91
92    fd_misses = perf_event_open(&pe_misses, 0, -1, -1, 0);
93    if (fd_misses == -1) {
94      fprintf(stderr, "Error opening leader %llx\n", pe_misses.config);
95      exit(EXIT_FAILURE);
96    }
97
98
99    FILE *fp;
100
101   //Run command
102   strcat(command, argv[1]);
103   //Language name
104   strcpy(path, "../");
105   strcpy(language, argv[2]);
106   strcat(language, ".csv");
107   strcat(path, language);
108   //Test name
109   strcpy(test, argv[3]);
110
111   if (argc > 4) {
112     ntimes = atoi(argv[4]);
113   }
114
115
116   fp = fopen(path, "a");
117
118
119   for (i = 0; i < ntimes; i++) {
120
121     fprintf(fp, "%s , ", test);
122
123
124 #ifdef RUNTIME
125     //begin = clock();
126     gettimeofday(&tvb, 0);
127 #endif
```

```
128
129
130         ioctl(fd_loads, PERF_EVENT_IOC_RESET, 0);
131         ioctl(fd_stores, PERF_EVENT_IOC_RESET, 0);
132         ioctl(fd_misses, PERF_EVENT_IOC_RESET, 0);
133
134         ioctl(fd_loads, PERF_EVENT_IOC_ENABLE, 0);
135         ioctl(fd_stores, PERF_EVENT_IOC_ENABLE, 0);
136         ioctl(fd_misses, PERF_EVENT_IOC_ENABLE, 0);
137
138         read(fd_loads, &before_loads, sizeof(long long));
139         read(fd_stores, &before_stores, sizeof(long long));
140         read(fd_misses, &before_misses, sizeof(long long));
141
142         system(command);
143
144         read(fd_loads, &count_loads, sizeof(long long));
145         read(fd_stores, &count_stores, sizeof(long long));
146         read(fd_misses, &count_misses, sizeof(long long));
147
148         printf("Used %lld loads, %lld stores and %lld misses\n", count_loads - before_loads,
                count_stores - before_stores,
149                 count_misses - before_misses);
150         fprintf(fp, "%lld, %lld, %lld, ", count_loads - before_loads, count_stores - before_stores
                ,
151                  count_misses - before_misses);
152         ioctl(fd_loads, PERF_EVENT_IOC_DISABLE, 0);
153         ioctl(fd_stores, PERF_EVENT_IOC_DISABLE, 0);
154         ioctl(fd_misses, PERF_EVENT_IOC_DISABLE, 0);
155
156
157 #ifdef RUNTIME
158         gettimeofday(&tva, 0);
159         time_spent = (tva.tv_sec - tvb.tv_sec) * 1000000 + tva.tv_usec - tvb.tv_usec;
160         time_spent = time_spent / 1000;
161 #endif
162
163
164 #ifdef RUNTIME
165         fprintf(fp, " %G \n", time_spent);
166 #endif
167       }
168
169
170       close(fd_loads);
171       close(fd_stores);
172       close(fd_misses);
173       fclose(fp);
174       fflush(stdout);
```

```
175
176     return 0;
177 }
```

Appendix D

compile_all.devnull.py

`compile_all.devnull.py` is an adaption of the original scheduling script[Per+18a] by Pereira et al. [Per+17a] to prevent processing of `stdout` by Python.

```
1  import sys, os
2  from subprocess import call, check_output, Popen, PIPE
3  from lazyme.string import color_print
4
5  path = '.'
6  action = 'compile'
7
8  def file_exists(file_path):
9      if not file_path:
10         return False
11     else:
12         return os.path.isfile(file_path)
13
14
15 def main():
16   FNULL = open(os.devnull, 'w')
17   for root, dirs, files in os.walk(path):
18     print('Checking ' + root)
19     makefile = os.path.join(root, "Makefile")
20     if file_exists(makefile):
21       filelist = os.listdir(root)
22       names = list(map(lambda s: '.'.join(s.split('.')[:-1]), filter(lambda s: s.endswith((".c
             ++", ".go", ".python3")), filelist)))
23       print(names)
24       for filename in names:
25         cmd = 'cd ' + root + '; make NAME=\"' + filename + '\" ' + action
26         #cmd = 'ls -la'
27         pipes = Popen(cmd, shell=True, stdout=FNULL, stderr=PIPE)
28         std_out, std_err = pipes.communicate()
29
```

```
30          if (action == 'compile') | (action == 'run'):
31            if pipes.returncode != 0:
32              # an error happened!
33              err_msg = "%s. Code: %s" % (std_err.strip(), pipes.returncode)
34              color_print('[E] Error on ' + root + ': ', color='red', bold=True)
35              print(err_msg)
36            elif len(std_err):
37              # return code is 0 (no error), but we may want to
38              # do something with the info on std_err
39              # i.e. logger.warning(std_err)
40              color_print('[OK]', color='green')
41            else:
42              color_print('[OK]', color='green')
43          if action == 'measure':
44            call(['sleep', '5'])
45    FNULL.close()
46
47  if __name__ == '__main__':
48    if len(sys.argv) == 2:
49      act = sys.argv[1]
50      if (act == 'compile') | (act == 'run') | (act == 'clean') | (act == 'measure'):
51        color_print('Performing \"' + act + '\" action...', color='yellow', bold=True)
52        action = act
53      else:
54        color_print('Error: Unrecognized action \"' + act + '\"', color='red')
55        sys.exit(1)
56    else:
57      color_print('Performing \"compile\" action...', color='yellow', bold=True)
58      action = 'compile'
59
60    main()
```

Appendix E

compile_all_random.py

`compile_all_random.py` is an adaption of compile_all.devnull.py (Appendix D) to randomize the order of the measured programs.

```
1  import sys, os
2  from subprocess import call, check_output, Popen, PIPE, DEVNULL
3  from lazyme.string import color_print
4  import random
5  from itertools import repeat
6
7  random.seed()
8
9  path = '.'
10 action = 'compile'
11
12 def file_exists(file_path):
13     if not file_path:
14         return False
15     else:
16         return os.path.isfile(file_path)
17
18
19 def main():
20   todo = []
21   for root, dirs, files in os.walk(path):
22     print('Checking ' + root)
23     makefile = os.path.join(root, "Makefile")
24     if file_exists(makefile):
25       filelist = os.listdir(root)
26       names = list(map(lambda s: '.'.join(s.split('.')[:-1]), filter(lambda s: s.endswith((".c
             ++", ".go", ".python3")), filelist)))
27       print(names)
28       for filename in names:
29         cmd = 'cd ' + root + '; make NAME=\"' + filename + '\" NTIMES=1 ' + action
```

```
30          # Put the command on the list
31          if action == 'measure':
32            todo.extend(repeat((root, cmd), 10))
33          else:
34            todo.append((root, cmd))
35          #cmd = 'ls -la'
36    # Shuffle the commands to obtain a random order
37    random.shuffle(todo)
38    for (root, cmd) in todo:
39      pipes = Popen(cmd, shell=True, stdout=DEVNULL, stderr=PIPE)
40      _, std_err = pipes.communicate()
41
42      if (action == 'compile') | (action == 'run'):
43        if pipes.returncode != 0:
44          # an error happened!
45          err_msg = "%s. Code: %s" % (std_err.strip(), pipes.returncode)
46          color_print('[E] Error on ' + root + ': ', color='red', bold=True)
47          print(err_msg)
48        elif len(std_err):
49          # return code is 0 (no error), but we may want to
50          # do something with the info on std_err
51          # i.e. logger.warning(std_err)
52          color_print('[OK]', color='green')
53        else:
54          color_print('[OK]', color='green')
55      if action == 'measure':
56        call(['sleep', '5'])
57
58  if __name__ == '__main__':
59    if len(sys.argv) == 2:
60      act = sys.argv[1]
61      if (act == 'compile') | (act == 'run') | (act == 'clean') | (act == 'measure'):
62        color_print('Performing \"' + act + '\" action...', color='yellow', bold=True)
63        action = act
64      else:
65        color_print('Error: Unrecognized action \"' + act + '\"', color='red')
66        sys.exit(1)
67    else:
68      color_print('Performing \"compile\" action...', color='yellow', bold=True)
69      action = 'compile'
70
71    main()
```