

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Program transformations for improving software energy efficiency

Bernardo Santos



Mestrado em Engenharia Informática e Computação

Supervisor: João Paulo Fernandes

Second Supervisor: Maja H. Kirkeby

Third Supervisor: Alberto Pardo

July 20, 2022

Program transformations for improving software energy efficiency

Bernardo Santos

Mestrado em Engenharia Informática e Computação

July 20, 2022

Abstract

Sustainable consumption of energy is a growing concern nowadays, as we start to realize just how unsustainable it is for our consumption to be mostly supported by non-renewable sources. It is only logical that the concept of responsible energy consumption should be applied to software development. In this dissertation, we set out to understand how program transformation techniques, in the form of compiler optimizations, impact software's energy consumption, in the context of Haskell and GHC. Compilers are very appropriate study objects for energy consumption reduction since they already relieve programmers from a responsibility they are typically not trained for. What if we could delegate one more responsibility to the compiler?

In order to understand how GHC optimization flags influence Haskell programs, a set of 26 flags and a set of 33 benchmarks were selected. A quantitative experiment was performed, in which the energy consumption and execution time of the benchmarks were collected when compiled with no optimizations and with an optimization at a time. By analyzing the resulting data, we concluded that, while we are not able to predict the influence an optimization will have on a program, some flags are more likely to either improve, worsen or maintain a similar performance. However, most of the studied flags increased the energy consumption and execution time of more benchmarks than they decreased. With an analysis of the energy consumption and execution time pairs, we were also able to conclude that these metrics are strongly related, and as one increases or decreases, the other increases or decreases, respectively.

Studying the impact of program transformations can be a step that leads to advances across programming languages, helping to incorporate energy efficiency as a concern for programming language designers and developers.

Keywords: Program transformations, Energy consumption, Compiler optimizations, Programming languages

Resumo

O consumo sustentável de energia é uma preocupação crescente nos dias de hoje, à medida que começamos a perceber o quão insustentável é que o nosso consumo seja suportado principalmente por fontes não renováveis. É apenas lógico que o conceito de consumo responsável de energia seja aplicado ao desenvolvimento de software. Nesta dissertação, pretendemos compreender como as técnicas de transformação de programas, na forma de otimizações do compiladores, impactam o consumo de energia do software, no contexto de Haskell e do GHC. Os compiladores são objetos de estudo muito apropriados para redução do consumo de energia, pois já libertam os programadores de uma responsabilidade para a qual normalmente não são treinados. E se pudéssemos delegar mais uma responsabilidade ao compilador?

Para entender como as otimizações do GHC influenciam os programas de Haskell, foram selecionados um conjunto de 26 otimizações e um conjunto de 33 programas. Foi realizada uma experiência quantitativa, na qual o consumo de energia e o tempo de execução dos programas foram coletados quando compilados sem otimizações e com uma otimização de cada vez. Ao analisar os dados resultantes, concluímos que, embora não consigamos prever a influência que uma otimização terá num programa, algumas otimizações têm maior probabilidade de melhorar, piorar ou manter um desempenho semelhante. No entanto, a maioria das otimizações estudadas aumentou o consumo de energia e o tempo de execução de mais programas que aquelas que diminuiu. Com uma análise dos pares de consumo de energia e tempo de execução, também pudemos concluir que estas métricas estão fortemente relacionadas e, à medida que uma aumenta ou diminui, a outra também aumenta ou diminui, respectivamente.

Estudar o impacto das transformações de programas pode ser um passo que leva a avanços nas linguagens de programação, ajudando a incorporar a eficiência energética como uma preocupação para designers e desenvolvedores de linguagens de programação.

Keywords: Program transformations, Energy consumption, Compiler optimizations, Programming languages

Contents

1	Introduction	1
1.1	World's Energy Consumption	1
1.2	Software's Energy Efficiency	2
1.3	Objectives	2
1.4	Document Structure	3
2	State of the Art	4
2.1	The functional paradigm, Haskell, and GHC	4
2.2	What influences software's energy consumption?	5
2.2.1	Programming Languages	5
2.2.2	Data Structures	6
2.2.3	Compiler Optimizations	6
2.2.4	Other Implementation Choices	6
2.3	Measuring software energy consumption	6
2.3.1	Methods	7
2.3.2	Technologies	7
2.4	Summary	8
3	Problem Statement and Methodology	10
3.1	Problem Statement	10
3.2	Methodology	11
3.2.1	Identify programs to study	11
3.2.2	Identify compiler optimizations	12
3.2.3	Develop a framework to automate the experiment	12
3.2.4	Analyse results	13
3.2.5	Hardware & Software Specifications	14
3.3	Summary	14
4	Results Analysis and Discussion	16
4.1	Energy Consumption Analysis	16
4.2	Execution Time Analysis	23
4.3	Relating Execution Time and Energy Consumption	27
4.4	Main Takeaways	30
4.5	Summary	31
5	Conclusions	32
5.1	Research Findings	32
5.2	Possible Limitations	33

5.3	Future Work	33
5.4	Conclusions	34
	References	35
A	Charts and Statistical Tests	38
A.1	Energy Consumption	38
A.1.1	Charts	38
A.1.2	Statistical Tests	52
A.2	Execution Time	69
A.2.1	Charts	69
A.2.2	Statistical Tests	83

List of Figures

1.1	Global primary energy consumption by source (taken from [27])	2
4.1	Energy consumption of <code>parfib</code> , <code>fasta</code> and constraints, when compiled with <code>-O0</code> and <code>-fcmm-sink</code>	17
4.2	Energy consumption of <code>parfib</code> , <code>fasta</code> and constraints, when compiled with <code>-O0</code> and <code>-fcase-folding</code>	17
4.3	Energy consumption of <code>parfib</code> , <code>fasta</code> and constraints, when compiled with <code>-O0</code> and <code>-floopification</code>	18
4.4	Energy consumption of <code>coins</code> , <code>callback001</code> and <code>k-nucleotide</code> , when compiled with <code>-O0</code> and <code>-floopification</code>	19
4.5	Execution time of <code>parfib</code> , <code>fasta</code> and constraints, when compiled with <code>-O0</code> and <code>-fcmm-sink</code>	23
4.6	Execution time of <code>parfib</code> , <code>fasta</code> and constraints, when compiled with <code>-O0</code> and <code>-fcase-folding</code>	24
4.7	Execution time of <code>parfib</code> , <code>fasta</code> and constraints, when compiled with <code>-O0</code> and <code>-floopification</code>	24
4.8	Scatter plot of energy consumption and execution time of all benchmarks compiled with all optimization flags	28
4.9	Histogram of Spearman's correlation coefficient for each benchmark-flag combination with $\rho < 0.05$	29
4.10	Scatter plot of energy consumption and execution time of <code>banner</code> when compiled with <code>-fcmm-elim-common-blocks</code>	30
4.11	Scatter plot of energy consumption and execution time of <code>banner</code> when compiled with <code>-fstg-lift-lams</code>	30
A.1	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-fcase-merge</code>	38
A.2	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-fcase-folding</code>	39
A.3	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-fcall-arity</code>	39
A.4	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-fexitification</code>	40
A.5	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-fcmm-elim-common-blocks</code>	40
A.6	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-fcmm-sink</code>	41
A.7	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-fcpr-anal</code>	41
A.8	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-fcse</code>	42

A.9	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-fstg-cse</code>	42
A.10	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-fdmd-tx-dict-sel</code>	43
A.11	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-fdo-eta-reduction</code>	43
A.12	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-fdo-lambda-eta-expansion</code>	44
A.13	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-ffloat-in</code>	44
A.14	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-ffull-laziness</code>	45
A.15	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-fignore-asserts</code>	45
A.16	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-fliberate-case</code>	46
A.17	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-floopification</code>	46
A.18	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-fllvm-pass-vectors-in-regs</code>	47
A.19	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-fspec-constr</code>	47
A.20	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-fspecialise</code>	48
A.21	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-fcross-module-specialise</code>	48
A.22	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-fsolve-constant-dicts</code>	49
A.23	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-fstrictness</code>	49
A.24	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-funbox-small-strict-fields</code>	50
A.25	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-fblock-layout-cfg</code>	50
A.26	Energy consumption of all benchmarks when compiled with <code>-O0</code> and <code>-fstg-lift-lams</code>	51
A.27	Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-fcase-merge</code>	69
A.28	Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-fcase-folding</code>	70
A.29	Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-fcall-arity</code>	70
A.30	Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-fexitification</code>	71
A.31	Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-fcmm-elim-common-blocks</code>	71
A.32	Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-fcmm-sink</code>	72
A.33	Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-fcpr-anal</code>	72
A.34	Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-fcse</code>	73
A.35	Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-fstg-cse</code>	73
A.36	Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-fdmd-tx-dict-sel</code>	74
A.37	Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-fdo-eta-reduction</code>	74

A.38 Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-fdo-lambda-eta-expansion</code>	75
A.39 Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-ffloat-in</code> . . .	75
A.40 Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-ffull-laziness</code>	76
A.41 Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-fignore-asserts</code>	76
A.42 Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-fliberate-case</code>	77
A.43 Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-floopification</code>	77
A.44 Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-fllvm-pass-vectors-in-regs</code>	78
A.45 Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-fspec-constr</code> .	78
A.46 Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-fspecialise</code> . .	79
A.47 Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-fcross-module-specialise</code>	79
A.48 Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-fsolve-constant-dicts</code>	80
A.49 Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-fstrictness</code> . .	80
A.50 Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-funbox-small-strict-fields</code>	81
A.51 Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-fblock-layout-cfg</code>	81
A.52 Execution time of all benchmarks when compiled with <code>-O0</code> and <code>-fstg-lift-lams</code>	82

List of Tables

3.1	Selected NoFib benchmarks	11
3.2	Optimization flags and "packages" which enable them	12
4.1	Number of optimized executables that have statistically significant differences compared to their non-optimized versions, that do not, and that did not respect the assumptions of ANOVA, by optimization flag	20
4.2	Number of optimized executables in which there was an increase, no significant difference and a decrease in energy consumption for each optimization flag, and corresponding percentages	21
4.3	Effect sizes of energy consumption of <i>fasta</i> for each optimization flag	21
4.4	Number of optimization flags that for each benchmark increased, maintained and decreased energy consumption and respective percentages	22
4.5	Number of optimized executables that have statistically significant differences compared to their non-optimized versions, that do not, and that did not respect the assumptions of ANOVA, by optimization flag	25
4.6	Number of optimized executables in which there was an increase, no significant difference and a decrease in execution time for each optimization flag, and corresponding percentages	26
4.7	Number of optimization flags that for each benchmark increased, maintained and decreased execution time and respective percentages	27
A.1	Benchmarks that have a statistically significant difference in energy consumption when compared to their non-optimized versions ($p < 0.05$), that do not ($p > 0.05$), and that did not respect the assumptions of ANOVA, by optimization flag	60
A.2	Effect sizes of the energy consumption of all significantly different benchmark-flag combinations	68
A.3	Benchmarks that have a statistically significant difference in execution time when compared to their non-optimized versions ($p < 0.05$), that do not ($p > 0.05$), and that did not respect the assumptions of ANOVA, by optimization flag	91
A.4	Effect sizes of the execution time of all significantly different benchmark-flag combinations	96

Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
CLGB	Computer Language Benchmark Game
CPU	Central Processing Unit
ICT	Information and Communications Technology
IT	Information Technology
GHC	Glasgow Haskell Compiler
GCC	GNU Compiler Collection
RAPL	Running Average Power Limit
MSR	Model Specific Registers

Chapter 1

Introduction

Software energy efficiency may be an important requirement in different contexts, for example, in embedded systems it is paramount that software consumes the least possible amount of energy, since these devices are battery-powered. However, this dissertation is concerned with the impact that software currently has on the environment, and aims to study ways to reduce it.

This chapter begins by presenting an overview of the world's energy consumption, demonstrating the importance of developing energy efficient software. It goes on to expose how this requirement is a concern for clients and developers, and analyse the current level of knowledge and resources in this area. Furthermore, the main motivations and objectives of the work are presented, as well as an overview of the organization of this document.

1.1 World's Energy Consumption

The world's energy consumption has been steadily rising since the 1950's, as seen in Figure 1.1. Just in 2018, the world consumed six times the amount of energy consumed in 1950. It comes as no surprise that the vast majority of this energy is produced from non-renewable sources, making our consumption unsustainable.

A sector that is present in all of our daily lives and has considerable impact on energy consumption is the Information and Communications Technology (ICT) sector. It was responsible for more than 7% of the energy consumed in 2020 [2], and, worst-case scenario, this consumption can reach a whopping 51% by 2030 [3]. Moreover, it is predicted that it will be responsible for 23% of all greenhouse gas emissions.

Many companies, like Microsoft and Google, have taken measures to reduce the environmental impact of their data centers [15]. They have relocated them to places where they can be fully powered by clean energy from renewable sources, and even colder locations, to cut on power used for cooling systems. These great initiatives focus on the physical aspects of IT systems, and do not consider software as a means to reduce energy consumption.

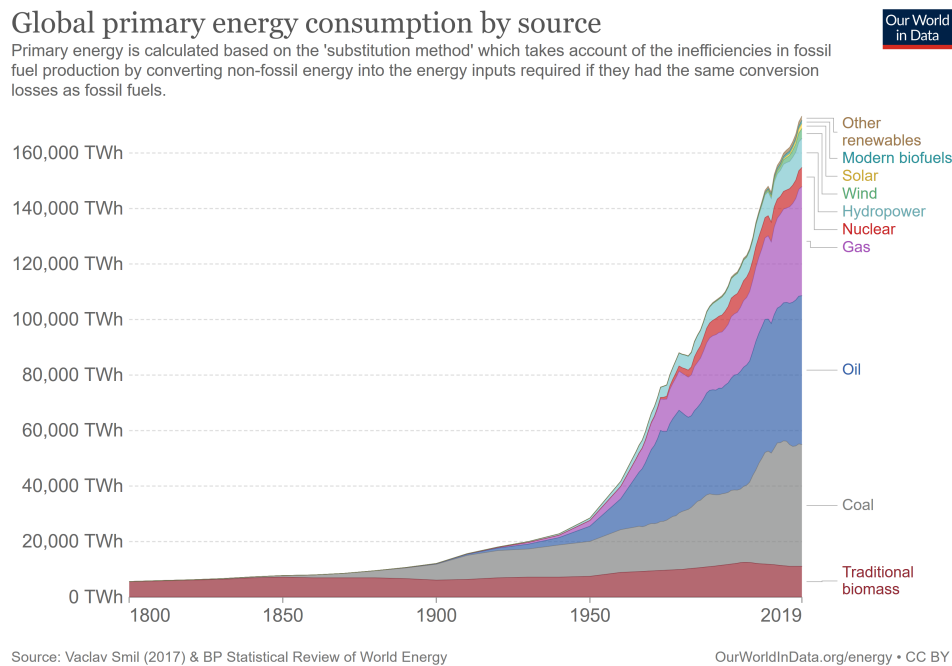


Figure 1.1: Global primary energy consumption by source (taken from [27])

1.2 Software's Energy Efficiency

The Green Computing movement aims to reduce the impact of computing in the environment, and one of the possible approaches to reach such goal is software optimization. There are many well-established studies on how to reduce the usage of resources, such as memory or execution time. But what about energy consumption?

Software's energy efficiency is a non-functional requirement, i.e, a requirement that specifies criteria that can be used to judge the operation of a system instead of specific behaviours, and it has been growing ever more important. However, when asked if they took into account power consumption when developing software, only 18% of the participants in [22] claimed to do so, and only 14% considered it a requirement. The same study concludes that programmers "lack consistent knowledge about the energy consumption of software", which is not helped by the fact that they struggle to find adequate answers to their question regarding this topic [26].

This work aims to help developers improve the energy efficiency of their software by studying the impact of program transformations used for compiler optimizations.

1.3 Objectives

Nowadays, most developers use high-level programming languages, since they allow them to quickly write highly portable code. This high-level code must be translated to machine code, which is often responsibility of the compiler, that can also apply optimizations to enhance the

software's performance. These optimizations can improve both execution time and resource usage, but what effect do these transformations have on the program's energy consumption?

To determine this, Haskell and the Glasgow Haskell Compiler (GHC), have been chosen as study objects for this dissertation. Due to the purity of functional languages, there are more opportunities for compile time optimizations to be applied in this paradigm. Haskell happens to be one of the most popular functional languages at the moment, and GHC, apart from being the most used compiler [8], also has an extensive catalog of optimizations.

This dissertation intends to understand how optimizations applied by GHC impact Haskell programs' energy efficiency, and how this impact can be leveraged to reduce software's energy consumption.

1.4 Document Structure

In Chapter 2, we start by presenting the functional paradigm and the characteristics that make it interesting for this study. Following, we present a review of the literature on the topic of software energy efficiency, as well as the comparison of the methodologies of the reviewed works and the technologies used to measure energy consumption.

The research questions this work intends to provide an answer to, and the methodology adopted to do so, are presented in Chapter 3.

In Chapter 4, the results of the statistical tests performed on the data collected during the experiment are presented and analysed, in order to give answers to the research questions of the previous chapter.

Finally, Chapter 5 concludes this work, by providing the answers to the research questions, identifying the possible limitations or threats to the validity of this work, and identifying some possibilities for future work.

Chapter 2

State of the Art

In Section 2.1, the functional paradigm is presented, along with the characteristics that make it keen for studying the impact of compiler optimizations. Haskell's presence in the industry and some features of GHC are also discussed. The following section, 2.2, analyses some of the related work in the software energy consumption topic, from works focused on the consumption patterns of different programming languages, to data structures or other small implementation choices. Finally, in Section 2.3, the methods and technologies used by the reviewed works are presented.

2.1 The functional paradigm, Haskell, and GHC

The functional paradigm is particularly suitable to study the impact of optimizations due to some of the characteristics of pure functional programs. Referential transparency¹ is one of them, expressions that are referential transparent can be replaced with their corresponding values without changing the program's behaviour. This makes code context-independent, meaning expressions can be executed in any order or any context and the result will always be the same.

In [13], John Hughes writes: "Our ability to decompose a problem into parts depends directly on our ability to glue solutions together. To support modular programming, a language must provide good glue. Functional programming languages provide two new kinds of glue — higher-order functions and lazy evaluation.". Support for higher-order functions², functions that take functions as arguments or return functions, makes it possible to build programs as a series of functions glued together to produce new results. While lazy evaluation³, which is delaying evaluation of an expression until its value is needed, together with referential transparency, can enable caching optimizations such as memoization⁴, which stores values of a function for certain input values so that it is not calculated every time the function is called with those inputs.

¹https://wiki.haskell.org/Referential_transparency, accessed 25/02/2022

²https://wiki.haskell.org/Higher_order_function, accessed 25/02/2022

³https://wiki.haskell.org/Lazy_evaluation, accessed 25/02/2022

⁴<https://wiki.haskell.org/Memoization>, accessed 25/02/2022

Immutability of data, another consequence of referential transparency, can many times allow the compiler to make assumptions that are unsafe in other paradigms, increasing opportunities for function inlining⁵. This means the compiler inserts the function body into the compiled code instead of inserting a function call.

In lazy purely functional languages, such as Haskell, the compiler has the freedom to reorder and combine evaluation of expressions. An example of this is deforestation⁶, which is able to remove intermediate data structures, thus improving the program's performance.

According to the Stack Overflow Developer Survey 2021⁷, Haskell was the most popular lazy purely functional language. This is reflected in the industry, as Haskell is used in production by more than 200 companies [12, 4], from banking entities, such as the the Standard Chartered, to IT companies, like Facebook, which has been using Haskell since 2015 to fight spam, malware and other abuses to their system [17].

Haskell is a compiled language, and, as mentioned before, the most popular Haskell compiler is GHC [8]. It started being developed in the 1990's, as part of an academic research funded by the UK government, and has been in active development until today [18]. It has an extensive catalog of optimizations⁸, making it the perfect compiler to use for this study.

2.2 What influences software's energy consumption?

Software energy efficiency is a non-functional requirement that has been getting more and more attention. Over time, plenty of works have investigated different aspects that might influence software energy consumption. This section presents a review of some of those works.

2.2.1 Programming Languages

In [5], the authors set out to establish a ranking of 10 different programming languages based on their energy efficiency. They conclude that compiled languages are both faster and more energy efficient than their interpreted counterparts, a conclusion also reached in [1]. Also, there are some cases where slower languages end up consuming less energy, proving that energy consumption is not always directly proportional to execution time. One of the limitations identified by the authors was the fact that only the CPU energy consumption was monitored, leaving out other components, such as memory. This concern is addressed in [23], where they focus on relating energy consumption, execution time and memory usage. The 27 languages studied were ranked according to 4 different sets of objectives: Time & Memory, Energy & Time, Energy & Memory and Energy & Time & Memory. These results were later validated in [24].

⁵https://wiki.haskell.org/Function_inlining, accessed 26/02/2022

⁶<https://wiki.haskell.org/Deforestation>, accessed 25/02/2022

⁷<https://insights.stackoverflow.com/survey/2021>, accessed 26/02/2022

⁸https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/using-optimisation.html, accessed 26/02/2022

2.2.2 Data Structures

When deciding what data structure to use, developers have to consider trade-offs between time and memory complexity. The goal of the authors in [20] is to study that trade-off when considering energy consumption. For this purpose they present an empirical evaluation of different Haskell data structure implementations. They conclude that for the same operation different implementations can have a difference of up to 95% in energy consumption.

In [25], the authors study the energy consumption of the different Java Collection Framework implementations. For each method of an implementation they present energy consumption measurements when handling small, medium and large data sets. With these results they were able to optimize a set of Java programs and reduce energy consumption by 6.2%.

2.2.3 Compiler Optimizations

In [1], the authors study the impact of GCC's `-O1`, `-O2` and `-O3` compiler flags. They conclude that compiler options do have considerable influence on energy efficiency. Code optimized using the `-O2` and `-O3` flags is more energy efficient.

Similarly, in [20] a comparison between GHC's `-O0`, `-O1` and `-O2` is presented. For some operations in some implementations, the non-optimized code (`-O0`) performed better, both in terms of execution time and energy consumption, than the same code with optimizations enabled (`-O1` and `-O2`). In the GHC User's Guide⁹, the GHC authors themselves warn users about the lack of consistency of the `-O*` flags. According to this study, these optimizations can either improve or worsen the software's performance.

The works above study "packages" of optimizations. In both GHC and GCC, the `-O*` flags, are bundles of several singular optimizations assembled for general purpose use. For example, in GHC, the `-O1` is meant to generate good-quality code in a reasonable amount of time. Unlike these works, this dissertation focuses on singular optimizations.

2.2.4 Other Implementation Choices

In [1], in addition to studying the impact of different programming languages and taking a dive into compiler optimizations, the authors also consider the influence that small differences in implementation may have. More specifically the use of C++'s `new` and `delete` operators instead of C's `malloc` and `free`. Results show that for large input size `malloc` consumes less energy than `new`.

2.3 Measuring software energy consumption

In each of the works analysed in Section 2.2, the authors describe the methods and technologies they used to measure and analyse software energy consumption. This section focus on comparing

⁹https://downloads.haskell.org/ghc/latest/docs/html/users_guide/using-optimisation.html, accessed 23/02/2022

and discussing both, the approaches taken in those , and the various technologies used to measure energy consumption.

2.3.1 Methods

The main goal of [20] was to evaluate data structures' energy consumption. For this purpose the authors selected a benchmark suite that allowed them to evaluate each operation individually. These micro-benchmarks were executed multiple times and both execution time and energy consumption were collected. A time limit was also imposed to ensure practicality.

In [25] the method followed is similar to the one in [20]. In this work, the authors executed each test 10 times for data structures with 25000, 250000 and 1000000 elements. The results presented are the average of the execution time and energy consumption measurements, after the lowest and highest 20% outliers were removed.

In both [5] and [24], the selected programs were benchmarks of the Computer Language Benchmark Game (CLBG). Each of the benchmarks was compiled according to the instructions and ran individually to assure its correctness. A framework was developed in order to gather execution time, peak memory usage and energy consumption. After this, each benchmark was executed and measured 10 times, and between each measurement the system was left to cool. Just like in [25], the lowest and highest outliers were removed before processing and analysing the results.

2.3.2 Technologies

There are several possible methods to measure energy consumption. In [16], the authors classify them into three categories: software-based, hardware-based and a hybrid approaches.

2.3.2.1 Software-based

The software-based approaches use software to estimate energy consumption instead of actually measuring it. Apart from their reduced cost and ease of use, these approaches generally allow for greater granularity, meaning that it is possible to analyse the energy consumption of specific processes, or even different procedures inside a process. The main drawback is that the obtained results are less accurate than those of other approaches, this is due to the fact that they are actually estimates.

All of the works analysed in Section 2.2 use this approach. In [5, 20, 23, 24, 25], the tool used to estimate energy consumption was Intel's Running Average Power Limit (RAPL). The exception is [1], in which the Intel Power Governor¹⁰ is used, however this tool uses RAPL internally. RAPL is an interface, available in Intel's processors since their second generation, that allows for estimation of energy consumption by reading Model Specific Registers (MSR). It is able to provide very accurate measurements, as proven in [28, 11].

¹⁰<https://github.com/intel/powergov>, accessed 24/02/2022

Another tool to measure energy consumption is pTop [6], it is a process-level power profiling tool. It provides power consumption measures of running processes in joules. It provides measures for the CPU, network interface, memory and hard disk. Its overhead is relatively low, but not negligible, unlike RAPL's [5].

PowerTOP¹¹, a Linux tool to diagnose issues with power consumption and power management, is also able to report the energy consumption of applications and system components, namely the CPU, network interface and monitor. Just like pTop, it can only do so at the process-level.

2.3.2.2 Hardware-based

These approaches use physical power-meters to measure the energy consumption of the device, or components of the device, where the software is running. Even though the measurements are more accurate, there are several downsides to these approaches. For instance, it is impossible to determine the consumption of a particular process. The costs and availability of the material needed also exceed those of the previous approaches.

2.3.2.3 Hybrid

This is a combination of both the previous approaches. A hardware device is used to measure energy consumption, while the software component collects additional information from the device's hardware. The main problem of this approach is the cost, just like in hardware based ones. However, results can be as fine-grained as the ones obtained through software-based approaches.

A tool that fits this description is PowerScope [9]. It uses a digital multimeter to measure energy consumption and another computer to control it and store the measurements. It is able to map energy consumption to program structure, meaning that it can determine the consumption of a particular process, and even of different procedures inside a process.

PowerPack [10] is another hybrid alternative. The hardware part of it is composed of sensors, meters, circuits and data acquisition devices that enable direct power measurement and instrumentation. While the software components include drivers for those meters and sensors, as well as APIs for controlling power profiling and code synchronization. These allow for fine-grain component-level measurements.

2.4 Summary

This chapter begins by showing that pure functional programs present themselves as a great candidate to study the impact of program transformations, implemented as compiler optimizations. Being that one of the reasons Haskell was chosen, along with its popularity amongst other functional languages, and its presence in the industry. When it comes to GHC, it was a logical choice,

¹¹<https://github.com/fenrus75/powertop>, accessed 24/02/2022

as it is the most developed and popular Haskell compiler, and has a great number of optimization options.

After a review of the literature, it is possible to conclude that much quality work has been done in this area. However, none of the reviewed works really target the same problem as this dissertation. While the ones that study compiler optimizations come close, this work intends on understanding the impact of individual optimizations. Even though they don't target the same issue, these works provide much useful information on how to accurately measure software energy consumption, both method and technology wise.

Chapter 3

Problem Statement and Methodology

In this chapter, we define the problem we propose to address and identify the main research questions we aim at providing answers to; this is done in Section 3.1. Then, in Section 3.2, we describe the methodology we adopted in our study.

3.1 Problem Statement

As stated in Chapter 1, the world’s energy consumption is unsustainable and the ICT sector is responsible for a considerable part of this consumption. Improving software energy efficiency would be a step towards making our consumption more sustainable.

It is possible to find in the literature different ways to reduce software energy consumption. These different possibilities range from choosing more efficient languages [23] to using specific data structures instead of alternative ones [25]. The challenge regarding their practical use, however, is that they require a deep knowledge from programmers, and a knowledge programmers are not trained to have.

In our work, we leverage an alternative approach and study whether an existing compiler can already be assumed to generate energy-efficient executables. Compiler optimizations are already used to obtain more efficient software in terms of execution time and memory usage, which makes them particularly attractive to study. This is even more appealing since, if our hypothesis confirms, they can be used in a transparent way by programmers to improve the energy efficiency of their software. The idea is that programmers, in principle, could then write software as usual, and delegate on the compiler to produce its energy-optimized versions.

Ultimately, this dissertation intends to answer three research questions, in the context of Haskell and GHC:

RQ₁ How do compiler optimizations influence the energy efficiency of programs?

RQ₂ How do compiler optimizations influence the execution time of programs?

RQ₃ How are time and energy consumption related?

3.2 Methodology

This section describes the method used in this work, which was idealized taking into consideration the methods used in the reviewed literature. The developed method is comprised of five distinct steps. First, we selected 33 programs and 26 compiler optimization flags to serve as a base for our study. The methodology for selecting the programs and optimization flags is described in further detail in Section 3.2.1 and Section 3.2.2, respectively. Compiling, executing and collecting data for every execution of a benchmark-flag combination manually would be unfeasible, as such we implemented a framework to automate the experiment, which is described in Section 3.2.3. After collecting the results, we perform a statistical analysis of the data, which is described in Section 3.2.4. All of the source code developed and the data collected during the experiment can be found at <https://github.com/bernas670/masters-thesis>.

3.2.1 Identify programs to study

The first step was identifying the set of programs that would be compiled with and without applying optimizations. For this purpose, the Haskell benchmark suite NoFib¹ was adopted. This suite is composed of many different benchmarks, from actual command line applications to algorithms like Fast Fourier Transform and benchmarks from The Computer Language Benchmark Game².

NoFib contains over a hundred benchmarks divided into various categories. Compiling and executing all benchmarks with different optimization flags would be infeasible due to time constraints. As such, 20% of the programs from each category were selected to be used in our study, which are presented in Table 3.1.

Category	Benchmarks
gc	cacheprof, constraints, hash
imaginary	bernoulli, exp3_8, gen_regexps, integrate
parallel	blackscholes, coins, parfib, queens
real	anna, cacheprof, compress2, fluid, gg, parser
shootout	binary-trees, fasta, k-nucleotide, pidigits
smp	callback001, callback002, chan
spectral	ansi, atom, awards, banner, clausify, cryptarithm1, life, power, treejoin

Table 3.1: Selected NoFib benchmarks

¹<https://gitlab.haskell.org/ghc/nofib>, accessed 26/02/2022

²<https://benchmarksgame-team.pages.debian.net/benchmarksgame/>, accessed 26/02/2022

3.2.2 Identify compiler optimizations

As previously mentioned, GHC has an extensive catalog of optimizations, but collecting a significant number of measurements that would allow for a comparison between all of them would not be viable.

Like other compilers, GHC offers convenient "packages" of optimization flags that are intended to be used with a general goal in mind:

- `-O0` - turns off all optimizations
- `-O1` - generates good quality code without taking too long to compile
- `-O2` - applies all non-dangerous optimizations, even if it means significantly longer compile times

The flags considered in this study are the ones that comprise both `-O1` and `-O2` packages of optimizations, Table 3.2. Flags such as `-fliberate-case-threshold`, that control thresholds for other optimizations were not considered, so the values in use are default.

Flag	-O1	-O2	Flag	-O1	-O2
<code>-fcase-merge</code>	×	×	<code>-ffull-laziness</code>	×	×
<code>-fcase-folding</code>	×	×	<code>-fignore-asserts</code>	×	×
<code>-fcall-arity</code>	×	×	<code>-fliberate-case</code>		×
<code>-fexitification</code>	×	×	<code>-floopification</code>	×	×
<code>-fcmm-elim-common-blocks</code>	×	×	<code>-fllvm-pass-vectors-in-regs</code>	×	×
<code>-fcmm-sink</code>	×	×	<code>-fspec-constr</code>		×
<code>-fcpr-anal</code>	×	×	<code>-fspecialise</code>	×	×
<code>-fcse</code>	×	×	<code>-fcross-module-specialise</code>	×	×
<code>-fstg-cse</code>	×	×	<code>-fsolve-constant-dicts</code>	×	×
<code>-fdmd-tx-dict-sel</code>	×	×	<code>-fstrictness</code>	×	×
<code>-fdo-eta-reduction</code>	×	×	<code>-funbox-small-strict-fields</code>	×	×
<code>-fdo-lambda-eta-expansion</code>	×	×	<code>-fblock-layout-cfg</code>	×	×
<code>-ffloat-in</code>	×	×	<code>-fstg-lift-lams</code>	×	×

Table 3.2: Optimization flags and "packages" which enable them

3.2.3 Develop a framework to automate the experiment

For each program-flag combination, ten measurements of the energy consumed by the program were collected using RAPL, one of the tools presented in Section 2.3.2. Apart from its negligible overhead and accurate results [5, 28, 11], this tool was also used in similar studies [5, 20, 23, 24, 25]. As done in the aforementioned studies, the program is executed resorting to the C language's `system` call, and energy measurements are taken immediately before and after its execution. The program's energy consumption is the delta between these two measurements.

To automate compilation, execution, and data collection, we took advantage of NoFib's make build system, by using built-in options and modifying it in order to better fit our use case.

When it comes to options, the `EXTRA_HC_OPTS` and `NoFibRuns` were used. The first is used to pass extra options for compilation, which allowed us to specify both flags and the location of packages demanded by some benchmarks. `NoFibRuns` determines how many times the benchmark is executed. For example, if we were to compile a benchmark only with the `-fcase-merge` optimization active and run it five times, the make command would look something like `make NoFibRuns=5 EXTRA_HC_OPTS="-package-db <path to package db> -O0 -fcase-merge"`. The `-O0` flag deactivates all previously set flags, ensuring that only the flags specified after it are active.

A wrapper program was developed to collect and save the energy consumption measurements. After inspecting the build system's flow, we concluded that the best approach was to slightly modify the `mk/target.mk` file, which is responsible for executing the benchmark `NoFibRuns` times.

One of the biggest concerns when collecting samples is ensuring the number of variables in the system is kept to a minimum. One of the aspects that has been proven to influence CPU energy consumption is temperature [30]. While in other works, the authors tackled this problem by introducing a fixed cooldown period between program executions, this framework goes further and considers the system's current temperature. As soon as it starts executing, it collects ten measurements of the CPU temperature with an interval of five seconds between them. It then calculates the base temperature, the average of all samples. Before executing a benchmark, the CPU temperature is sampled every two seconds, and it only starts running when three consecutive samples are lower or at most one celsius degree higher than the base temperature.

The framework takes as input two text files, one with the benchmarks and another with the flags, and outputs a CSV file containing the execution time and energy consumption of each benchmark execution.

3.2.4 Analyse results

To draw conclusions from the collected data, we first need to make sure that the measurements for each benchmark compiled without any optimization and with a single optimization are comparable. To do so, an analysis of variance test [31], or ANOVA, was performed. This statistical test indicates whether or not two groups are significantly different, allowing them to be compared against each other. ANOVA has three assumptions:

1. all samples are drawn independently from each other
2. data is normally distributed
3. homogeneity of variance

The first assumption is automatically fulfilled because of how the data was acquired. To check for data normality, the outliers are removed, and then the Shapiro-Wilks test [29] is performed for each program-flag combination. The normally distributed combinations are then tested for homogeneity of variance using Bartlett's test [19].

In order to better understand the effects that each optimization had on each benchmark, the effect size was calculated using Hedge's g [7], which is the most adequate method for the sample size used in this study ($n < 10$).

The correlation between execution time and energy consumption can also be analysed by calculating Spearman's correlation coefficient [31] for each program-flag combination.

3.2.5 Hardware & Software Specifications

After developing and testing the framework, it was time to collect results. A machine was set up with a minimal installation of Ubuntu Server 22.04 with kernel version 5.15.0-33-generic in order to minimize the number of services running alongside the experiments. The system's specifications are as follows: Intel(R) Core(TM) i7-4720HQ CPU @ 2.60GHz, with 16 GiB RAM @ 1600 MHz.

The version of GHC used was the one available in the Ubuntu repositories at the time of the experiment, 8.8.4.

The checkout of NoFib at the time was `bca0196`. As mentioned before, some of the benchmarks in NoFib require external packages, the packages and respective versions that were used were the following:

```
old-locale-1.0.0.7
old-time-1.1.0.3
parallel-3.2.2.0
primitive-0.7.4.0
random-1.2.1.1
regex-base-0.94.0.2
regex-compat-0.95.2.1
regex-posix-0.96.0.1
splitmix-0.1.0.4
unboxed-ref-0.4.0.0
```

3.3 Summary

Section 3.1, emphasises the environmental problem that we face, and reinforces that reducing the energy consumption of software is a step in the right direction. But it is unrealistic to expect all programmers to have knowledge on this topic, making compiler optimizations a potentially valuable tool, as they have the potential to improve software energy efficiency without requiring much effort from developers. At the end of this section, the research questions this work intends to give answer to are presented.

The method followed is described in Section 3.2, and each of the Subsections provide more insight on the choices that were made for each step. The method is quite simple, a set of programs

and a set optimizations flags were selected, then a framework was developed to automate compilation, execution and data collection. At last the experiment was performed and the results were analyzed resorting to statistical tests.

Chapter 4

Results Analysis and Discussion

In this chapter we present, analyse and discuss the results of the statistical tests performed on the energy consumption and execution time data collected during the experiment, Sections 4.1 and 4.2. After analysing each metric individually, the relation between the two is analysed in Section 4.3. Then, in Section 4.4, the key takeaways from the combined analysis of the results is provided.

4.1 Energy Consumption Analysis

In order to study the impact different compilation optimizations have in the benchmarks we considered, we started by systematically removing the outliers and plotting the resulting data in box plots. While in this section we do not show all plots, the complete set can be found in Section A.1.1.

In Figure 4.1, three box plots are presented that contain the measurements of energy consumption for three different benchmarks: `parfib`, `fasta` and `constraints`, when compiled without optimizations, `-O0`, and with the `-fcmm-sink`. We can observe that, while `parfib`'s energy consumption dropped considerably, `fasta`'s increased a large amount, and `constraint`'s appears to not vary significantly.

Figure 4.2, presents the energy consumption measurements in the same fashion as Figure 4.1, but for the `-fcase-folding` flag. With this flag, `fasta`'s energy consumption increased even more considerably, it consumed around 180 joules when compiled with no optimizations and over 400 joules when optimized. There were no noticeable differences in the other benchmarks' consumption, apart from a smaller variation of the measurements for `constraints`.

In Figure 4.3, we see that the three benchmarks recorded an increase in their energy consumption when compiled with `-floopification`. Once again, `fasta` had the most significant increase. While `parfib`'s and `constraints`' increased less than 10 joules, `fasta` went from consuming around 180 joules to roughly 280 joules.

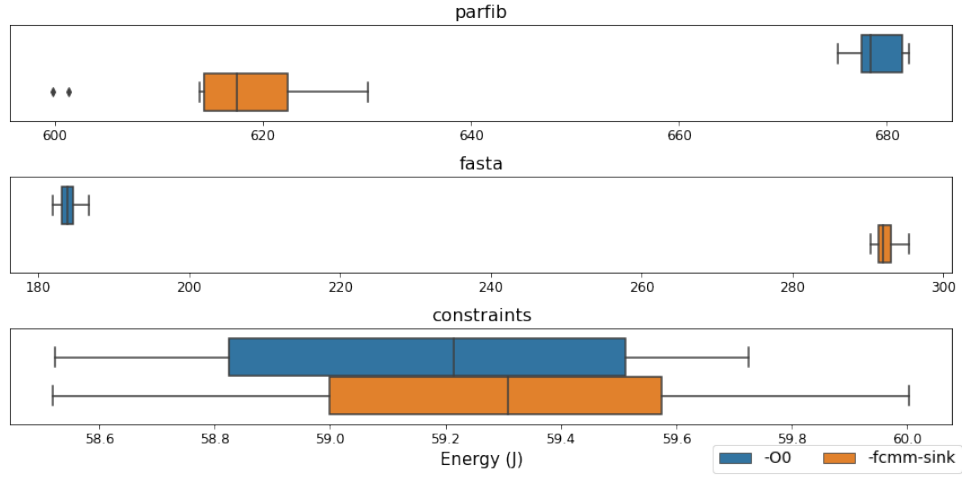


Figure 4.1: Energy consumption of `parfib`, `fasta` and `constraints`, when compiled with `-O0` and `-fcmm-sink`

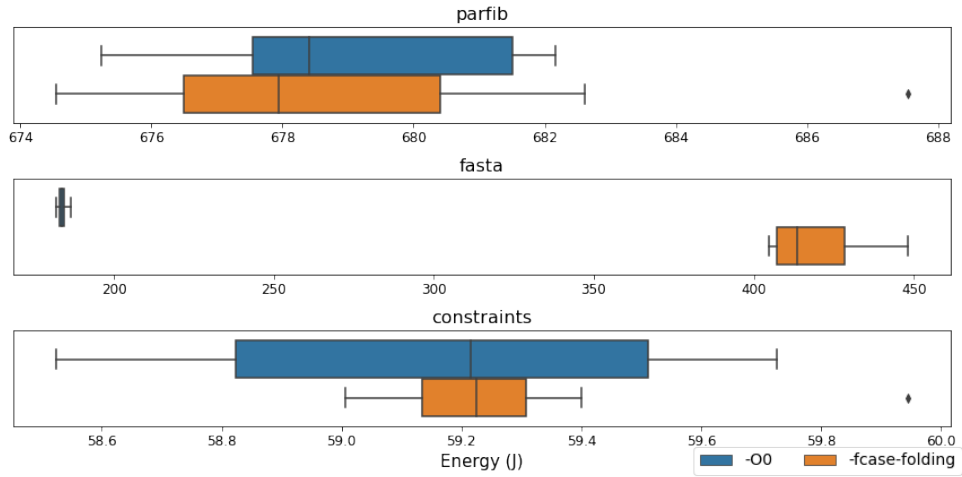


Figure 4.2: Energy consumption of `parfib`, `fasta` and `constraints`, when compiled with `-O0` and `-fcase-folding`

With this preliminary analysis of the charts we can start to formulate some hypothesis. Taking into account `parfib`'s and `constraints`' consumption, it appears that energy efficiency is influenced in different ways by different optimization flags. However, `fasta` appears to be an exception, its consumption increases significantly when compiled with any of the selected optimization flags, which can be observed in Section A.1.1. Another interesting observation is that the same flag influences programs' energy consumption in different ways. Even though for the three previously analysed benchmarks the `-floopification` flag increased consumption, this is not the case for all of them, as can be seen in Figure 4.4 and Figure A.17.

Performing statistical tests allows for a more in depth and comprehensive analysis of the results, thus proving or disproving the previously formulated hypothesis. In order to verify that

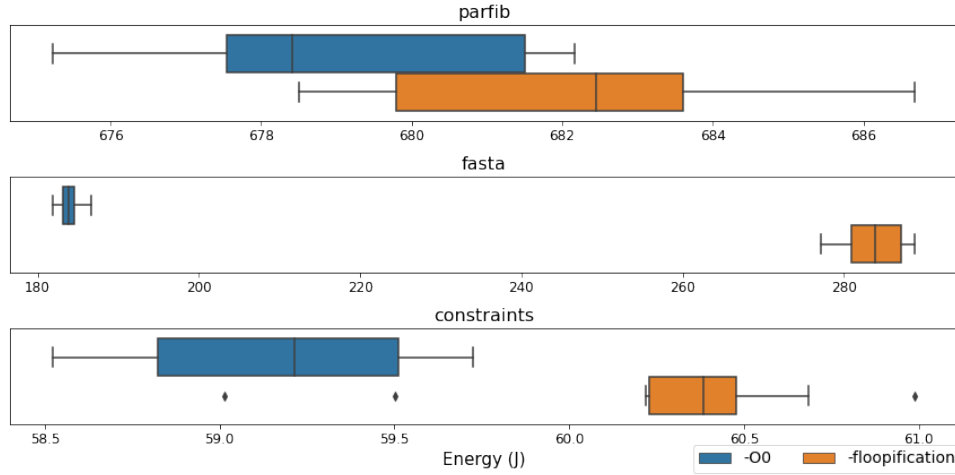


Figure 4.3: Energy consumption of `parfib`, `fasta` and `constraints`, when compiled with `-O0` and `-floopification`

the data collected for a benchmark compiled with and without optimizations was comparable, the ANOVA statistical test, which indicates whether or not two groups have a statistically significant difference, was performed. The results of this test are summarized by optimization flag in Table 4.1. The first column, ($p < 0.05$), presents the number of executables that have a statistically significant difference in energy consumption compared to their non-optimized version, while the second column, ($p > 0.05$), presents the count of executables that did not have a significant difference. The last column, labeled as "Excluded", shows the number of benchmarks which did not respect all of the assumptions of ANOVA, and are not considered in the rest of the energy consumption analysis. An extensive list that contains all the benchmarks that go into each of the aforementioned categories is presented in Table A.1.

The results we obtained allowed us to conclude that 43.59% of the optimized benchmarks have significantly different consumptions than their non-optimized versions, while 39.63% did not see a significant difference. The remaining 16.78% of the benchmark-flag combinations did not fulfill the requirements, the assumptions of ANOVA, and are therefore not considered.

Even though the data points to the fact that the optimizations have impact on energy consumption, we do not yet know whether this impact results in an increase or decrease in energy efficiency. To analyse this, the effect sizes were calculated for each benchmark-flag combination. This was accomplished using Hedges' g , whose results can be interpreted in the following way: a positive effect size translates to an increase in energy consumption by the optimized program, and a negative effect size to a decrease in consumption. The absolute value of the effect size indicates that the groups differ by that number of standard deviations. As a rule of thumb, absolute effect sizes of around 0.2 are considered small, 0.5 medium and 0.8 and above are large. However, as can be verified in Table A.2, none of the calculated effect sizes have an absolute value lower than 0.8.

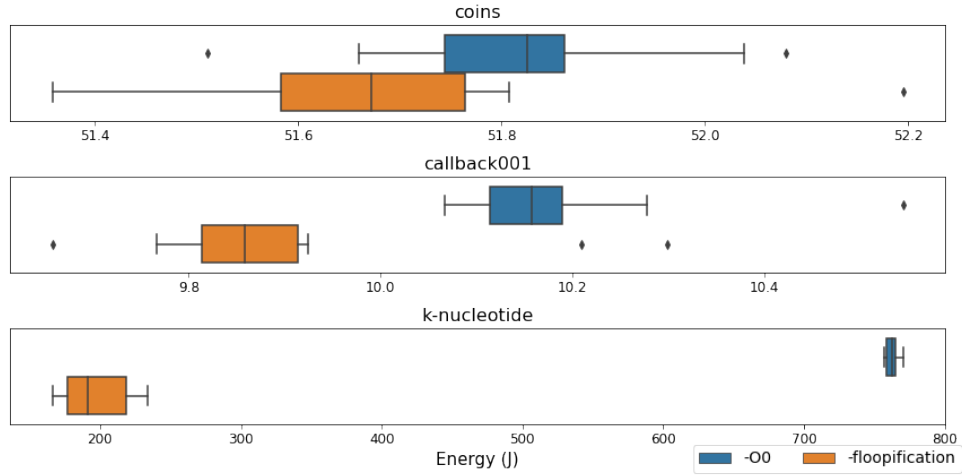


Figure 4.4: Energy consumption of `coins`, `callback001` and `k-nucleotide`, when compiled with `-O0` and `-floopification`

Table 4.2 shows the number of executables which recorded an increase, no statistically significant difference and a decrease in energy consumption, respectively labeled with $\uparrow$, $\simeq$ and $\downarrow$, by optimization flag, and the corresponding percentages. Analysing the data in the table we are able to observe the impact that each optimization flag had on all benchmarks. One of the most interesting cases is `-fstg-lift-lams`, for which the number of programs that increased, decreased and did not have a statistically significant difference in energy consumption was exactly the same. The only other flag that approximates this behaviour is `-fcmm-elim-common-blocks`, but, like the majority of the flags, the number of benchmarks that did not have a statistically significant difference in consumption is larger than the ones which did either increase or decrease their consumption. According to this data, the flags that had a significant impact on the largest and smallest number of benchmarks were `-fblock-layout-cfg`(92%) and `-fcase-merge`(28.57%), respectively.

Focusing now on how energy consumption was impacted, for 23 out of the 26 flags that were considered in this study, the number of benchmarks which saw an increase in their energy consumption was greater than those which saw a decrease. `-fblock-layout-cfg` was the flag that increased the consumption of the largest number of benchmarks, and, while `-fcall-arity` has the smallest ratio of increase, 17.86%, `-fcmm-sink` also increased the consumption of only five benchmarks. When it comes to decreasing the energy consumption, we verify that this flag, `-fcmm-sink`, improved the energy efficiency of the most benchmarks, with a ratio of 61.54%. However, most of the flags had a quite low ratio of improvement when compared to this, the second best being `-fstg-lift-lams`. Most of them have ratios lower than 20%, the smallest one being `-ffloat-in's`, which only decreased the consumption of one benchmark, resulting in a ratio of 3.85%.

Previously it was mentioned that `fasta` consumed more energy when compiled with any of the optimization flags considered in this study. Table 4.3 presents the effect sizes of this benchmark for each flag and it is possible to see that all of these values are very large. The smallest effect size is 18.60, and they go all the way up to 85.12. These account for some of the largest values of the

Flag	$\rho < 0.05$	$\rho > 0.05$	<i>Excluded</i>
-fcase-merge	8	20	5
-fcase-folding	9	18	6
-fcall-arity	9	19	5
-fexitification	13	14	6
-fcmm-elim-common-blocks	16	11	6
-fcmm-sink	21	5	7
-fcpr-anal	13	16	4
-fcse	16	12	5
-fstg-cse	20	7	6
-fdmd-tx-dict-sel	14	15	4
-fdo-eta-reduction	12	15	6
-fdo-lambda-eta-expansion	13	15	5
-ffloat-in	14	12	7
-ffull-laziness	15	11	7
-fignore-asserts	14	14	5
-fliberate-case	11	17	5
-floopification	19	8	6
-flvm-pass-vectors-in-regs	18	10	5
-fspec-constr	16	11	6
-fspecialise	11	15	7
-fcross-module-specialise	12	16	5
-fsolve-constant-dicts	13	13	7
-fstictness	13	16	4
-funbox-small-strict-fields	11	18	4
-fblock-layout-cfg	23	2	8
-fstg-lift-lams	20	10	3
Total	374	340	144

Table 4.1: Number of optimized executables that have statistically significant differences compared to their non-optimized versions, that do not, and that did not respect the assumptions of ANOVA, by optimization flag

entire experiment.

Looking at Table 4.4, which follows the same structure of Table 4.2, we can identify one more case similar to *fasta*, *awards*. Interestingly enough, there is no benchmark that presents the opposite behaviour, reducing energy consumption for all flags. The closest one is *callback001*, which saw its consumption reduced by 91.67% of the considered flags that respected the assumptions of ANOVA. However, if we look at the effect size values, presented in Table A.2, they are significantly smaller than both *fasta*'s and *awards*'. Other benchmarks, such as *callback002* and *anna*, were not significantly affected by most of the flags.

Flag	Energy					
	↑	≈	↓	% ↑	% ≈	% ↓
-fcase-merge	6	20	2	21.43	71.43	7.14
-fcase-folding	6	18	3	22.22	66.67	11.11
-fcall-arity	5	19	4	17.86	67.86	14.29
-fexitification	9	14	4	33.33	51.85	14.81
-fcmm-elim-common-blocks	8	11	8	29.63	40.74	29.63
-fcmm-sink	5	5	16	19.23	19.23	61.54
-fcpr-anal	8	16	5	27.59	55.17	17.24
-fcse	10	12	6	35.71	42.86	21.43
-fstg-cse	12	7	8	44.44	25.93	29.63
-fdmd-tx-dict-sel	8	15	6	27.59	51.72	20.69
-fdo-eta-reduction	7	15	5	25.93	55.56	18.52
-fdo-lambda-eta-expansion	11	15	2	39.29	53.57	7.14
-ffloat-in	13	12	1	50.00	46.15	3.85
-ffull-laziness	12	11	3	46.15	42.31	11.54
-fignore-asserts	12	14	2	42.86	50.00	7.14
-fliberate-case	9	17	2	32.14	60.71	7.14
-floopification	12	8	7	44.44	29.63	25.93
-fllvm-pass-vectors-in-regs	15	10	3	53.57	35.71	10.71
-fspec-constr	13	11	3	48.15	40.74	11.11
-fspecialise	8	15	3	30.77	57.69	11.54
-fcross-module-specialise	10	16	2	35.71	57.14	7.14
-fsolve-constant-dicts	11	13	2	42.31	50.00	7.69
-fstrictness	11	16	2	37.93	55.17	6.90
-funbox-small-strict-fields	9	18	2	31.03	62.07	6.90
-fblock-layout-cfg	16	2	7	64.00	8.00	28.00
-fstg-lift-lams	10	10	10	33.33	33.33	33.33

Table 4.2: Number of optimized executables in which there was an increase, no significant difference and a decrease in energy consumption for each optimization flag, and corresponding percentages

Flag	Effect Size	Flag	Effect Size
-fcase-merge	47.08	-ffull-laziness	33.66
-fcase-folding	20.83	-fignore-asserts	49.55
-fcall-arity	18.60	-fliberate-case	85.12
-fexitification	60.40	-floopification	31.88
-fcmm-elim-common-blocks	33.58	-fllvm-pass-vectors-in-regs	19.80
-fcmm-sink	68.54	-fspec-constr	33.39
-fcpr-anal	21.90	-fspecialise	21.25
-fcse	37.48	-fcross-module-specialise	47.93
-fstg-cse	30.97	-fsolve-constant-dicts	34.93
-fdmd-tx-dict-sel	32.75	-fstrictness	26.35
-fdo-eta-reduction	48.54	-funbox-small-strict-fields	22.49
-fdo-lambda-eta-expansion	24.27	-fblock-layout-cfg	58.29
-ffloat-in	54.36	-fstg-lift-lams	56.69

Table 4.3: Effect sizes of energy consumption of *fasta* for each optimization flag

Benchmark	$\uparrow$	$\simeq$	$\downarrow$	Energy		
				% $\uparrow$	% $\simeq$	% $\downarrow$
hash	11	9	6	42.31	34.62	23.08
gen_regexps	20	2	3	80.00	8.00	12.00
integrate	21	3	1	84.00	12.00	4.00
fasta	26	0	0	100.00	0.00	0.00
pidigits	5	15	4	20.83	62.50	16.67
ansi	20	1	2	86.96	4.35	8.70
awards	26	0	0	100.00	0.00	0.00
coins	2	13	10	8.00	52.00	40.00
compress2	2	14	10	7.69	53.85	38.46
binary-trees	20	5	1	76.92	19.23	3.85
parser	11	11	4	42.31	42.31	15.38
life	10	15	1	38.46	57.69	3.85
anna	2	21	3	7.69	80.77	11.54
callback001	0	2	22	0.00	8.33	91.67
power	1	16	8	4.00	64.00	32.00
cacheprof	4	13	9	15.38	50.00	34.62
real_cacheprof	14	12	0	53.85	46.15	0.00
gg	2	15	8	8.00	60.00	32.00
constraints	7	18	0	28.00	72.00	0.00
treejoin	8	16	0	33.33	66.67	0.00
parfib	8	12	3	34.78	52.17	13.04
blackscholes	2	18	4	8.33	75.00	16.67
bernouilli	17	7	1	68.00	28.00	4.00
callback002	0	23	2	0.00	92.00	8.00
chan	3	21	2	11.54	80.77	7.69
atom	2	17	7	7.69	65.38	26.92
cryptarithm1	8	15	2	32.00	60.00	8.00
queens	2	19	4	8.00	76.00	16.00
exp3_8	2	0	0	100.00	0.00	0.00
clausify	0	5	1	0.00	83.33	16.67

Table 4.4: Number of optimization flags that for each benchmark increased, maintained and decreased energy consumption and respective percentages

4.2 Execution Time Analysis

Similar to what was done in the previous section, we start our analysis by removing the outliers and plotting the resulting data in box plots. Not all of the plots are shown in this section, however the complete set can be found in Section A.2.1.

Figure 4.5, shows three box plots that contain the execution times for `parfib`, `fasta` and `constraints`, when compiled without optimizations, `-O0`, and with `-fcmm-sink`. We can observe that, while `parfib`'s execution time was reduced by around three seconds, `fasta`'s was increased a similar amount. However, `constraints`' appears to not vary significantly.

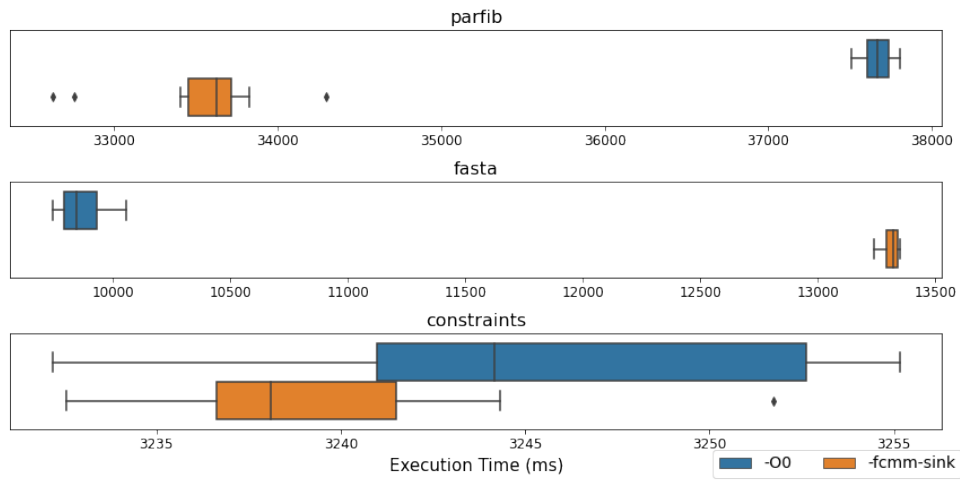


Figure 4.5: Execution time of `parfib`, `fasta` and `constraints`, when compiled with `-O0` and `-fcmm-sink`

Looking at Figure 4.6, which presents the execution time for the same benchmarks but now compiled with `-fcase-folding`, we immediately notice that `fasta` took much longer to finish executing when compiled with this flag, a difference of almost 40 seconds. But this is not reflected in neither `parfib` nor `constraints`, for which the differences in execution times is in the order of the tens of milliseconds.

When compiled with `-floopification`, `fasta` increases its execution time once again, as can be observed in Figure 4.7. With the same flag, `constraints`' execution time is also increased, although in a much smaller scale than `fasta`'s, and `parfib`'s decreases.

Just like energy consumption, execution time seems to be influenced by the optimization flags. For example, `-fcmm-sink` has different effects on each of the three benchmarks in Figure 4.5. So it appears that the same flag can increase, decrease or maintain the execution time of different benchmarks. Likewise, it seems to be possible that a benchmark is impacted in different ways by different flags.

In order to be sure that the data was comparable, we performed the ANOVA statistical test again, this time for the execution time data. Table 4.5 presents the number of benchmarks for which there is a statistically significant difference, column labeled $p < 0.05$, that did not, $p > 0.05$,

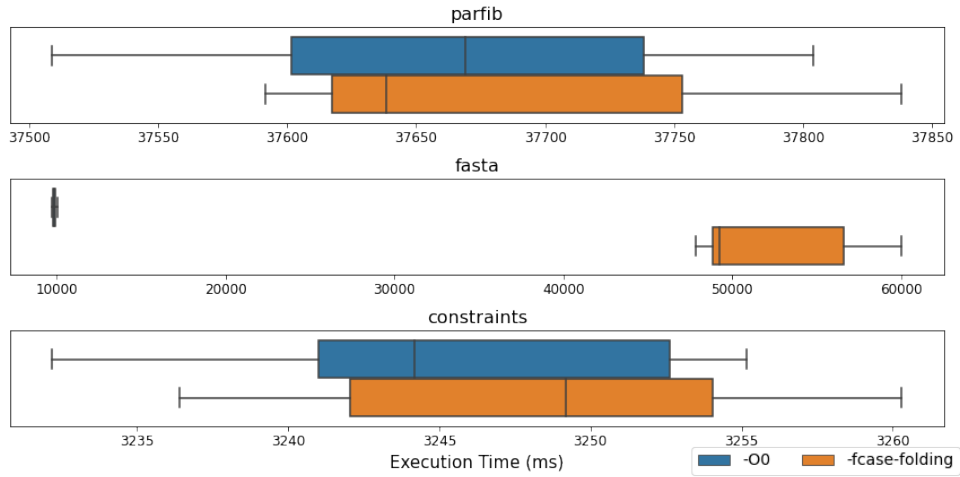


Figure 4.6: Execution time of `parfib`, `fasta` and `constraints`, when compiled with `-O0` and `-fcase-folding`

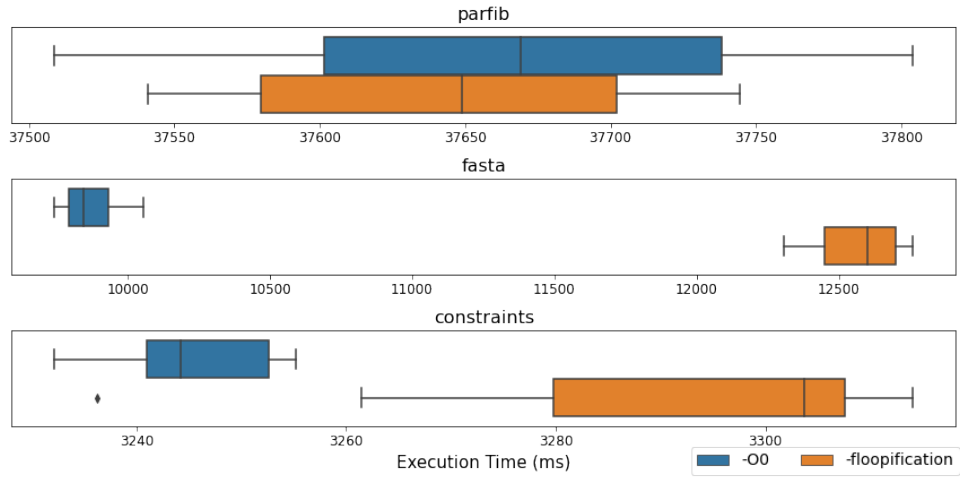


Figure 4.7: Execution time of `parfib`, `fasta` and `constraints`, when compiled with `-O0` and `-floopification`

and that did not respect the assumptions of ANOVA, last column, by optimization flag. The list of benchmarks that comprise each of the aforementioned categories is presented in Table A.3. Looking now at Table 4.5, we see that out of the 858 benchmark-flag combinations for which execution time data was collected, 260 (30.30%) had their execution time significantly increased or decreased, 365 (42.54%) did not have a significant difference in execution time, and the remaining 233 (27.16%) did not respect all of the assumptions of ANOVA.

The effect sizes of the execution time were calculated for each of the benchmark-flag combinations which were identified by ANOVA as having a statistically significant difference when compared to the non-optimized version of the benchmark. In this context, a positive effect size means that when compiled with an optimization flag the execution time increased, whereas a neg-

Flag	$\rho < 0.05$	$\rho > 0.05$	<i>Excluded</i>
-fcase-merge	3	25	5
-fcase-folding	2	19	12
-fcall-arity	7	17	9
-fexitification	8	18	7
-fcmm-elim-common-blocks	14	9	10
-fcmm-sink	22	3	8
-fcpr-anal	6	17	10
-fcse	7	15	11
-fstg-cse	13	8	12
-fdmd-tx-dict-sel	10	19	4
-fdo-eta-reduction	7	17	9
-fdo-lambda-eta-expansion	9	13	11
-ffloat-in	8	14	11
-ffull-laziness	11	12	10
-fignore-asserts	10	12	11
-fliberate-case	6	16	11
-floopification	17	10	6
-flvm-pass-vectors-in-regs	9	16	8
-fspec-constr	9	14	10
-fspecialise	9	19	5
-fcross-module-specialise	9	13	11
-fsolve-constant-dicts	12	11	10
-fstictness	7	17	9
-funbox-small-strict-fields	10	17	6
-fblock-layout-cfg	19	4	10
-fstg-lift-lams	16	10	7
Total	260	365	233

Table 4.5: Number of optimized executables that have statistically significant differences compared to their non-optimized versions, that do not, and that did not respect the assumptions of ANOVA, by optimization flag

ative effect size means that it decreased. Once again, non of the benchmark-flag combinations presented an effect size lower than 0.8, which can be verified in Table A.4.

Table 4.6 presents a count of the benchmarks in which there was an increase, no statistically significant difference and a decrease in execution time, respectively labeled with $\uparrow$, $\simeq$ and $\downarrow$, by optimization flag, and the corresponding percentages. With a quick analysis of the table, it is noticeable that both `-fcase-merge` and `-fcase-folding` did not reduce the execution time of any benchmark that was flagged by ANOVA as having a statistically significant difference. These two flags were responsible for the two highest ratios of benchmarks in which there was no statistically significant difference in execution time, `-fcase-merge` with 89.29% and `-fcase-folding` with 90.48%. Consequently, they present the smallest ratios when it comes to increasing execution times, only 10.71% of the benchmarks compiled with `-fcase-merge` and 9.52% of those compiled with `-fcase-folding` had their execution times extended. Another flag that yielded interesting

results was `-fcmm-sink`, which had the highest percentage of benchmarks that recorded a decrease in their execution times, 60.00%, and the lowest percentage of benchmarks for which there was no statistically significant difference, 12.00%. The flag that recorded the highest ratio of programs that increased their execution time was `-fblock-layout-cfg`, 39.13%.

Flag	Execution Time					
	↑	≈	↓	% ↑	% ≈	% ↓
<code>-fcase-merge</code>	3	25	0	10.71	89.29	0.00
<code>-fcase-folding</code>	2	19	0	9.52	90.48	0.00
<code>-fcall-arity</code>	5	17	2	20.83	70.83	8.33
<code>-fexitification</code>	4	18	4	15.38	69.23	15.38
<code>-fcmm-elim-common-blocks</code>	7	9	7	30.43	39.13	30.43
<code>-fcmm-sink</code>	7	3	15	28.00	12.00	60.00
<code>-fcpr-anal</code>	3	17	3	13.04	73.91	13.04
<code>-fcse</code>	4	15	3	18.18	68.18	13.64
<code>-fstg-cse</code>	6	8	7	28.57	38.10	33.33
<code>-fdmd-tx-dict-sel</code>	6	19	4	20.69	65.52	13.79
<code>-fdo-eta-reduction</code>	3	17	4	12.50	70.83	16.67
<code>-fdo-lambda-eta-expansion</code>	6	13	3	27.27	59.09	13.64
<code>-ffloat-in</code>	5	14	3	22.73	63.64	13.64
<code>-ffull-laziness</code>	6	12	5	26.09	52.17	21.74
<code>-fignore-asserts</code>	6	12	4	27.27	54.55	18.18
<code>-fliberate-case</code>	3	16	3	13.64	72.73	13.64
<code>-floopification</code>	9	10	8	33.33	37.04	29.63
<code>-flvm-pass-vectors-in-regs</code>	7	16	2	28.00	64.00	8.00
<code>-fspec-constr</code>	7	14	2	30.43	60.87	8.70
<code>-fspecialise</code>	7	19	2	25.00	67.86	7.14
<code>-fcross-module-specialise</code>	5	13	4	22.73	59.09	18.18
<code>-fsolve-constant-dicts</code>	8	11	4	34.78	47.83	17.39
<code>-fstrictness</code>	4	17	3	16.67	70.83	12.50
<code>-funbox-small-strict-fields</code>	6	17	4	22.22	62.96	14.81
<code>-fblock-layout-cfg</code>	9	4	10	39.13	17.39	43.48
<code>-fstg-lift-lams</code>	4	10	12	15.38	38.46	46.15

Table 4.6: Number of optimized executables in which there was an increase, no significant difference and a decrease in execution time for each optimization flag, and corresponding percentages

In order to have a better understanding of the impact on each of the benchmarks, Table 4.7 was created, which is structured similarly to Table 4.6, but presents the counts and percentages of flags that increased, decreased or had no statistically significant impact on each benchmark's execution time. With a quick glance, we verify that `fasta` and `awards` recorded an increase in their execution times for all flags that respected the assumptions of ANOVA. It is also noticeable that a considerable number of flags did not increase their execution time when compiled with any of the flags, for the majority of these benchmarks around 70% of the flags did not have a statistically significant impact on execution time, leaving very few flags that reduced execution time. However, both `gg` and `cacheprof` had their execution time reduced by 80.95% and 88.46%,

the highest percentages of the entire experiment.

Benchmark	Execution Time					
	↑	≈	↓	% ↑	% ≈	% ↓
fasta	22	0	0	100.00	0.00	0.00
awards	23	0	0	100.00	0.00	0.00
cryptarithm1	21	3	1	84.00	12.00	4.00
ansi	2	12	3	11.76	70.59	17.65
gg	0	4	17	0.00	19.05	80.95
parser	5	14	3	22.73	63.64	13.64
queens	9	9	4	40.91	40.91	18.18
cacheprof	0	3	23	0.00	11.54	88.46
gen_regexps	19	3	2	79.17	12.50	8.33
anna	1	12	13	3.85	46.15	50.00
compress2	0	18	3	0.00	85.71	14.29
constraints	4	17	1	18.18	77.27	4.55
hash	9	11	5	36.00	44.00	20.00
exp3_8	4	12	0	25.00	75.00	0.00
integrate	3	20	2	12.00	80.00	8.00
callback001	1	10	2	7.69	76.92	15.38
callback002	3	20	0	13.04	86.96	0.00
power	0	17	7	0.00	70.83	29.17
parfib	0	14	7	0.00	66.67	33.33
blackscholes	0	19	6	0.00	76.00	24.00
bernouilli	5	14	1	25.00	70.00	5.00
real_cacheprof	1	17	5	4.35	73.91	21.74
binary-trees	1	13	1	6.67	86.67	6.67
pidigits	0	13	3	0.00	81.25	18.75
chan	1	18	2	4.76	85.71	9.52
atom	2	13	3	11.11	72.22	16.67
life	2	19	1	9.09	86.36	4.55
coins	4	17	2	17.39	73.91	8.70
clausify	0	7	1	0.00	87.50	12.50

Table 4.7: Number of optimization flags that for each benchmark increased, maintained and decreased execution time and respective percentages

4.3 Relating Execution Time and Energy Consumption

Now that an individual analysis of both energy consumption and execution time were performed, it is time to try to understand how these metrics relate. First, we plotted the energy consumption and execution time pairs collected for all flags and benchmarks in a scatter plot, presented in Figure 4.8. Immediately, it is noticeable that there is an odd cluster of points that had an energy consumption between 400 and 500 joules with execution times between 45 and 70 seconds. This cluster only

contains 30 of the 8904 samples plotted and they all belong to *fasta* when compiled with `-fcase` `-merge`, `-fcase-folding` and `-fcall-arity`. This is a rather unexpected and interesting, however we leave understanding why this happens for future research.

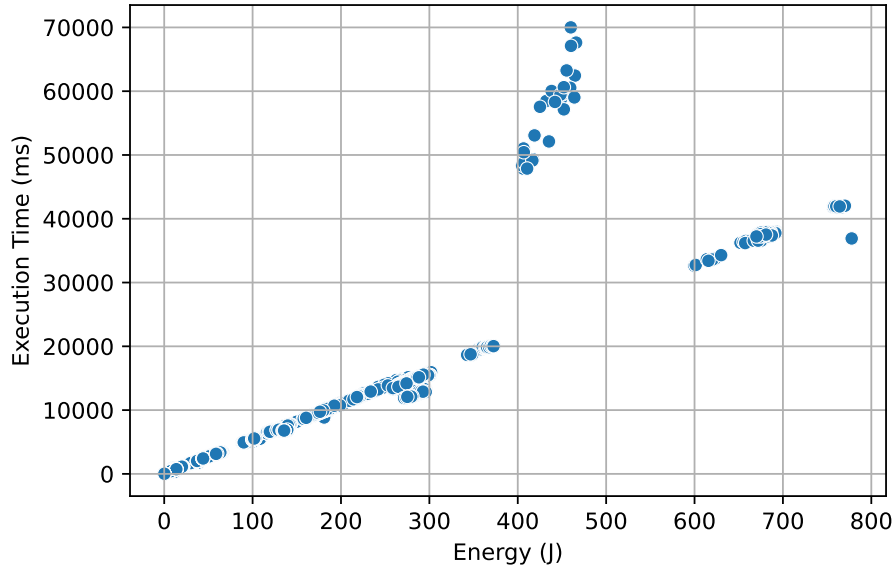


Figure 4.8: Scatter plot of energy consumption and execution time of all benchmarks compiled with all optimization flags

To measure the strength and direction of the relationship between energy consumption and execution time, the Spearman's correlation coefficient was calculated using all of samples. The correlation coefficient can range from -1 to +1, the closer the absolute value is to 1 the stronger the relation, and the sign of the coefficient indicates whether the relation is a positive or negative monotonic relation. In a positive monotonic relation, as one variable increases so does the other, while, in a negative monotonic relation, as one variable increases, the other tends to decrease. The ρ value represents the probability that the correlation between time and energy occurred by chance. The results of Spearman's were a correlation coefficient of 0.997347 and a ρ value of 0.0, meaning that as energy consumption increases, so does execution time, and that this correlation did not occur by chance.

Performing a similar analysis of the correlation, but this time for each of the benchmark-flag combinations, also yielded some interesting results, which can be visualized in Figure 4.9. These results were acquired by calculating Spearman's correlation coefficient for each combination and considering only ρ values lower than 0.05. In this context the sign of the coefficient tells us the direction of the relation, a positive coefficient means that as energy consumption increases so does execution time, while a negative coefficient means that as energy consumption increases, the execution time decreases. The absolute value of the coefficient translates to the strength of the relationship, for example, an absolute value of 1 indicates the data is perfectly linear.

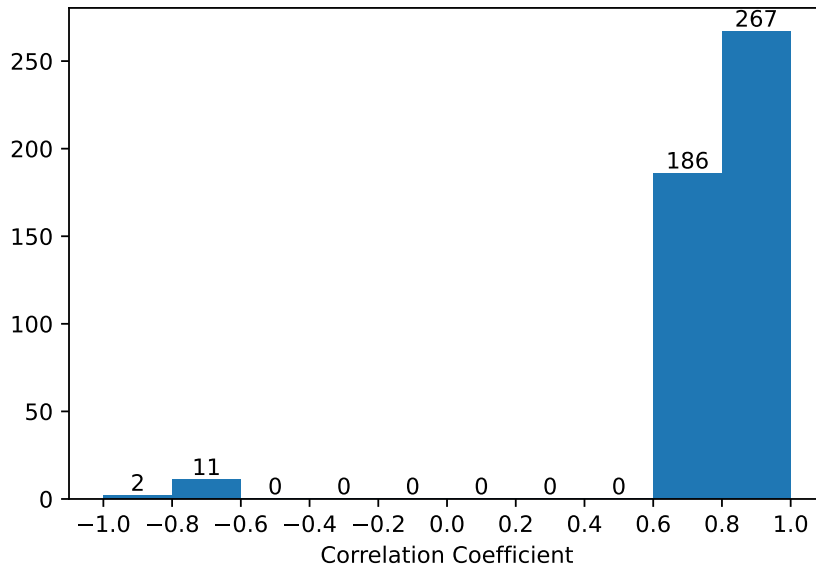


Figure 4.9: Histogram of Spearman's correlation coefficient for each benchmark-flag combination with $\rho < 0.05$

A first glance at Figure 4.9 reveals that for most of the benchmark-flag combinations as energy consumption increases, so does execution time. However, for 13 of them this is not the case. For these combinations, an increase in energy consumption translates to a decrease of the execution time. The two lowest coefficients were both for the `banner` benchmark when compiled with `-fcmm` `-elim-common-blocks` (-0.90303) and `-fstg-lift-lams` (-0.80606), for which the scatter plots of the samples can be observed in Figure 4.10 and 4.11, respectively.

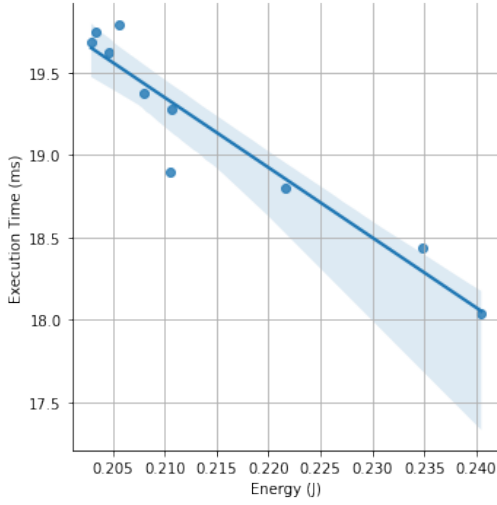


Figure 4.10: Scatter plot of energy consumption and execution time of `banner` when compiled with `-fcmm-elim-common-blocks`

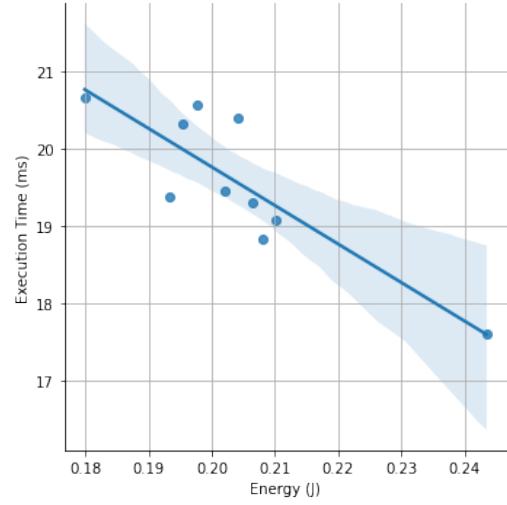


Figure 4.11: Scatter plot of energy consumption and execution time of `banner` when compiled with `-fstg-lift-lams`

4.4 Main Takeaways

Throughout the statistical analysis of the data and the discussion of the results, many observations were made regarding the influence optimization flags have on energy consumption and execution time, Section 4.1 and 4.2, and the existence of a relation between both of these metrics, Section 4.3. Following are the main takeaways that resulted from this analysis.

- We are not able to predict how an optimization flag will influence a benchmark's energy consumption or execution time;
- Most of the considered flags increased energy consumption for more benchmarks than the ones they reduced; similarly for execution time;
- The flag that improved the performance of most benchmarks, both in terms of energy consumption and execution time, was `-fcmm-sink`;
- `-fblock-layout-cfg` was responsible for increasing both execution time and energy consumption of the largest number of benchmarks;
- `fasta`'s and `awards`' execution time and energy consumption were increased by every single flag
- Energy consumption and execution time are strongly related, and as one increases, so does the other. Most benchmark-flag combinations follow this trend, however, for 13 of them as energy consumption increases, execution time decreases.

4.5 Summary

In Sections 4.1 and 4.2, the plotted data for energy consumption and execution time is analysed and some preliminary hypothesis are formulated. These hypothesis are confirmed by the results of the statistical tests, which, as was expected, allow us to verify that no flag influences all benchmarks in the same way. As such, it is not possible to predict how a benchmark's energy consumption or execution time will be influenced when compiled with an optimization flag. However, it appears that some flags are more likely to influence these metrics in certain ways than others.

The analysis of the correlation between energy consumption and execution time also yielded interesting results, presented and discussed in Section 4.3. According to the collected data, these two metrics are strongly related and as one increases, so does the other. Surprisingly, it appears that for some benchmark-flag combinations as energy consumption increases, execution time decreases.

Chapter 5

Conclusions

The goal of this dissertation was to understand the impact that a set of GHC’s optimizations have on a set of Haskell benchmarks. In Section 5.1, we provide answers to the research questions, based on the data collected for the chosen flags and benchmarks. The threats to the validity of this work, and its limitations are presented in Section 5.2. Finally, Section 5.4, concludes this work and discusses the obtained results and the drawn conclusions.

5.1 Research Findings

This work set out to answer three research questions regarding the impact of compiler optimizations on a program’s energy consumption and execution time, in the context of Haskell and GHC, and the relation between those two variables. After collecting and analysing the results of the experiment we now have the required knowledge to provide proper answers to those questions.

RQ₁ How do compiler optimizations influence the energy efficiency of programs?

From what we could observe in the energy consumption data of the considered set of flags and benchmarks, different flags influence benchmarks in different ways. This means that a program’s consumption may not vary significantly, may increase or decrease. And, even though we are not able to predict how a flag will influence the energy efficiency of a program, the results lead us to believe that some flags, like `-fcmm-sink` have a higher chance of improving the energy efficiency of a program.

RQ₂ How do compiler optimizations influence the execution time of programs?

Just like energy consumption, the execution time is influenced by the optimization flags used in compilation. Once again, it is not possible to predict how a flag will influence a program, but, from the impact that the different flags had on the chosen set of benchmarks,

it appears that some flags have a greater potential to improve this metric, such as `-fcmm-sink`, while others will most likely not have a significant impact, like `-fcase-merge` and `-fcase-folding`.

RQ₃ How are time and energy consumption related?

From the samples of both energy consumption and execution time collected for the chosen sets of benchmarks and flags, we can confidently assert that there is a strong relation between these two metrics. As energy consumption increases, so does execution time. However, when we calculated the correlation coefficients for each benchmark-flag combination, some indicated that as energy consumption increased, execution time decreased.

5.2 Possible Limitations

Even though all the work was done in order to minimize the number of limitations and threats to validity, there are still some that could not be mitigated. In this work the focus is on the Haskell programming language, for the many different reasons that have already been stated in Section 2.1, because of this it is not possible to claim that the results of this study are verifiable across programming languages.

Only using one system to perform the experiment helps with the consistency of the results, as hardware influences energy consumption. But, even though the chosen system is representative of commonly used hardware, replicating the experiment with other hardware may yield different results.

Similarly to hardware, if a different set of compiler flags, or even a different compiler was used to realize this study, it is not possible to guarantee that the results and, therefore, the conclusions drawn from them would be the same.

An aspect that has been proven to influence energy consumption is the size of the data a program processes [14, 21]. In the performed experiment, the data that the programs handled was never modified in order to keep the results consistent. However, if the experiment was to be replicated with different amounts of data it is possible that the results would be different.

5.3 Future Work

With this study we were able to see how the chosen set of GHC optimizations influenced the chosen set of Haskell benchmarks, both in terms of energy consumption and execution time. In the future, it would be interesting to extend this work by studying the influence of other optimization flags, or even perform a similar study for other programming languages and compilers.

During the analysis of the results, we identified two benchmarks that recorded an increase in both execution time and energy consumption when compiled with all of the considered flags,

`fasta` and `awards`. Understanding why this happened could be an intriguing topic to explore in future work.

While analysing the correlation between energy consumption and execution time, we noticed that a cluster of samples did not follow the trend of the other samples, and had a much longer execution time than was expected. All of these samples belonged to the `fasta` benchmark, when compiled with `-fcase-merge`, `-fcase-folding` and `-fcall-arity`. Making sense of this behaviour could provide more insight into the relation between energy consumption and execution time.

Another interesting observation that resulted from the analysis of the correlation, were the benchmark-flag combinations for which the relation between the two metrics was not a positive monotonic relation, but instead a negative one. This means that, instead of execution time increasing when energy consumption increases, it decreases. In this work we do not try to fathom the reason behind this, but it would be interesting to understand why this happens.

5.4 Conclusions

This dissertation intends to understand how optimizations applied by GHC impact Haskell programs' energy efficiency, and how this impact can be leveraged to reduce software's energy consumption.

In order to achieve this goal, we chose a set of Haskell benchmarks and a set of GHC optimization flags, compiled each benchmark without optimizations and with each of the flags, and collected energy consumption and execution time data. Through the analysis of the results we were able to conclude that, while we are not able to predict how a flag will influence a program, some flags are more likely to increase, decrease or maintain the performance of a program when compared to its non optimized version, both in terms of energy consumption and execution time. We were also able to conclude that, for the considered sets of benchmarks and flags, there is a strong relationship between energy and time. As energy consumption increases, so does execution time.

This work aims to help improve the energy efficiency of software by providing new resources and knowledge to developers when it comes to the impact that compiler optimizations have on this non-functional requirement.

References

- [1] Sarah Abdulsalam, Donna Lakowski, Qijun Gu, Tongdan Jin, and Ziliang Zong. Program energy efficiency: The impact of language, compiler and implementation choices. In *International Green Computing Conference*, pages 1–6, 2014.
- [2] Anders Andrae. New perspectives on internet electricity use in 2030. *Engineering and Applied Science Letters*, 3:19–31, Jun 2020.
- [3] Anders S. G. Andrae and Tomas Edler. On global electricity usage of communication technology: Trends to 2030. *Challenges*, 6(1):117–157, 2015.
- [4] Sergey Bushnyak. Companies that use haskell in production. Available at: <https://haskellcosm.com/>, May 2021.
- [5] Marco Couto, Rui Pereira, Francisco Ribeiro, Rui Rua, and João Saraiva. Towards a green ranking for programming languages. In *Proceedings of the 21st Brazilian Symposium on Programming Languages, SBLP 2017*, New York, NY, USA, 2017. Association for Computing Machinery.
- [6] Thanh Do, Suhil Rawshdeh, and Weisong Shi. ptop: A process-level power profiling tool. Jan 2009.
- [7] Paul D. Ellis. *The Essential Guide to Effect Sizes: Statistical Power, Meta-Analysis, and the Interpretation of Research Results*. Cambridge University Press, 2010.
- [8] Taylor Fausak. 2021 state of haskell survey results. Available at: <https://taylor.fausak.me/2021/11/16/haskell-survey-results/>, Nov 2021.
- [9] J. Flinn and M. Satyanarayanan. Powerscope: a tool for profiling the energy usage of mobile applications. In *Proceedings WMCSA’99. Second IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–10, 1999.
- [10] Rong Ge, Xizhou Feng, Shuaiwen Song, Hung-Ching Chang, Dong Li, and Kirk W. Cameron. Powerpack: Energy profiling and analysis of high-performance systems and applications. *IEEE Transactions on Parallel and Distributed Systems*, 21(5):658–671, 2010.
- [11] Marcus Hähnel, Björn Döbel, Marcus Völpl, and Hermann Härtig. Measuring energy consumption for short code paths using rapl. *SIGMETRICS Perform. Eval. Rev.*, 40(3):13–17, Jan 2012.
- [12] HaskellWiki. Haskell in industry. Available at: https://wiki.haskell.org/index.php?title=Haskell_in_industry&oldid=63468, 2020.

- [13] J. Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, Jan 1989.
- [14] Steve Kerrison and Kerstin Eder. Energy modeling of software for a hardware multithreaded embedded microprocessor. *ACM Trans. Embed. Comput. Syst.*, 14(3), Apr 2015.
- [15] Patrick Kurp. Green computing. *Commun. ACM*, 51(10):11–13, Oct 2008.
- [16] Javier Mancebo, Coral Calero, Felix Garcia, M^a Angeles Moraga, and Ignacio Garcia-Rodriguez de Guzman. Feetings: Framework for energy efficiency testing to improve environmental goal of the software. *Sustainable Computing: Informatics and Systems*, 30:100558, 2021.
- [17] Simon Marlow. Fighting spam with haskell. Available at: <https://engineering.fb.com/2015/06/26/security/fighting-spam-with-haskell/>, Jun 2018.
- [18] Simon Marlow and Simon Peyton Jones. *The Glasgow Haskell Compiler*. Lulu, the architecture of open source applications, volume 2 edition, January 2012.
- [19] R.L. Mason, R.F. Gunst, and J.L. Hess. *Statistical Design and Analysis of Experiments: With Applications to Engineering and Science*. Wiley Series in Probability and Statistics. Wiley, 2003.
- [20] Gilberto Melfe, Alcides Fonseca, and João Paulo Fernandes. Helping developers write energy efficient haskell through a data-structure evaluation. In *Proceedings of the 6th International Workshop on Green and Sustainable Software*, GREENS '18, page 9–15, New York, NY, USA, 2018. Association for Computing Machinery.
- [21] James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. Data dependent energy modeling for worst case energy consumption analysis. *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems*, Jun 2017.
- [22] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E. Hassan. What do programmers know about the energy consumption of software? Jul 2015.
- [23] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 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, page 256–267, New York, NY, USA, 2017. Association for Computing Machinery.
- [24] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609, 2021.
- [25] Rui Pereira, Marco Couto, João Saraiva, Jácome Cunha, and João Paulo Fernandes. 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, page 15–21, New York, NY, USA, 2016. Association for Computing Machinery.
- [26] Gustavo Pinto, Fernando Castor, and Yu David Liu. Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 22–31, New York, NY, USA, 2014. Association for Computing Machinery.

- [27] Hannah Ritchie and Max Roser. Energy. *Our World in Data*, 2020. Available at: <https://ourworldindata.org/energy>.
- [28] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, 32(2):20–27, 2012.
- [29] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.
- [30] Yewan Wang, D. Nortershauser, Stephane Masson, and Jean-Marc Menaud. Potential effects on server power metering and modeling. *Wireless Networks*, 11 2018.
- [31] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

Appendix A

Charts and Statistical Tests

A.1 Energy Consumption

A.1.1 Charts

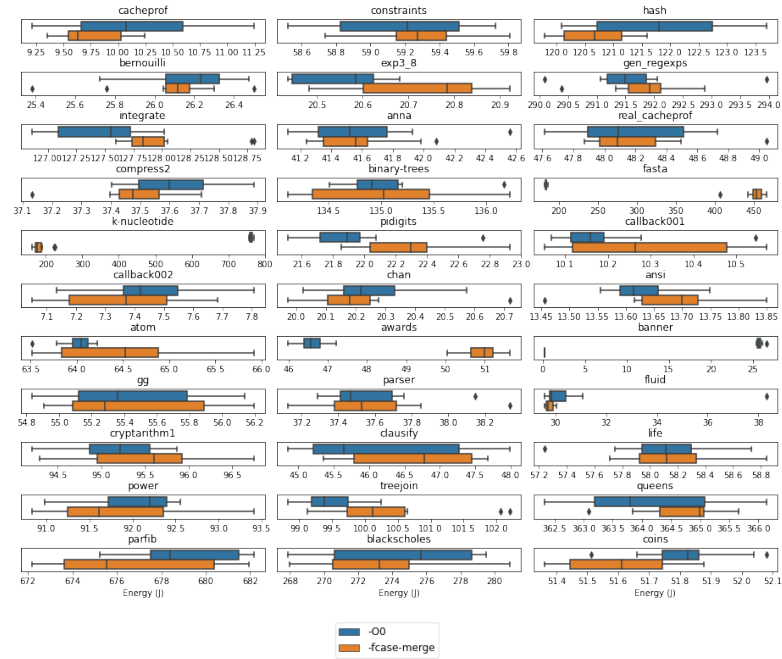
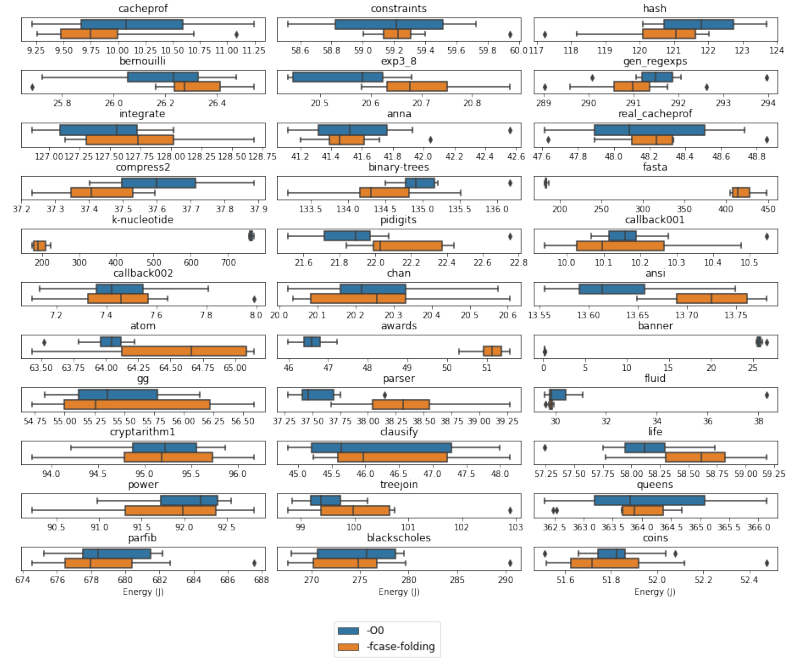
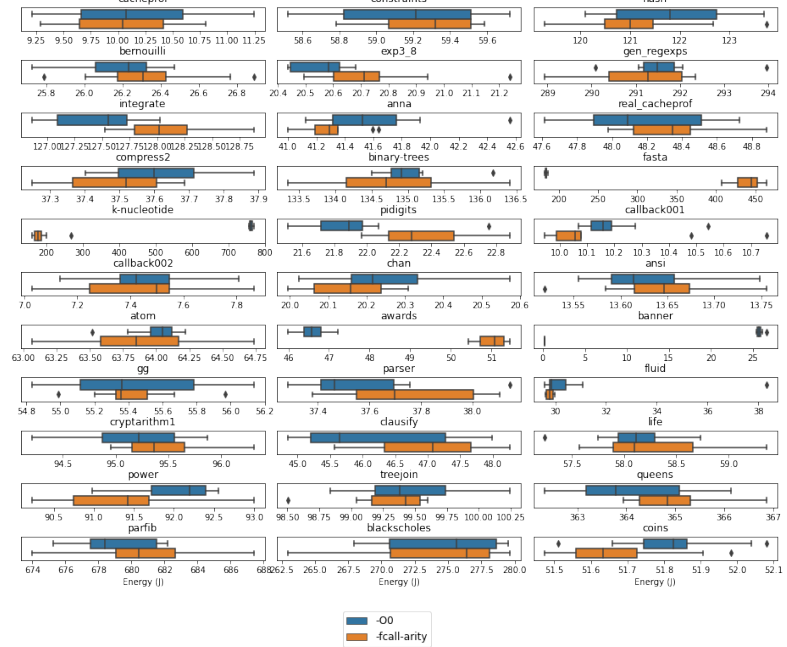
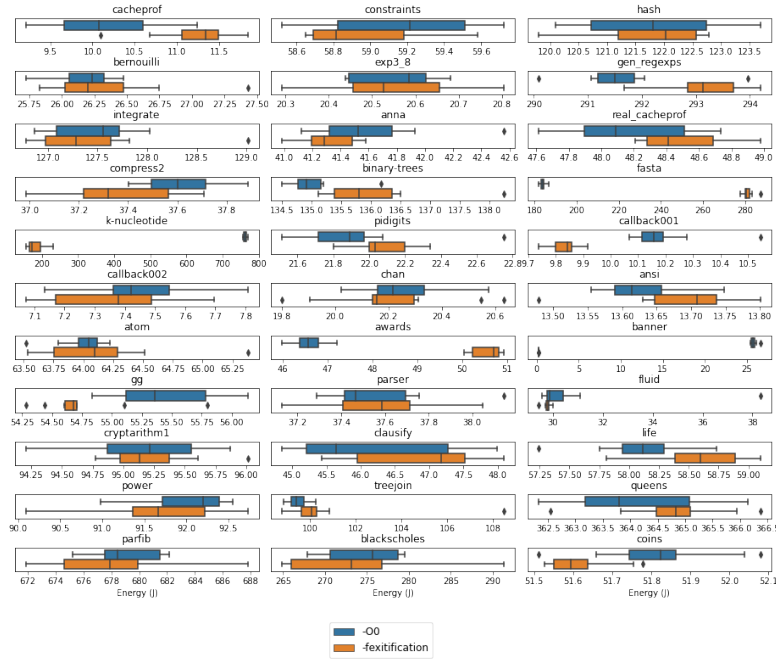
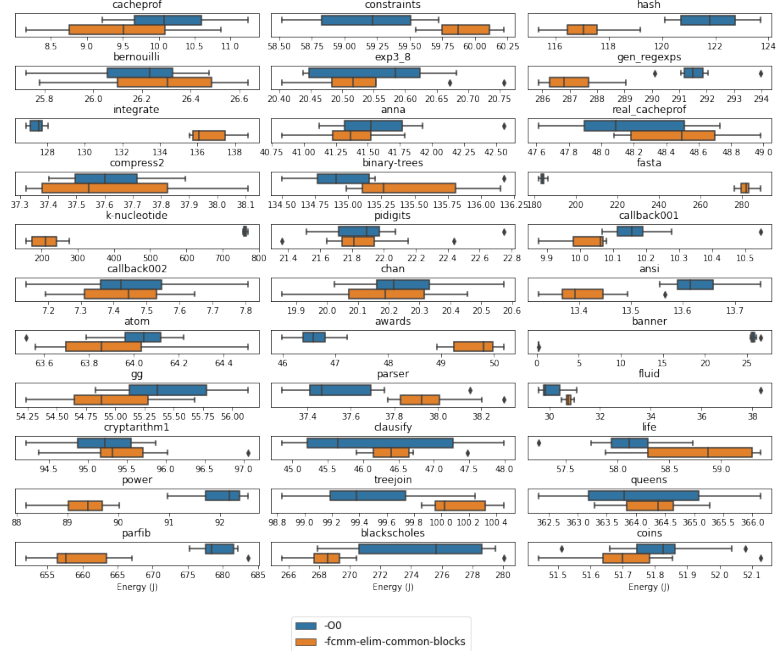
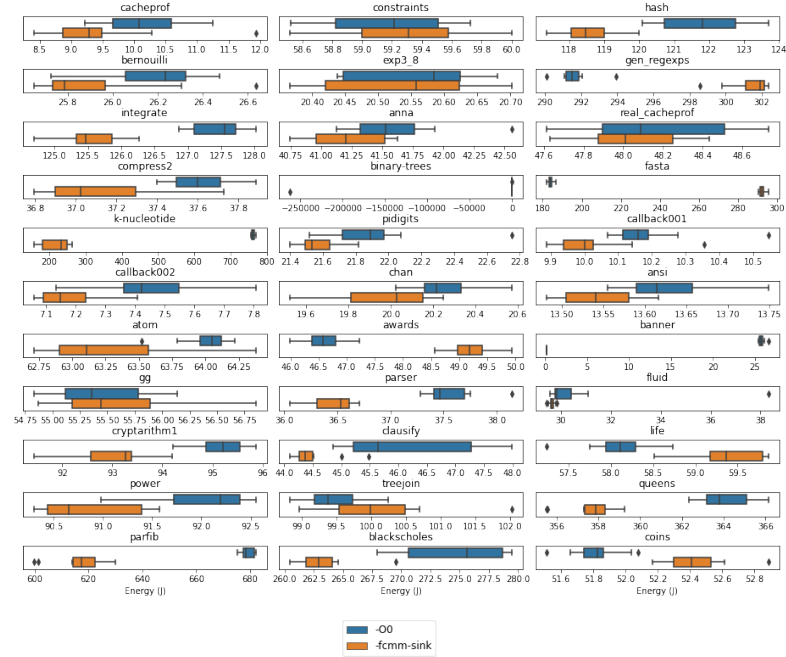
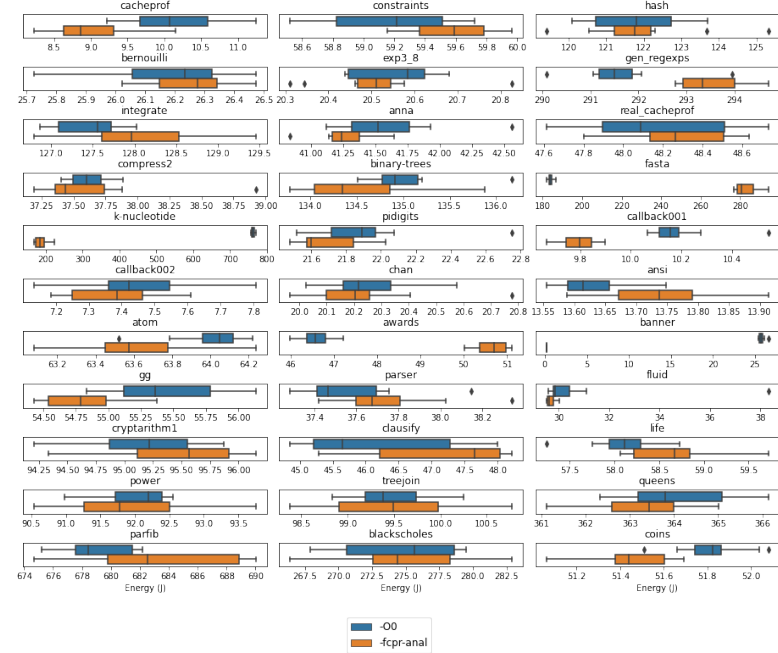
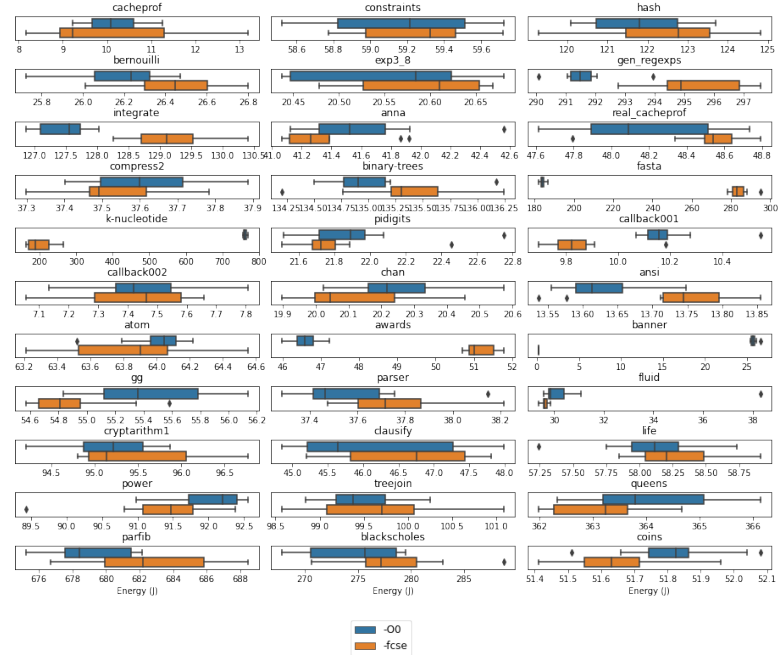
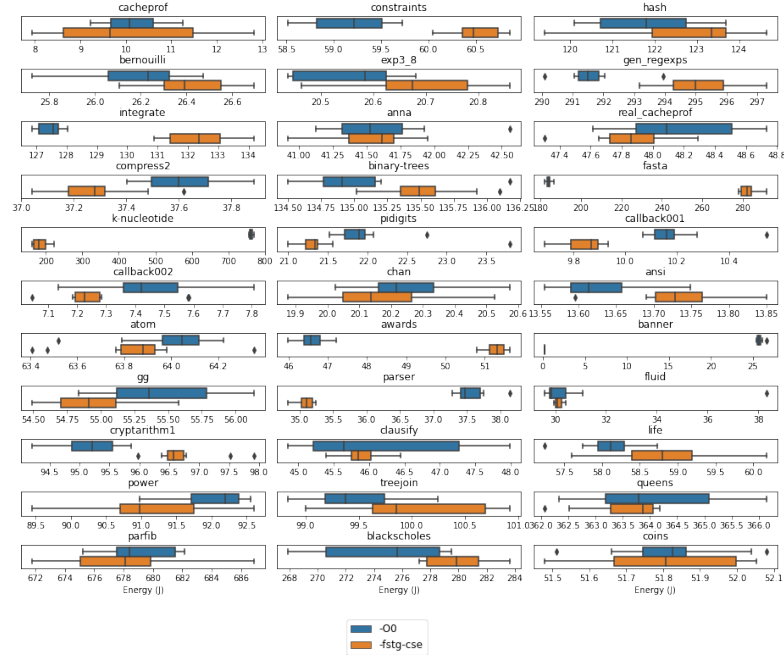


Figure A.1: Energy consumption of all benchmarks when compiled with `-O0` and `-fcase-merge`

Figure A.2: Energy consumption of all benchmarks when compiled with `-O0` and `-fcase-folding`Figure A.3: Energy consumption of all benchmarks when compiled with `-O0` and `-fcall-arity`

Figure A.4: Energy consumption of all benchmarks when compiled with `-O0` and `-fexitification`Figure A.5: Energy consumption of all benchmarks when compiled with `-O0` and `-fcmm-elim-common-blocks`

Figure A.6: Energy consumption of all benchmarks when compiled with `-O0` and `-fcmm-sink`Figure A.7: Energy consumption of all benchmarks when compiled with `-O0` and `-fcpr-anal`

Figure A.8: Energy consumption of all benchmarks when compiled with `-O0` and `-fcse`Figure A.9: Energy consumption of all benchmarks when compiled with `-O0` and `-fstg-cse`

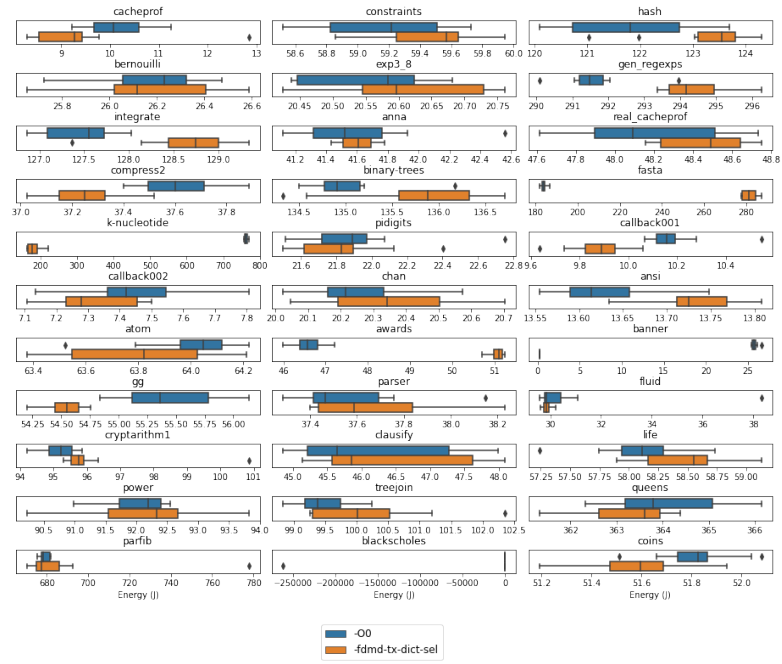


Figure A.10: Energy consumption of all benchmarks when compiled with `-O0` and `-fdmd-tx-dict-sel`

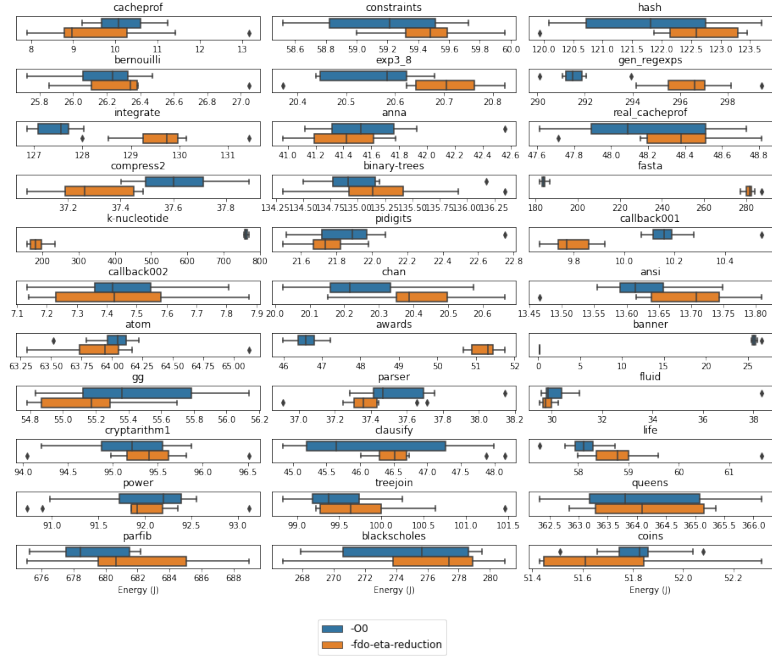


Figure A.11: Energy consumption of all benchmarks when compiled with `-O0` and `-fdo-eta-reduction`

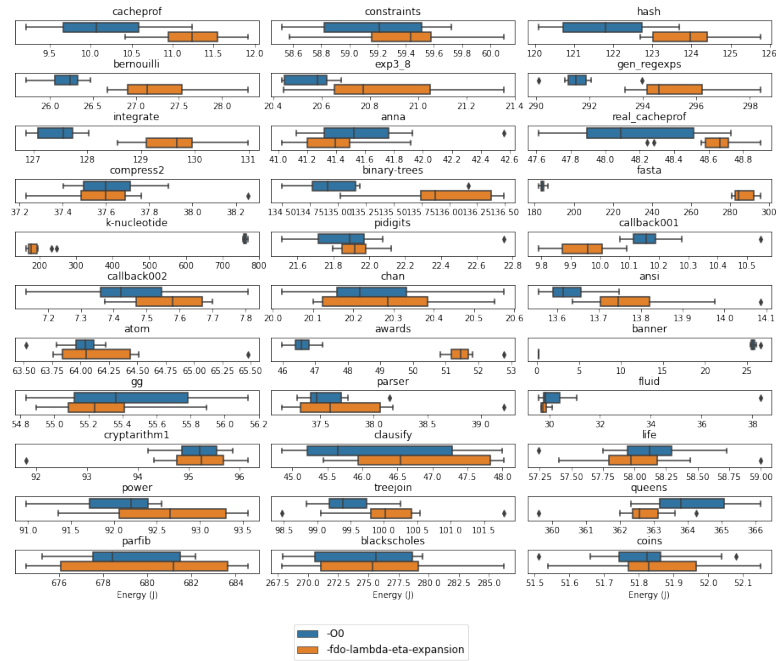


Figure A.12: Energy consumption of all benchmarks when compiled with `-O0` and `-fdo-lambda-eta-expansion`

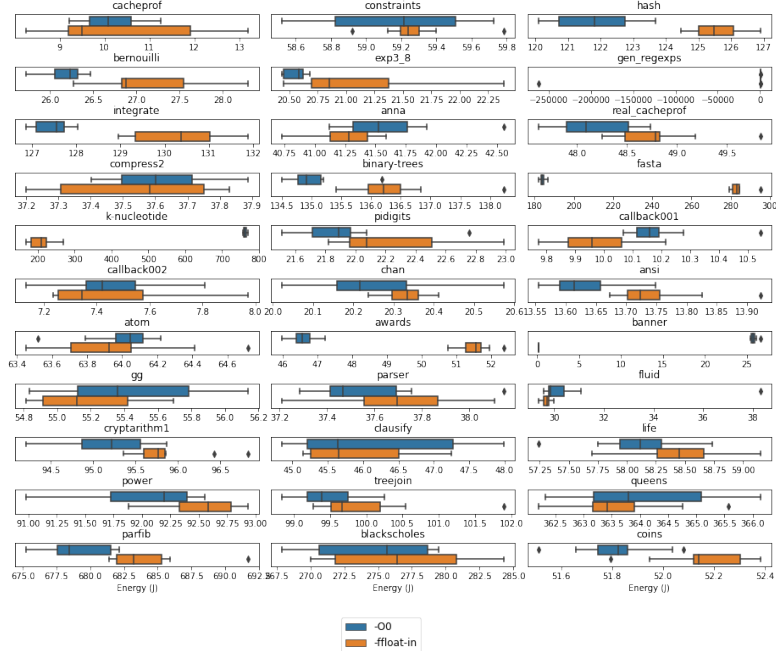


Figure A.13: Energy consumption of all benchmarks when compiled with `-O0` and `-ffloat-in`

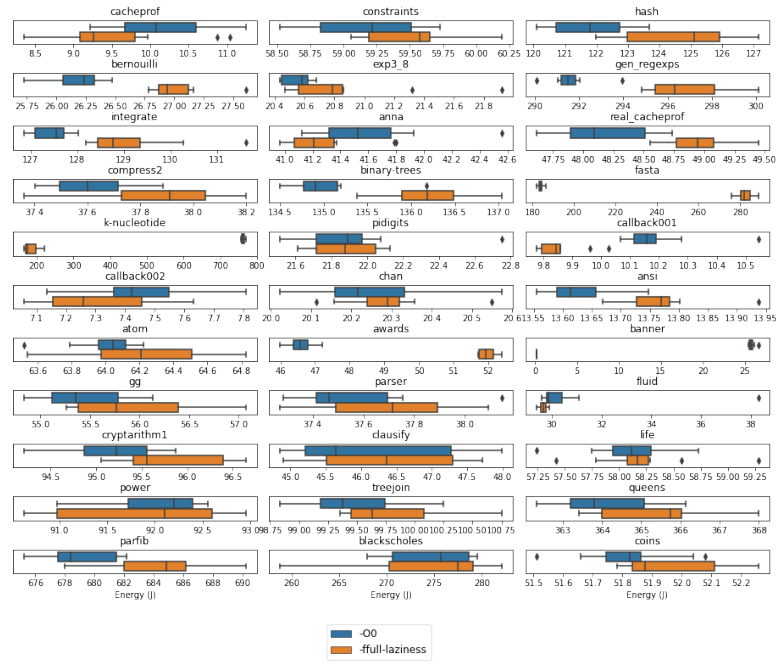


Figure A.14: Energy consumption of all benchmarks when compiled with `-O0` and `-ffull-laziness`

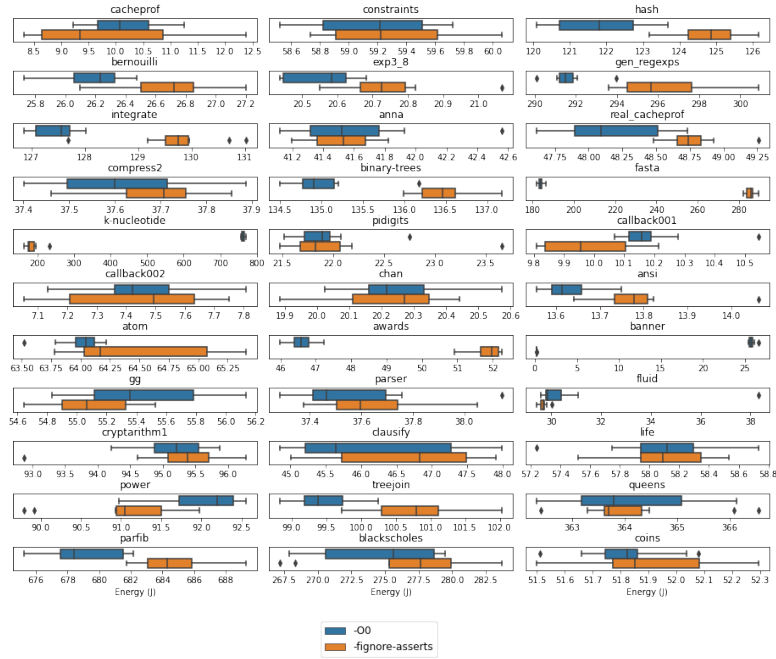


Figure A.15: Energy consumption of all benchmarks when compiled with `-O0` and `-fignore-asserts`

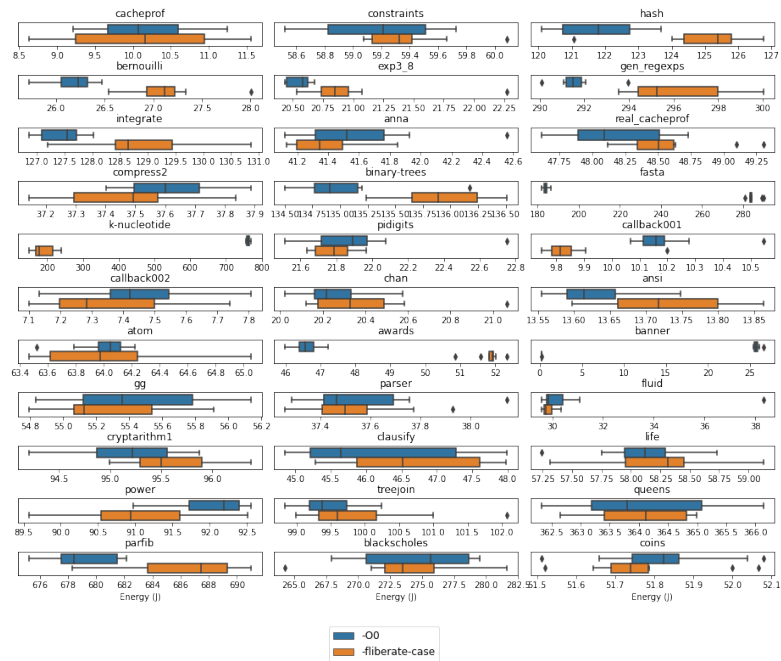


Figure A.16: Energy consumption of all benchmarks when compiled with `-O0` and `-fliberate-case`

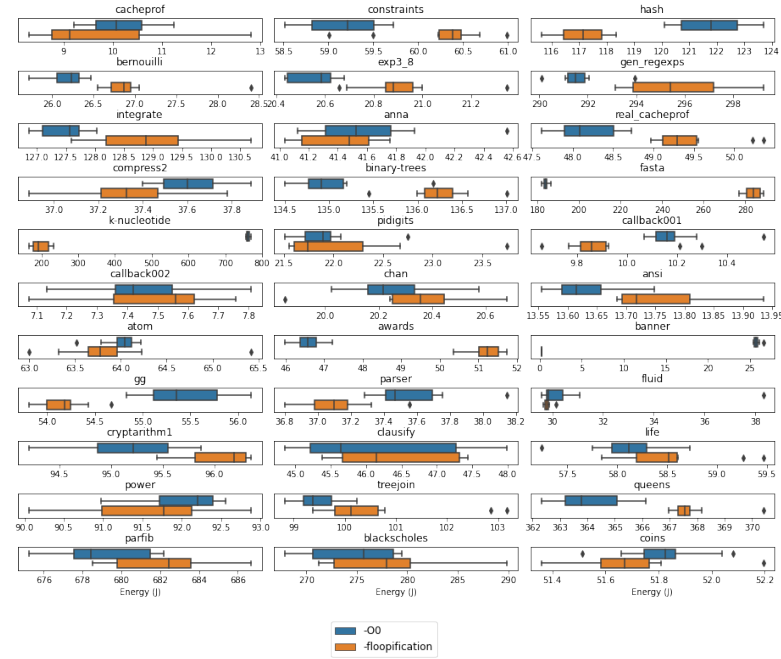


Figure A.17: Energy consumption of all benchmarks when compiled with `-O0` and `-floopification`

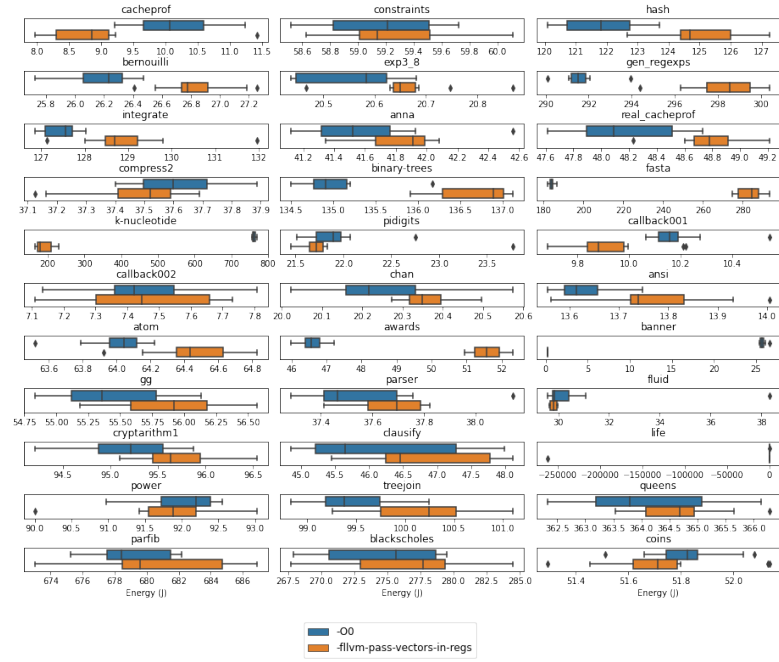


Figure A.18: Energy consumption of all benchmarks when compiled with `-O0` and `-fllvm-pass-vectors-in-regs`

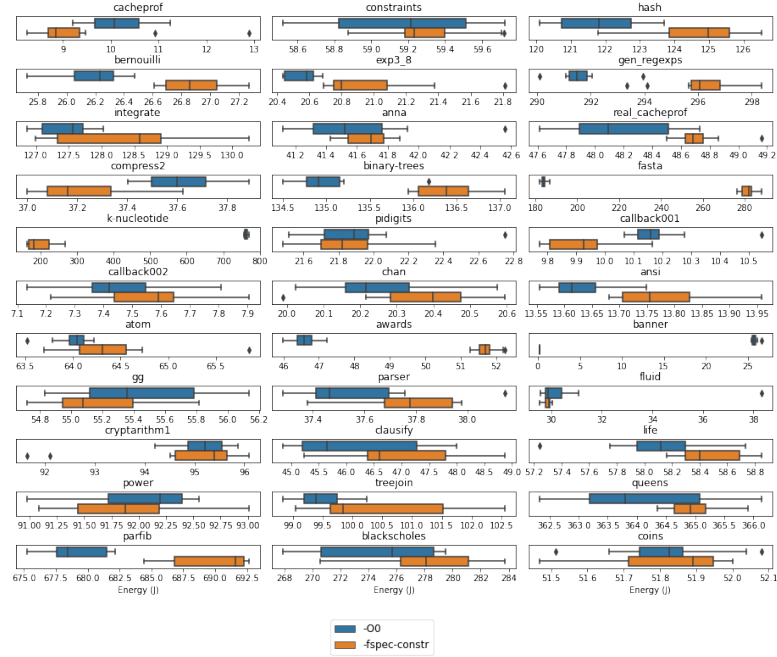
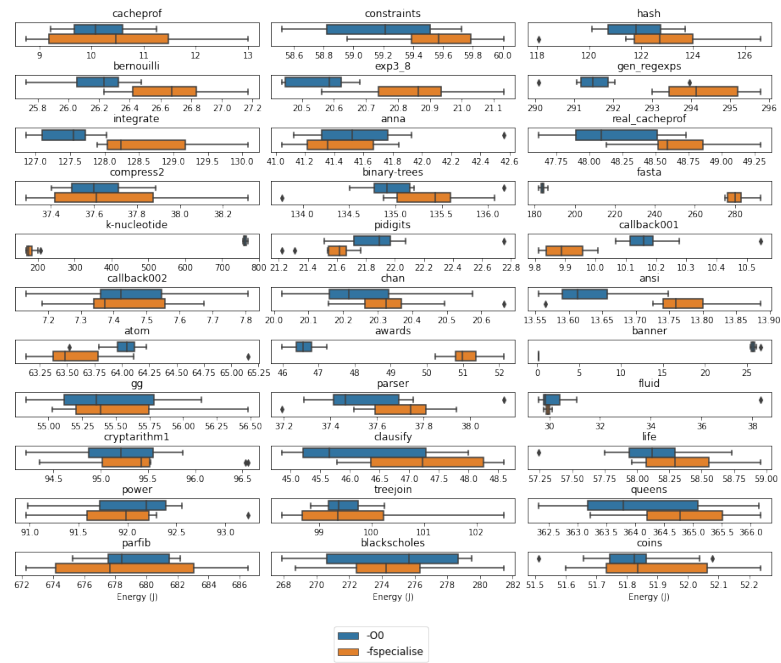
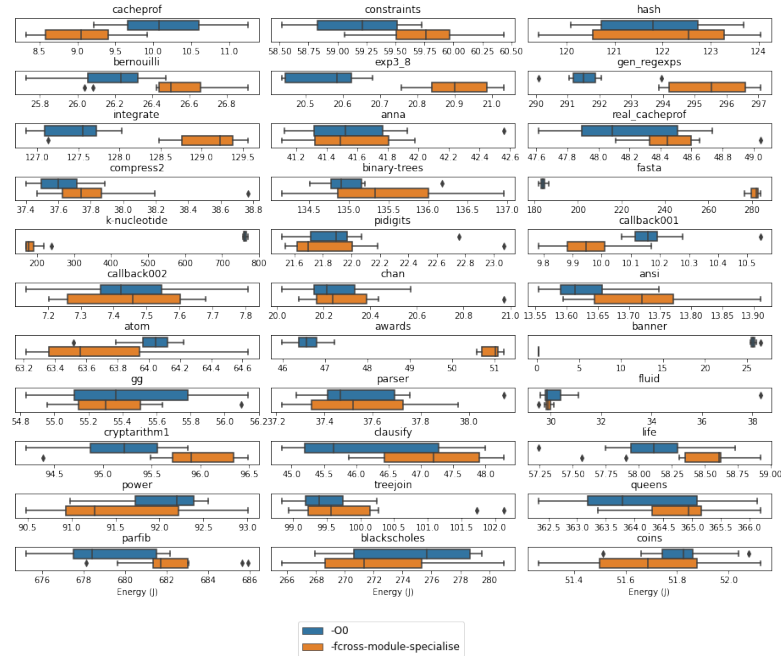


Figure A.19: Energy consumption of all benchmarks when compiled with `-O0` and `-fspec-constr`

Figure A.20: Energy consumption of all benchmarks when compiled with `-O0` and `-fspecialise`Figure A.21: Energy consumption of all benchmarks when compiled with `-O0` and `-fcross-module-specialise`

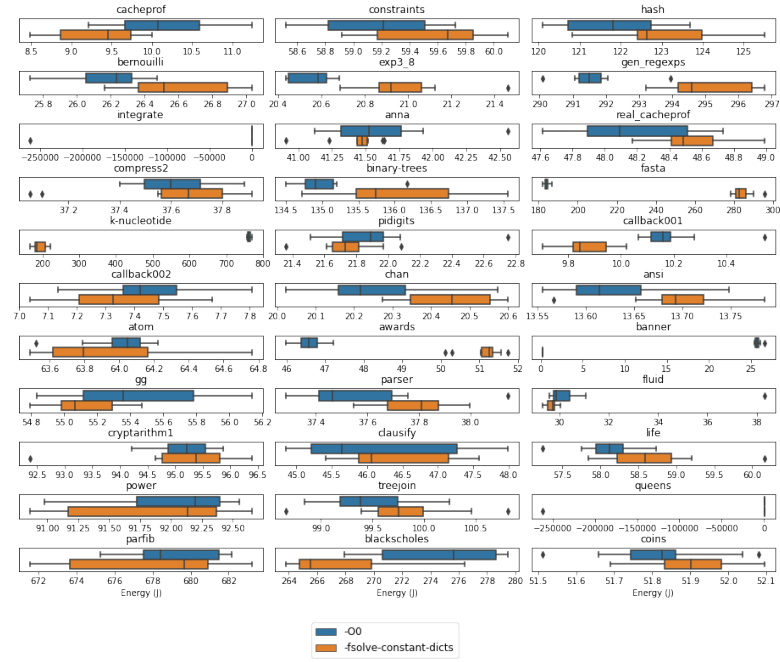


Figure A.22: Energy consumption of all benchmarks when compiled with `-O0` and `-fsolve-constant-dicts`

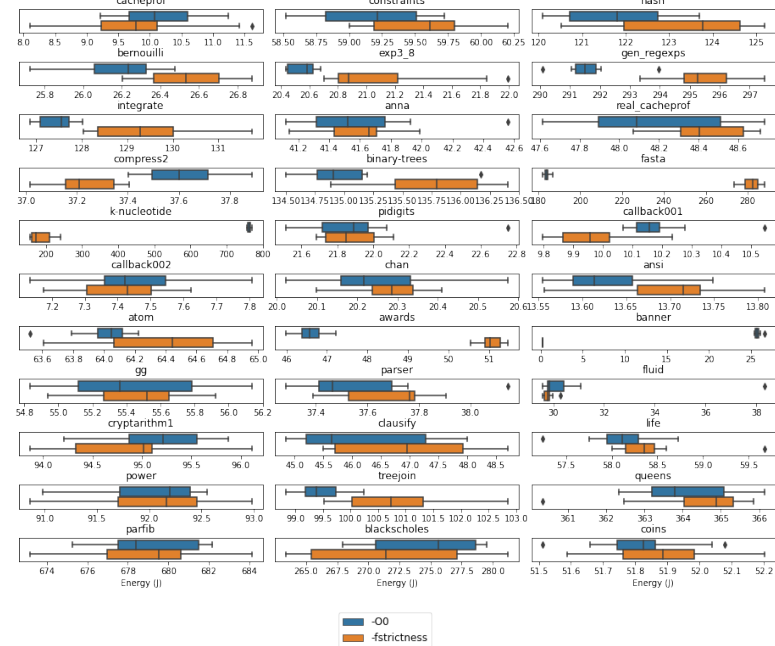


Figure A.23: Energy consumption of all benchmarks when compiled with `-O0` and `-fstictness`

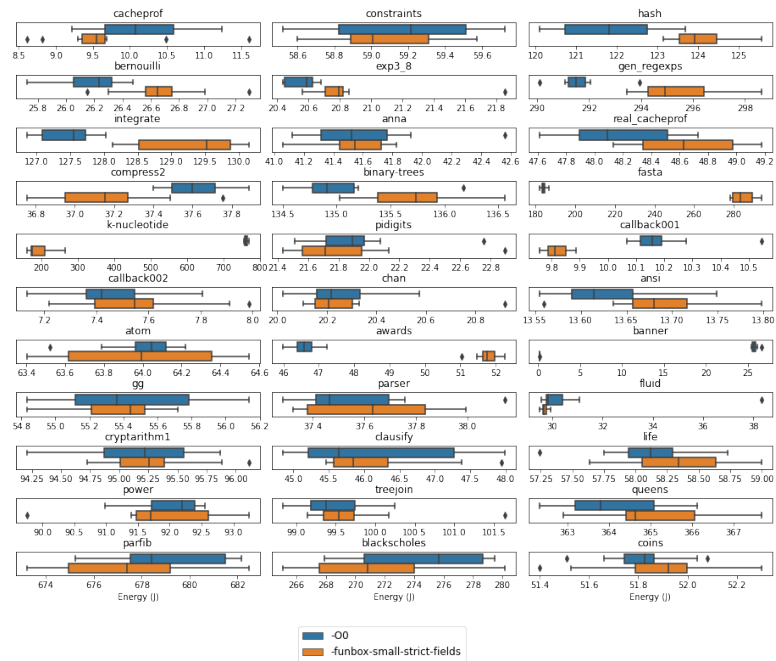


Figure A.24: Energy consumption of all benchmarks when compiled with `-O0` and `-funbox-small-strict-fields`

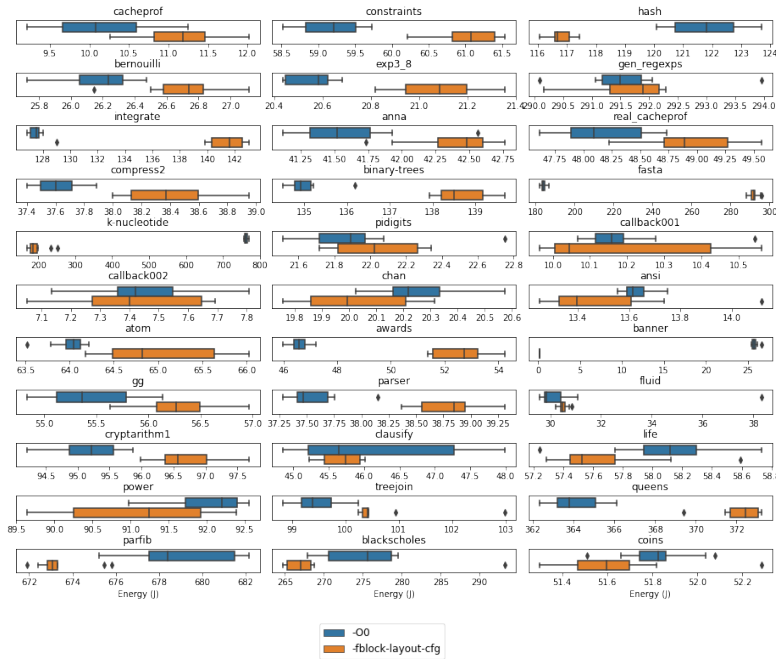


Figure A.25: Energy consumption of all benchmarks when compiled with `-O0` and `-fblock-layout-cfg`

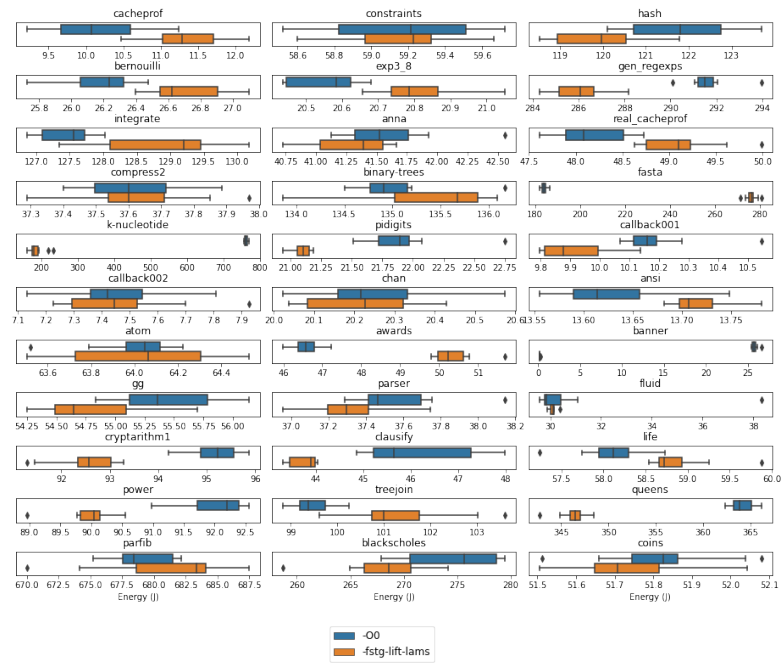


Figure A.26: Energy consumption of all benchmarks when compiled with `-O0` and `-fstg-lift-lams`

A.1.2 Statistical Tests

Flag	$\rho < 0.05$		$\rho > 0.05$		Did not respect assumptions	
	Count	Programs	Count	Programs	Count	Programs
-fcase-merge	8	[hash, gen_regexps, integrate, fasta, pidigits, ansi, awards, coins]	20	[cacheprof, constraints, bernouilli, anna, real_cacheprof, compress2, binary-trees, callback001, callback002, chan, atom, gg, parser, cryptarithm1, life, power, treejoin, queens, parfib, blackscholes]	5	[exp3_8, k-nucleotide, banner, fluid, clausify]
-fcase-folding	9	[gen_regexps, compress2, binary-trees, fasta, pidigits, ansi, awards, parser, life]	18	[cacheprof, constraints, hash, bernouilli, integrate, anna, real_cacheprof, callback001, callback002, chan, atom, cryptarithm1, power, treejoin, queens, parfib, blackscholes, coins]	6	[exp3_8, k-nucleotide, banner, gg, fluid, clausify]

-fcall-arity	9	[integrate, anna, fasta, pidigits, callback001, awards, parser, power, coins]	19	[cacheprof, constraints, hash, bernouilli, gen_regexp, real_cacheprof, compress2, binary-trees, callback002, chan, ansi, atom, gg, cryptarithm1, life, treejoin, queens, parfib, blackscholes]	5	[exp3_8, k-nucleotide, banner, fluid, clausify]
-fexitification	13	[cacheprof, gen_regexp, real_cacheprof, compress2, binary-trees, fasta, pidigits, callback001, ansi, awards, gg, life, coins]	14	[hash, bernouilli, integrate, anna, callback002, chan, atom, parser, cryptarithm1, power, treejoin, queens, parfib, blackscholes]	6	[constraints, exp3_8, k-nucleotide, banner, fluid, clausify]
-fcmm-elim-common-blocks	16	[constraints, hash, gen_regexp, integrate, binary-trees, fasta, ansi, awards, gg, parser, life, power, treejoin, parfib, blackscholes, coins]	11	[cacheprof, bernouilli, anna, real_cacheprof, compress2, pidigits, callback002, chan, atom, cryptarithm1, queens]	6	[exp3_8, k-nucleotide, callback001, banner, fluid, clausify]

-fcmm-sink	21	[cacheprof, hash, bernouilli, integrate, compress2, binary-trees, fasta, pidigits, callback001, callback002, chan, ansi, atom, awards, parser, cryptarithm1, life, queens, parfib, blackscholes, coins]	5	[constraints, anna, real_cacheprof, gg, treejoin]	7	[exp3_8, gen_regexp, k-nucleotide, banner, fluid, clausify, power]
-fcpr-anal	13	[cacheprof, constraints, gen_regexp, fasta, callback001, ansi, atom, awards, gg, parser, life, parfib, coins]	16	[hash, bernouilli, integrate, anna, real_cacheprof, compress2, binary-trees, pidigits, callback002, chan, cryptarithm1, power, treejoin, queens, blacksholes, clausify]	4	[exp3_8, k-nucleotide, banner, fluid]
-fcse	16	[bernouilli, gen_regexp, integrate, anna, real_cacheprof, binary-trees, fasta, callback001, ansi, awards, gg, parser, power, queens, parfib, coins]	12	[cacheprof, constraints, hash, compress2, pidigits, callback002, chan, atom, cryptarithm1, life, treejoin, blackscholes]	5	[exp3_8, k-nucleotide, banner, fluid, clausify]

-fstg-cse	20	[constraints, bernouilli, gen_regexp, integrate, compress2, binary-trees, fasta, pidigits, callback001, callback002, ansi, atom, awards, gg, parser, cryptarithm1, life, power, treejoin, blacksholes]	7	[cacheprof, hash, anna, real_cacheprof, chan, parfib, coins]	6	[exp3_8, k-nucleotide, banner, fluid, clausify, queens]
-fdmd-tx-dict-sel	14	[cacheprof, hash, gen_regexp, integrate, compress2, binary-trees, fasta, callback001, ansi, atom, awards, gg, cryptarithm1, coins]	15	[constraints, bernouilli, anna, real_cacheprof, pidigits, callback002, chan, parser, life, power, treejoin, queens, parfib, blacksholes, clausify]	4	[exp3_8, k-nucleotide, banner, fluid]
-fdo-eta-reduction	12	[cacheprof, gen_regexp, integrate, compress2, fasta, callback001, chan, ansi, atom, awards, parser, life]	15	[constraints, hash, anna, real_cacheprof, binary-trees, pidigits, callback002, gg, cryptarithm1, power, treejoin, queens, parfib, blacksholes, coins]	6	[bernouilli, exp3_8, k-nucleotide, banner, fluid, clausify]

-fdo-lambda-eta-expansion	13	[cacheprof, hash, bernouilli, gen_regexps, integrate, real_cacheprof, binary-trees, fasta, callback001, ansi, awards, treejoin, queens]	15	[constraints, anna, compress2, pidigits, callback002, chan, atom, gg, parser, cryptarithm1, life, power, blackscholes, coins, fluid]	5	[exp3_8, k-nucleotide, banner, clausify, parfib]
-ffloat-in	14	[hash, bernouilli, gen_regexps, integrate, real_cacheprof, binary-trees, fasta, callback001, ansi, awards, cryptarithm1, power, parfib, coins]	12	[cacheprof, constraints, anna, compress2, chan, atom, gg, parser, life, treejoin, queens, blackscholes]	7	[exp3_8, k-nucleotide, pidigits, callback002, banner, fluid, clausify]
-ffull-laziness	15	[cacheprof, hash, bernouilli, gen_regexps, integrate, anna, real_cacheprof, compress2, binary-trees, fasta, callback001, ansi, awards, cryptarithm1, parfib]	11	[constraints, pidigits, callback002, chan, atom, gg, parser, life, power, treejoin, queens]	7	[exp3_8, k-nucleotide, banner, fluid, clausify, blackscholes, coins]
-fignore-asserts	14	[hash, bernouilli, gen_regexps, integrate, real_cacheprof, binary-trees, fasta, callback001, ansi, awards, power, treejoin, parfib, blackscholes]	14	[cacheprof, constraints, anna, compress2, pidigits, callback002, chan, atom, gg, parser, cryptarithm1, life, queens, coins]	5	[exp3_8, k-nucleotide, banner, fluid, clausify]

-fliberate-case	11	[hash, bernouilli, gen_regexps, integrate, binary-trees, fasta, callback001, ansi, awards, power, parfib]	17	[cacheprof, constraints, anna, real_cacheprof, compress2, pidigits, callback002, chan, atom, gg, parser, cryptarithm1, life, treejoin, queens, blacksc-holes, coins]	5	[exp3_8, k-nucleotide, banner, fluid, clausify]
-floopification	19	[constraints, hash, bernouilli, gen_regexps, integrate, real_cacheprof, compress2, binary-trees, fasta, callback001, atom, awards, gg, parser, cryptarithm1, treejoin, queens, parfib, coins]	8	[cacheprof, anna, callback002, chan, life, power, blackscholes, clausify]	6	[exp3_8, k-nucleotide, pidigits, ansi, banner, fluid]
-flvm-pass-vectors-in-regs	18	[cacheprof, hash, bernouilli, gen_regexps, integrate, anna, real_cacheprof, binary-trees, fasta, callback001, ansi, atom, awards, gg, parser, cryptarithm1, tree-join, coins]	10	[constraints, compress2, pidigits, callback002, chan, life, power, queens, parfib, blackscholes]	5	[exp3_8, k-nucleotide, banner, fluid, clausify]

-fspec-constr	16	[cacheprof, hash, bernouilli, gen_regexps, integrate, real_cacheprof, compress2, binary-trees, fasta, callback001, chan, ansi, awards, parser, life, exp3_8]	11	[constraints, anna, pidigits, callback002, atom, gg, cryptarithm1, power, queens, blackscholes, coins]	6	[k-nucleotide, banner, fluid, clausify, treejoin, parfib]
-fspecialise	11	[constraints, bernouilli, gen_regexps, real_cacheprof, binary-trees, fasta, pidigits, callback001, atom, awards, parser]	15	[cacheprof, hash, anna, compress2, callback002, chan, gg, life, power, treejoin, queens, parfib, blackscholes, coins, clausify]	7	[exp3_8, integrate, k-nucleotide, ansi, banner, fluid, cryptarithm1]
-fcross-module-specialise	12	[cacheprof, constraints, bernouilli, gen_regexps, integrate, fasta, callback001, ansi, awards, cryptarithm1, life, parfib]	16	[hash, anna, real_cacheprof, compress2, binary-trees, pidigits, callback002, chan, atom, gg, parser, power, treejoin, queens, blackscholes, coins]	5	[exp3_8, k-nucleotide, banner, fluid, clausify]

-fsolve-constant-dicts	13	[cacheprof, hash, bernouilli, gen_regexps, integrate, real_cacheprof, binary-trees, fasta, callback001, chan, ansi, awards, parser]	13	[constraints, anna, compress2, pidigits, callback002, atom, gg, cryptarithm1, life, power, treejoin, queens, coins]	7	[exp3_8, k-nucleotide, banner, fluid, clausify, parfib, blackscholes]
-fstrictness	13	[hash, bernouilli, gen_regexps, integrate, compress2, binary-trees, fasta, callback001, ansi, awards, parser, treejoin, exp3_8]	16	[cacheprof, constraints, anna, real_cacheprof, pidigits, callback002, chan, atom, gg, cryptarithm1, life, power, queens, parfib, blacksholes, coins]	4	[k-nucleotide, banner, fluid, clausify]
-funbox-small-strict-fields	11	[hash, bernouilli, gen_regexps, integrate, real_cacheprof, compress2, binary-trees, fasta, callback001, ansi, awards]	18	[cacheprof, constraints, anna, pidigits, callback002, chan, atom, gg, parser, cryptarithm1, life, power, treejoin, queens, parfib, blackscholes, coins, clausify]	4	[exp3_8, k-nucleotide, banner, fluid]

-fblock-layout-cfg	23	[cacheprof, constraints, hash, bernouilli, integrate, anna, real_cacheprof, compress2, binary-trees, fasta, pidigits, chan, atom, awards, gg, parser, cryptarithm1, life, power, queens, parfib, blacksc-holes, coins]	2	[gen_regexp, call-back002]	8	[exp3_8, k-nucleotide, callback001, ansi, banner, fluid, clausify, treejoin]
-fstg-lift-lams	20	[cacheprof, hash, bernouilli, gen_regexp, integrate, real_cacheprof, binary-trees, fasta, pidigits, callback001, ansi, awards, gg, cryptarithm1, life, power, treejoin, queens, blackscholes, clausify]	10	[constraints, anna, compress2, callback002, chan, atom, parser, parfib, coins, fluid]	3	[exp3_8, k-nucleotide, banner]

Table A.1: Benchmarks that have a statistically significant difference in energy consumption when compared to their non-optimized versions ($\rho < 0.05$), that do not ($\rho > 0.05$), and that did not respect the assumptions of ANOVA, by optimization flag

Flag	Benchmark	Effect Size	Flag	Benchmark	Effect Size
-fcase-merge	hash	-1.1437	-ffull-laziness	fasta	33.66177

-fcase-merge	gen_regexps	1.0029	-ffull-laziness	callback001	-6.23866
-fcase-merge	integrate	1.0956	-ffull-laziness	ansi	1.94529
-fcase-merge	fasta	47.0791	-ffull-laziness	awards	17.34258
-fcase-merge	pidigits	1.6208	-ffull-laziness	cryptarithm1	1.04604
-fcase-merge	ansi	1.0065	-ffull-laziness	parfib	1.54083
-fcase-merge	awards	9.7659	-fignore-asserts	hash	2.48827
-fcase-merge	coins	-1.1940	-fignore-asserts	bernouilli	1.67802
-fcase-folding	gen_regexps	-1.1452	-fignore-asserts	gen_regexps	2.63046
-fcase-folding	compress2	-1.2417	-fignore-asserts	integrate	6.02867
-fcase-folding	binary-trees	-0.9739	-fignore-asserts	real_cacheprof	1.76196
-fcase-folding	fasta	20.8254	-fignore-asserts	binary-trees	4.72939
-fcase-folding	pidigits	1.4246	-fignore-asserts	fasta	49.54656
-fcase-folding	ansi	1.6317	-fignore-asserts	callback001	-1.35911
-fcase-folding	awards	11.9496	-fignore-asserts	ansi	2.03988
-fcase-folding	parser	2.2685	-fignore-asserts	awards	13.21979
-fcase-folding	life	0.9928	-fignore-asserts	power	-1.30658
-fcall-arity	integrate	1.5252	-fignore-asserts	treejoin	2.11535
-fcall-arity	anna	-1.2104	-fignore-asserts	parfib	2.21970
-fcall-arity	fasta	18.5956	-fignore-asserts	blackscholes	1.01710
-fcall-arity	pidigits	2.0665	-fliberate-case	hash	2.97841
-fcall-arity	callback001	-2.1205	-fliberate-case	bernouilli	3.33378
-fcall-arity	awards	11.5192	-fliberate-case	gen_regexps	2.47147
-fcall-arity	parser	1.0575	-fliberate-case	integrate	1.58613
-fcall-arity	power	-0.9253	-fliberate-case	binary-trees	2.75097
-fcall-arity	coins	-1.4013	-fliberate-case	fasta	85.11834
-fexitification	cacheprof	2.1278	-fliberate-case	callback001	-5.74026

-fexitification	gen_regexp	2.2122	-fliberate-case	ansi	1.24060
-fexitification	real_cache	0.9366	-fliberate-case	awards	18.72114
-fexitification	compress2	-1.2036	-fliberate-case	power	-1.33737
-fexitification	binary-trees	1.9576	-fliberate-case	parfib	2.02784
-fexitification	fasta	60.4018	-floopification	constraints	3.39469
-fexitification	pidigits	1.2927	-floopification	hash	-4.04385
-fexitification	callback001	-5.4068	-floopification	bernouilli	2.83133
-fexitification	ansi	1.2000	-floopification	gen_regexp	2.39627
-fexitification	awards	10.7597	-floopification	integrate	1.89957
-fexitification	gg	-2.0336	-floopification	real_cache	3.27965
-fexitification	life	1.0793	-floopification	compress2	-1.06948
-fexitification	coins	-2.2649	-floopification	binary-trees	5.93890
-fcmm-elim-common-blocks	constraints	2.0656	-floopification	fasta	31.88224
-fcmm-elim-common-blocks	hash	-3.7236	-floopification	callback001	-5.04012
-fcmm-elim-common-blocks	gen_regexp	-4.7761	-floopification	atom	-1.25020
-fcmm-elim-common-blocks	integrate	10.7122	-floopification	awards	11.69024
-fcmm-elim-common-blocks	binary-trees	1.4384	-floopification	gg	-3.63434
-fcmm-elim-common-blocks	fasta	33.5832	-floopification	parser	-2.54978
-fcmm-elim-common-blocks	ansi	-3.9175	-floopification	cryptarithm1	1.85333
-fcmm-elim-common-blocks	awards	7.1320	-floopification	treejoin	1.07877
-fcmm-elim-common-blocks	gg	-0.9989	-floopification	queens	3.33352
-fcmm-elim-common-blocks	parser	2.6341	-floopification	parfib	1.16424
-fcmm-elim-common-blocks	life	1.1248	-floopification	coins	-1.36032
-fcmm-elim-common-blocks	power	-4.8324	-flvm-pass-vectors-in-regs	cache	-2.41796
-fcmm-elim-common-blocks	treejoin	1.6416	-flvm-pass-vectors-in-regs	hash	2.37980
-fcmm-elim-common-blocks	parfib	-5.3144	-flvm-pass-vectors-in-regs	bernouilli	2.77926

-fcmm-elim-common-blocks	blackscholes	-1.8336	-flvm-pass-vectors-in-regs	gen_regexps	6.72605
-fcmm-elim-common-blocks	coins	-1.1168	-flvm-pass-vectors-in-regs	integrate	2.77278
-fcmm-sink	cacheprof	-1.4700	-flvm-pass-vectors-in-regs	anna	1.14377
-fcmm-sink	hash	-3.0323	-flvm-pass-vectors-in-regs	real_cacheprof	2.08754
-fcmm-sink	bernouilli	-1.4939	-flvm-pass-vectors-in-regs	binary-trees	4.78547
-fcmm-sink	integrate	-4.2382	-flvm-pass-vectors-in-regs	fasta	19.79754
-fcmm-sink	compress2	-1.8943	-flvm-pass-vectors-in-regs	callback001	-3.59262
-fcmm-sink	binary-trees	5.2023	-flvm-pass-vectors-in-regs	ansi	1.31733
-fcmm-sink	fasta	68.5357	-flvm-pass-vectors-in-regs	atom	2.34203
-fcmm-sink	pidigits	-1.5052	-flvm-pass-vectors-in-regs	awards	11.77508
-fcmm-sink	callback001	-2.2716	-flvm-pass-vectors-in-regs	gg	1.05075
-fcmm-sink	callback002	-1.4779	-flvm-pass-vectors-in-regs	parser	1.08203
-fcmm-sink	chan	-1.3757	-flvm-pass-vectors-in-regs	cryptarithm1	1.05895
-fcmm-sink	ansi	-1.5910	-flvm-pass-vectors-in-regs	treejoin	1.28425
-fcmm-sink	atom	-1.8280	-flvm-pass-vectors-in-regs	coins	-1.23636
-fcmm-sink	awards	6.4904	-fspec-constr	cacheprof	-2.31713
-fcmm-sink	parser	-5.1247	-fspec-constr	hash	1.90421
-fcmm-sink	cryptarithm1	-2.8719	-fspec-constr	bernouilli	2.83338
-fcmm-sink	life	3.0293	-fspec-constr	gen_regexps	6.46063
-fcmm-sink	queens	-5.3155	-fspec-constr	integrate	1.01870
-fcmm-sink	parfib	-14.2047	-fspec-constr	real_cacheprof	1.70318
-fcmm-sink	blackscholes	-3.4608	-fspec-constr	compress2	-2.00543
-fcmm-sink	coins	4.1679	-fspec-constr	binary-trees	4.46584
-fcpr-anal	cacheprof	-1.7625	-fspec-constr	fasta	33.39405
-fcpr-anal	constraints	1.0904	-fspec-constr	callback001	-2.08114
-fcpr-anal	gen_regexps	3.2685	-fspec-constr	chan	1.04894

-fcpr-anal	fasta	21.8951	-fspec-constr	ansi	1.74236
-fcpr-anal	callback001	-5.1490	-fspec-constr	awards	15.71733
-fcpr-anal	ansi	1.1230	-fspec-constr	parser	1.30260
-fcpr-anal	atom	-1.5588	-fspec-constr	life	0.99057
-fcpr-anal	awards	10.8064	-fspec-constr	exp3_8	1.90390
-fcpr-anal	gg	-1.5304	-fspecialise	constraints	1.00367
-fcpr-anal	parser	0.9517	-fspecialise	bernouilli	1.66073
-fcpr-anal	life	0.9883	-fspecialise	gen_regexp	3.26624
-fcpr-anal	parfib	1.0259	-fspecialise	real_cacheprof	1.36724
-fcpr-anal	coins	-2.1233	-fspecialise	binary-trees	1.60362
-fcse	bernouilli	0.9663	-fspecialise	fasta	21.25005
-fcse	gen_regexp	3.0252	-fspecialise	pidigits	-1.17590
-fcse	integrate	3.0821	-fspecialise	callback001	-3.68411
-fcse	anna	-1.2472	-fspecialise	atom	-2.14946
-fcse	real_cacheprof	1.3477	-fspecialise	awards	9.15737
-fcse	binary-trees	1.4935	-fspecialise	parser	1.50477
-fcse	fasta	37.4755	-fcross-module-specialise	cacheprof	-1.68135
-fcse	callback001	-4.9436	-fcross-module-specialise	constraints	1.35305
-fcse	ansi	2.3463	-fcross-module-specialise	bernouilli	1.91569
-fcse	awards	11.7343	-fcross-module-specialise	gen_regexp	3.67857
-fcse	gg	-1.6083	-fcross-module-specialise	integrate	4.25446
-fcse	parser	1.2247	-fcross-module-specialise	fasta	47.93232
-fcse	power	-0.9711	-fcross-module-specialise	callback001	-2.06135
-fcse	queens	-0.9045	-fcross-module-specialise	ansi	1.08309
-fcse	parfib	1.1153	-fcross-module-specialise	awards	14.14276
-fcse	coins	-1.1837	-fcross-module-specialise	cryptarithm1	1.75895

-fstg-cse	constraints	3.5432	-fcross-module-specialise	life	1.63807
-fstg-cse	bernouilli	1.0182	-fcross-module-specialise	parfib	1.23301
-fstg-cse	gen_regexps	3.5419	-fsolve-constant-dicts	cacheprof	-1.27260
-fstg-cse	integrate	5.5290	-fsolve-constant-dicts	hash	0.92132
-fstg-cse	compress2	-2.3115	-fsolve-constant-dicts	bernouilli	1.32103
-fstg-cse	binary-trees	2.2014	-fsolve-constant-dicts	gen_regexps	3.17945
-fstg-cse	fasta	30.9747	-fsolve-constant-dicts	integrate	2.24080
-fstg-cse	pidigits	-2.9437	-fsolve-constant-dicts	real_cacheprof	1.11716
-fstg-cse	callback001	-4.1695	-fsolve-constant-dicts	binary-trees	1.53391
-fstg-cse	callback002	-1.2787	-fsolve-constant-dicts	fasta	34.93129
-fstg-cse	ansi	1.9253	-fsolve-constant-dicts	callback001	-3.25699
-fstg-cse	atom	-1.4307	-fsolve-constant-dicts	chan	1.32239
-fstg-cse	awards	13.9897	-fsolve-constant-dicts	ansi	1.35302
-fstg-cse	gg	-1.1798	-fsolve-constant-dicts	awards	15.46720
-fstg-cse	parser	-14.9261	-fsolve-constant-dicts	parser	1.78712
-fstg-cse	cryptarithm1	3.1456	-fstrictness	hash	0.90827
-fstg-cse	life	1.0425	-fstrictness	bernouilli	1.44133
-fstg-cse	power	-1.0912	-fstrictness	gen_regexps	4.07796
-fstg-cse	treejoin	0.9818	-fstrictness	integrate	2.08033
-fstg-cse	blackscholes	1.4077	-fstrictness	compress2	-2.43822
-fdmd-tx-dict-sel	cacheprof	-1.7284	-fstrictness	binary-trees	1.86823
-fdmd-tx-dict-sel	hash	1.7657	-fstrictness	fasta	26.35395
-fdmd-tx-dict-sel	gen_regexps	3.8525	-fstrictness	callback001	-1.80874
-fdmd-tx-dict-sel	integrate	3.2624	-fstrictness	ansi	0.94157
-fdmd-tx-dict-sel	compress2	-2.1262	-fstrictness	awards	13.22274
-fdmd-tx-dict-sel	binary-trees	2.0319	-fstrictness	parser	0.99028

-fdmd-tx-dict-sel	fasta	32.7504	-fstrictness	treejoin	1.59451
-fdmd-tx-dict-sel	callback001	-2.8181	-fstrictness	exp3_8	1.60392
-fdmd-tx-dict-sel	ansi	1.6724	-funbox-small-strict-fields	hash	2.03507
-fdmd-tx-dict-sel	atom	-0.9946	-funbox-small-strict-fields	bernouilli	1.98464
-fdmd-tx-dict-sel	awards	15.6364	-funbox-small-strict-fields	gen_regexps	2.75169
-fdmd-tx-dict-sel	gg	-2.5845	-funbox-small-strict-fields	integrate	2.83099
-fdmd-tx-dict-sel	cryptarithm1	1.1675	-funbox-small-strict-fields	real_cacheprof	1.22331
-fdmd-tx-dict-sel	coins	-1.1922	-funbox-small-strict-fields	compress2	-2.51565
-fdo-eta-reduction	cacheprof	-0.9580	-funbox-small-strict-fields	binary-trees	2.01577
-fdo-eta-reduction	gen_regexps	4.8667	-funbox-small-strict-fields	fasta	22.49287
-fdo-eta-reduction	integrate	4.5101	-funbox-small-strict-fields	callback001	-6.06306
-fdo-eta-reduction	compress2	-1.8552	-funbox-small-strict-fields	ansi	1.12954
-fdo-eta-reduction	fasta	48.5436	-funbox-small-strict-fields	awards	16.40976
-fdo-eta-reduction	callback001	-4.6671	-fblock-layout-cfg	cacheprof	1.61375
-fdo-eta-reduction	chan	0.9233	-fblock-layout-cfg	constraints	4.19597
-fdo-eta-reduction	ansi	1.1647	-fblock-layout-cfg	hash	-5.15800
-fdo-eta-reduction	atom	-0.9553	-fblock-layout-cfg	bernouilli	2.50709
-fdo-eta-reduction	awards	12.7816	-fblock-layout-cfg	integrate	16.04795
-fdo-eta-reduction	parser	-1.0782	-fblock-layout-cfg	anna	3.50188
-fdo-eta-reduction	life	1.0833	-fblock-layout-cfg	real_cacheprof	1.87001
-fdo-lambda-eta-expansion	cacheprof	1.7932	-fblock-layout-cfg	compress2	2.84056
-fdo-lambda-eta-expansion	hash	1.7355	-fblock-layout-cfg	binary-trees	7.46313
-fdo-lambda-eta-expansion	bernouilli	2.6826	-fblock-layout-cfg	fasta	58.29314
-fdo-lambda-eta-expansion	gen_regexps	2.6781	-fblock-layout-cfg	pidigits	0.96986
-fdo-lambda-eta-expansion	integrate	3.5897	-fblock-layout-cfg	chan	-1.16849
-fdo-lambda-eta-expansion	real_cacheprof	1.7663	-fblock-layout-cfg	atom	1.76192

-fdo-lambda-eta-expansion	binary-trees	2.4392	-fblock-layout-cfg	awards	7.75259
-fdo-lambda-eta-expansion	fasta	24.2686	-fblock-layout-cfg	gg	1.98570
-fdo-lambda-eta-expansion	callback001	-2.2609	-fblock-layout-cfg	parser	5.01509
-fdo-lambda-eta-expansion	ansi	1.4267	-fblock-layout-cfg	cryptarithm1	2.60715
-fdo-lambda-eta-expansion	awards	13.4524	-fblock-layout-cfg	life	-2.13130
-fdo-lambda-eta-expansion	treejoin	1.0377	-fblock-layout-cfg	power	-1.16242
-fdo-lambda-eta-expansion	queens	-1.3459	-fblock-layout-cfg	queens	7.72800
-ffloat-in	hash	3.3925	-fblock-layout-cfg	parfib	-3.03857
-ffloat-in	bernouilli	1.9755	-fblock-layout-cfg	blackscholes	-2.29742
-ffloat-in	gen_regexp	12.9157	-fblock-layout-cfg	coins	-1.70281
-ffloat-in	integrate	3.4231	-fstg-lift-lams	cacheprof	1.83221
-ffloat-in	real_cacheprof	1.4451	-fstg-lift-lams	hash	-1.52405
-ffloat-in	binary-trees	3.3045	-fstg-lift-lams	bernouilli	2.04286
-ffloat-in	fasta	54.3607	-fstg-lift-lams	gen_regexp	-5.92576
-ffloat-in	callback001	-1.4583	-fstg-lift-lams	integrate	1.73643
-ffloat-in	ansi	1.6146	-fstg-lift-lams	real_cacheprof	2.25207
-ffloat-in	awards	13.3587	-fstg-lift-lams	binary-trees	0.92690
-ffloat-in	cryptarithm1	1.1451	-fstg-lift-lams	fasta	56.69193
-ffloat-in	power	1.1328	-fstg-lift-lams	pidigits	-5.04631
-ffloat-in	parfib	1.8767	-fstg-lift-lams	callback001	-2.36930
-ffloat-in	coins	2.8462	-fstg-lift-lams	ansi	1.67093
-ffull-laziness	cacheprof	-1.6199	-fstg-lift-lams	awards	9.72330
-ffull-laziness	hash	1.7155	-fstg-lift-lams	gg	-1.30422
-ffull-laziness	bernouilli	3.7034	-fstg-lift-lams	cryptarithm1	-4.52144
-ffull-laziness	gen_regexp	3.4911	-fstg-lift-lams	life	2.11081
-ffull-laziness	integrate	2.5172	-fstg-lift-lams	power	-4.56735

-ffull-laziness	anna	-1.4628	-fstg-lift-lams	treejoin	1.93266
-ffull-laziness	real_cacheprof	2.3057	-fstg-lift-lams	queens	-14.33892
-ffull-laziness	compress2	1.0493	-fstg-lift-lams	blackscholes	-1.38373
-ffull-laziness	binary-trees	3.0988	-fstg-lift-lams	clausify	-2.51479

Table A.2: Effect sizes of the energy consumption of all significantly different benchmark-flag combinations

A.2 Execution Time

A.2.1 Charts

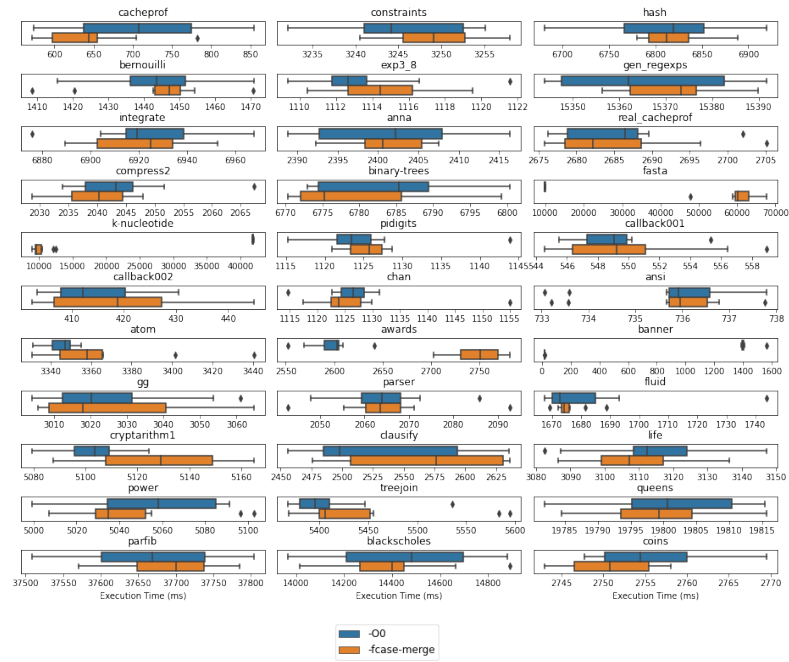
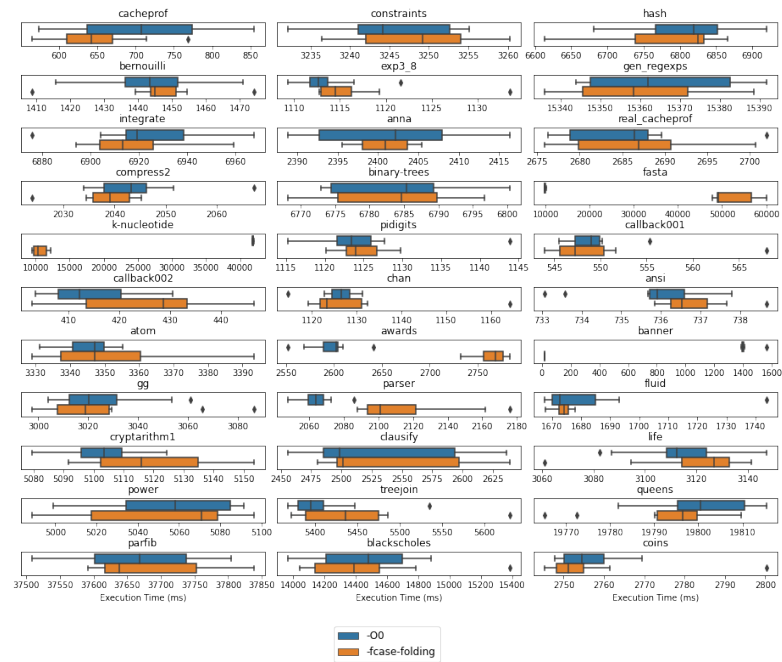
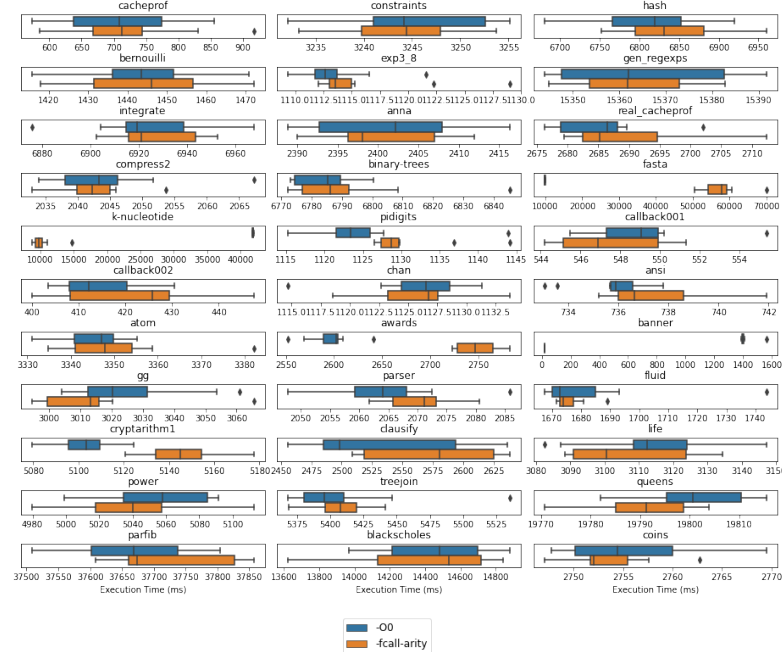
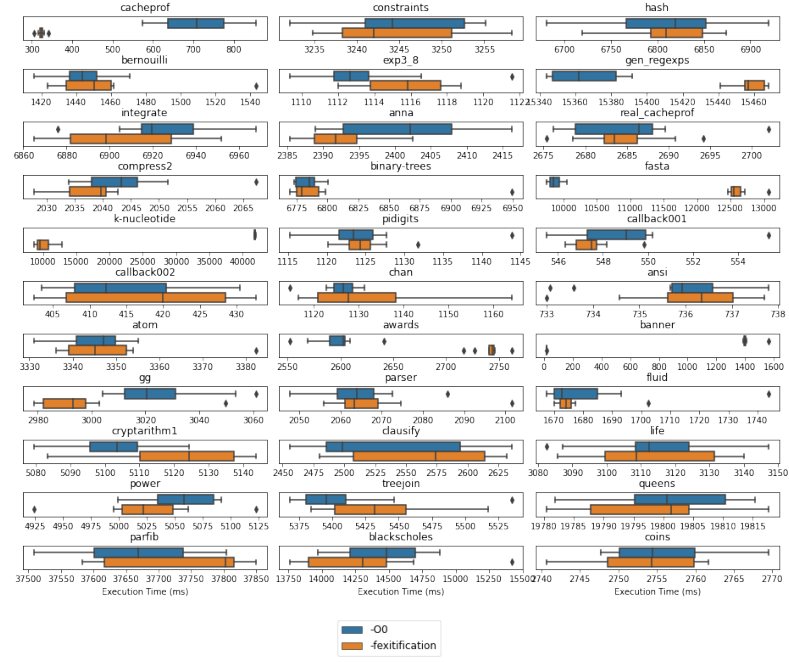
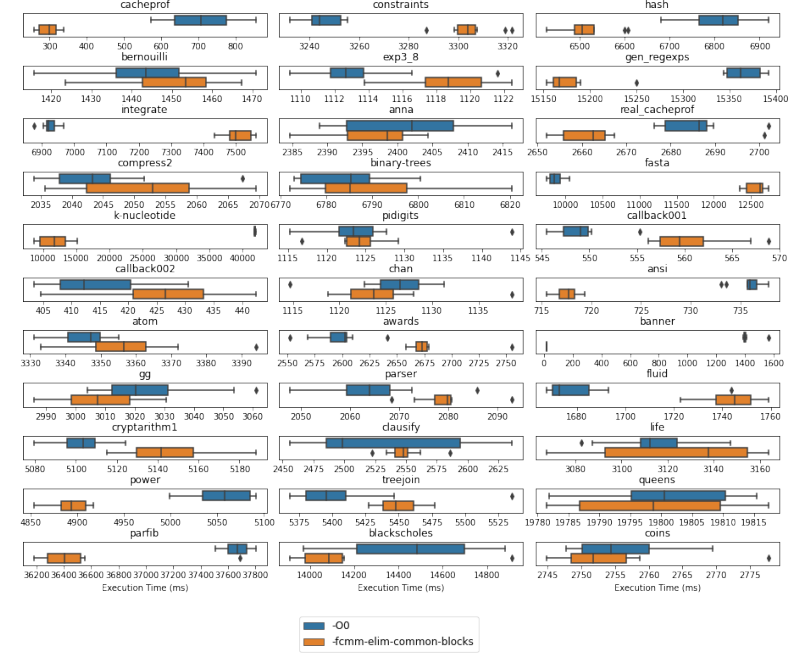
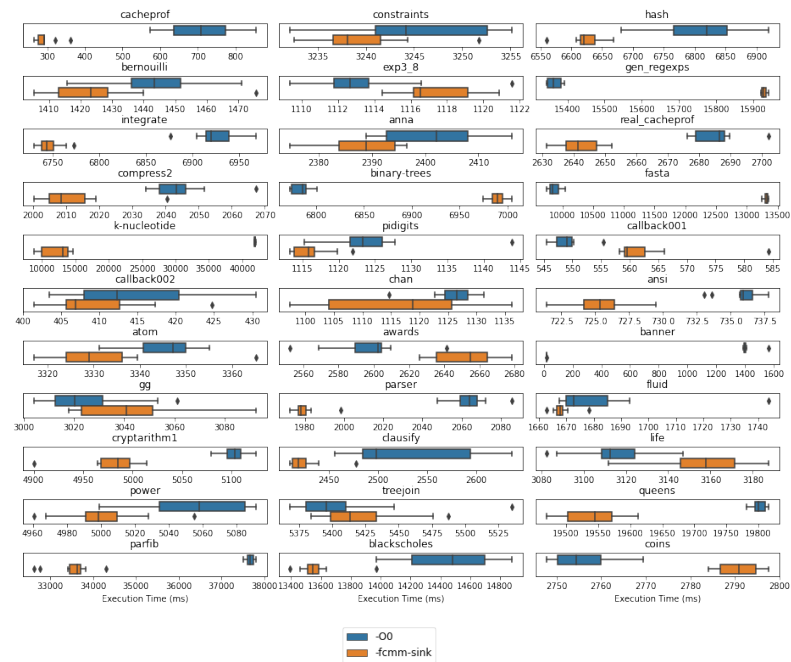
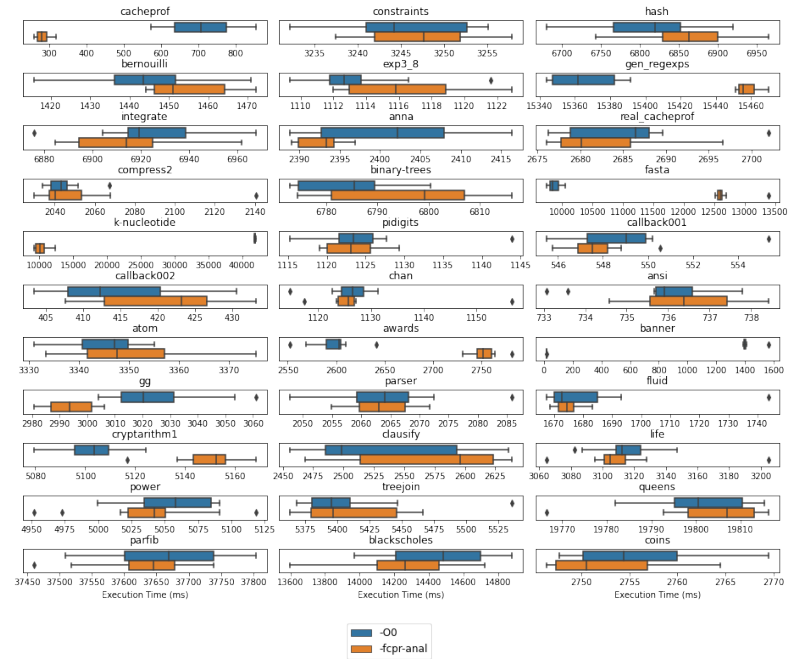
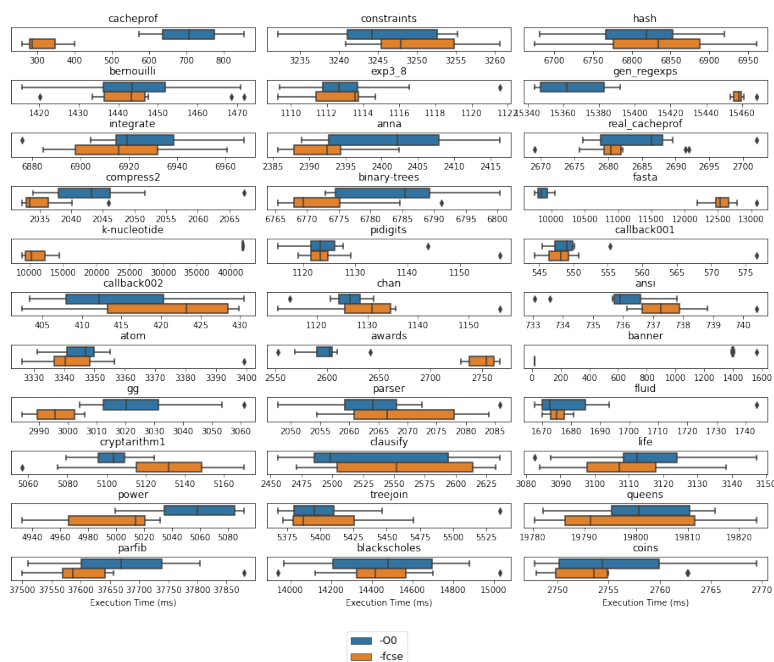
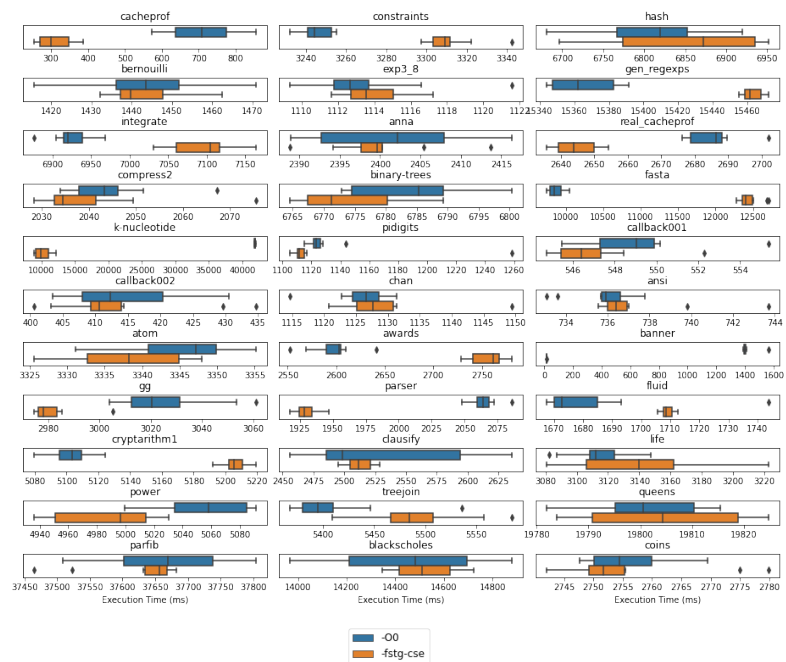


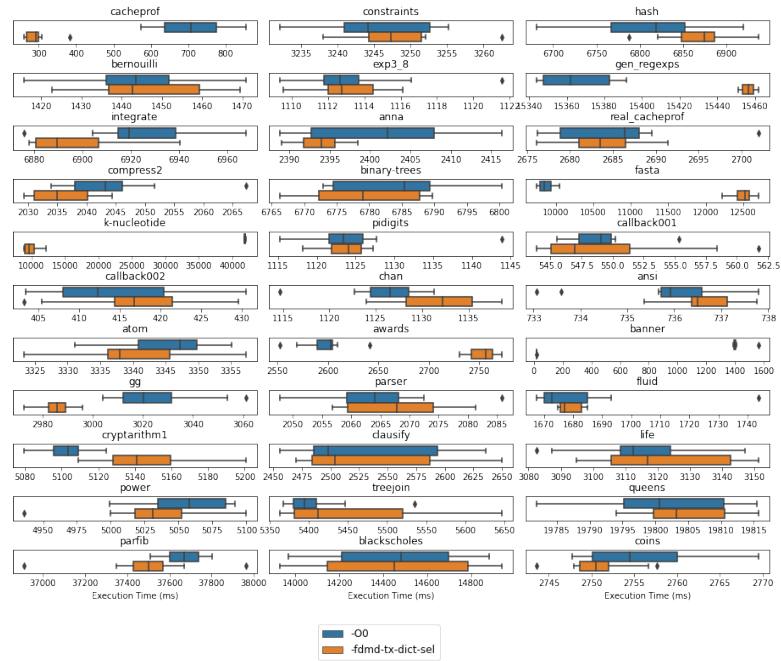
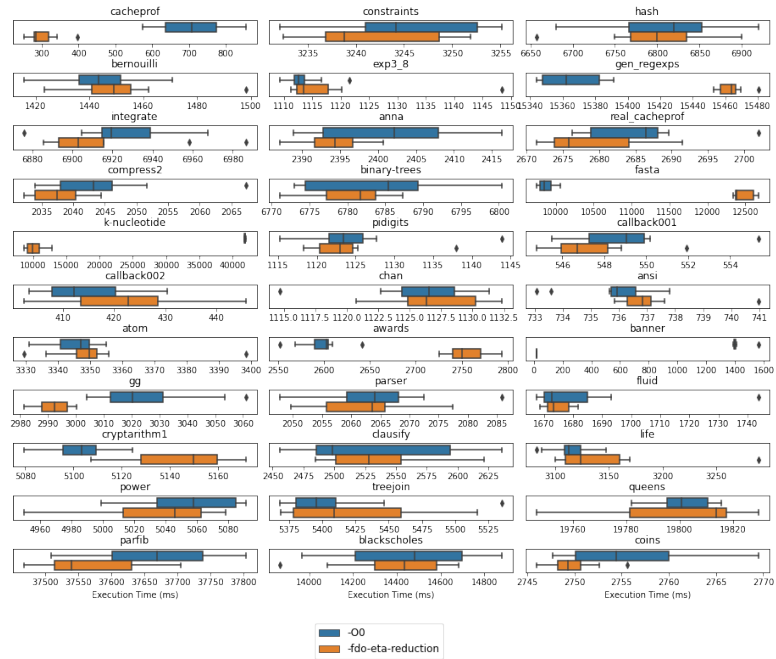
Figure A.27: Execution time of all benchmarks when compiled with `-O0` and `-fcase-merge`

Figure A.28: Execution time of all benchmarks when compiled with `-O0` and `-fcase-folding`Figure A.29: Execution time of all benchmarks when compiled with `-O0` and `-fcall-arity`

Figure A.30: Execution time of all benchmarks when compiled with `-O0` and `-fexitification`Figure A.31: Execution time of all benchmarks when compiled with `-O0` and `-fcmm-elim-common-blocks`

Figure A.32: Execution time of all benchmarks when compiled with `-O0` and `-fcmm-sink`Figure A.33: Execution time of all benchmarks when compiled with `-O0` and `-fcpr-anal`

Figure A.34: Execution time of all benchmarks when compiled with `-O0` and `-fcse`Figure A.35: Execution time of all benchmarks when compiled with `-O0` and `-fstg-cse`

Figure A.36: Execution time of all benchmarks when compiled with `-O0` and `-fdmd-tx-dict-sel`Figure A.37: Execution time of all benchmarks when compiled with `-O0` and `-fdo-eta-reduction`

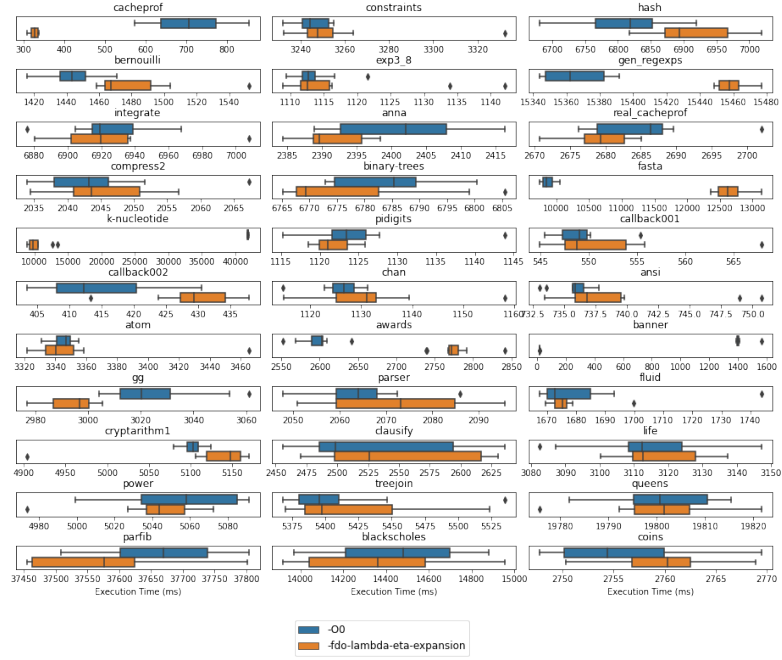


Figure A.38: Execution time of all benchmarks when compiled with `-O0` and `-fdo-lambda-eta-expansion`

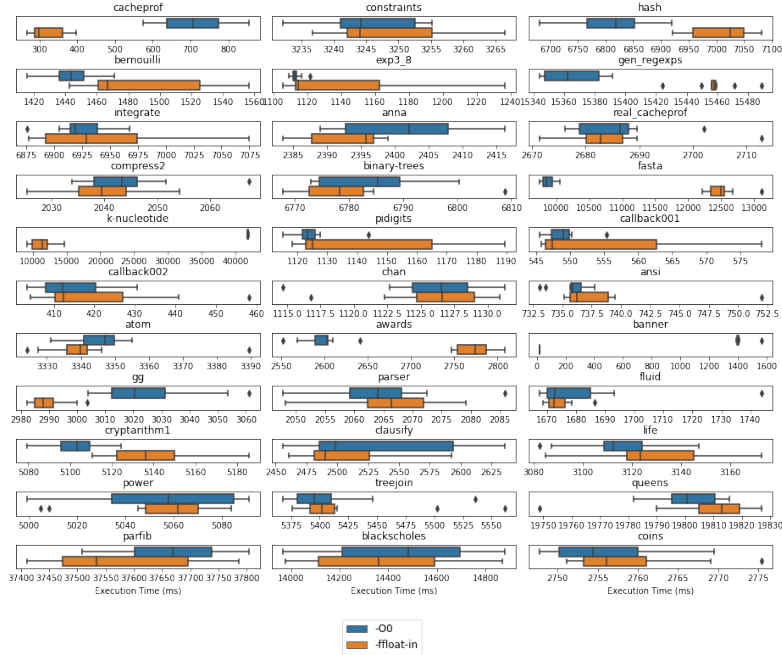
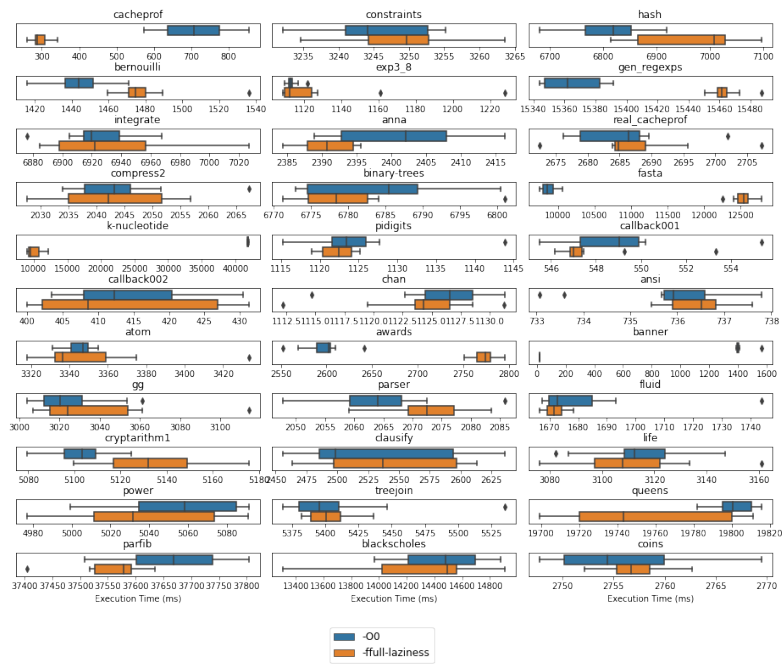
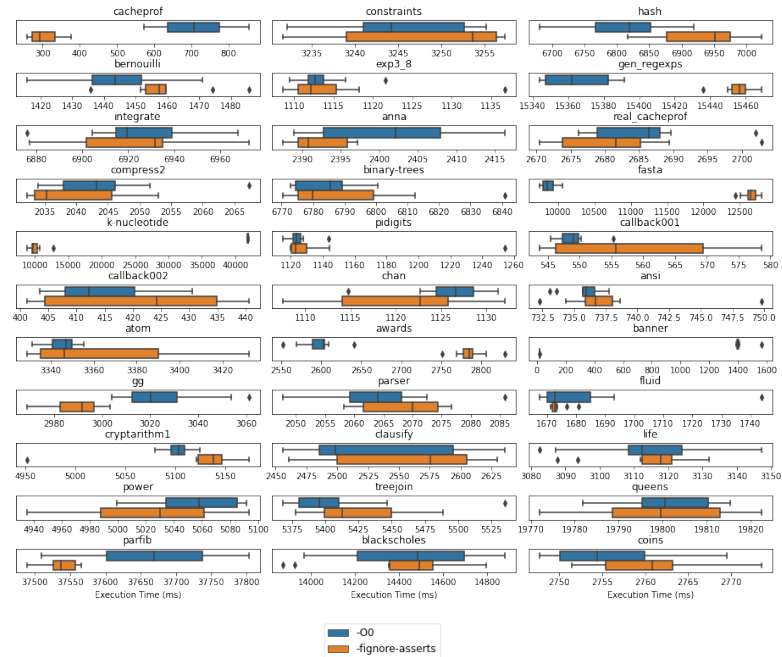
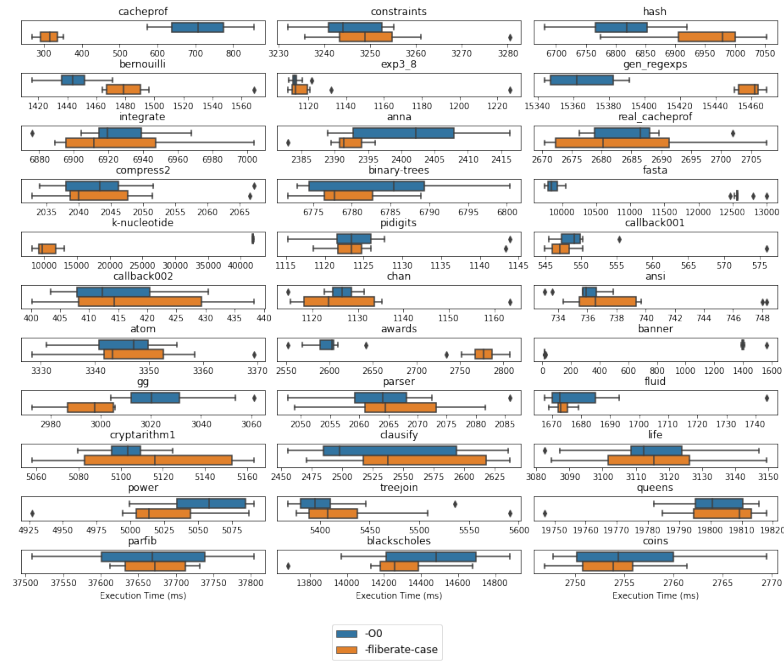
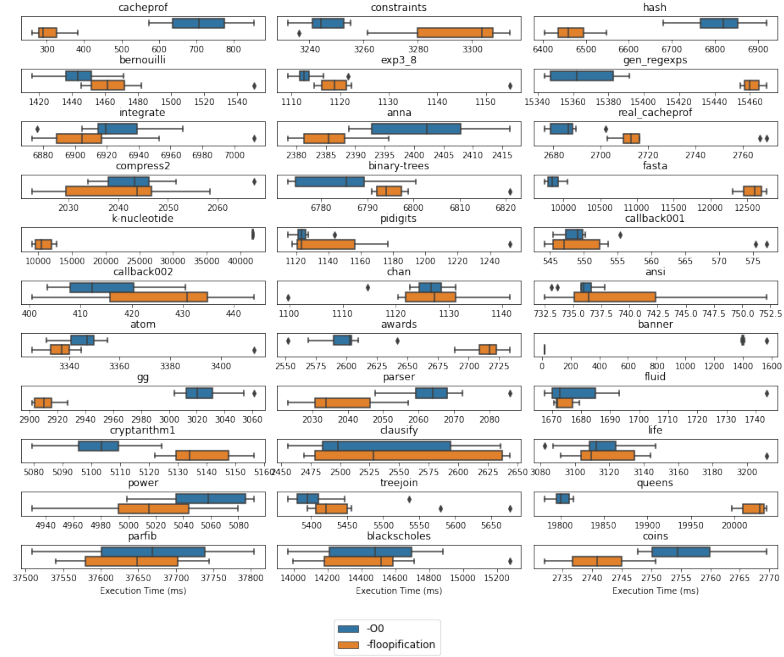


Figure A.39: Execution time of all benchmarks when compiled with `-O0` and `-ffloat-in`

Figure A.40: Execution time of all benchmarks when compiled with `-O0` and `-full-laziness`Figure A.41: Execution time of all benchmarks when compiled with `-O0` and `-fignore-asserts`

Figure A.42: Execution time of all benchmarks when compiled with `-O0` and `-fliberate-case`Figure A.43: Execution time of all benchmarks when compiled with `-O0` and `-floppification`

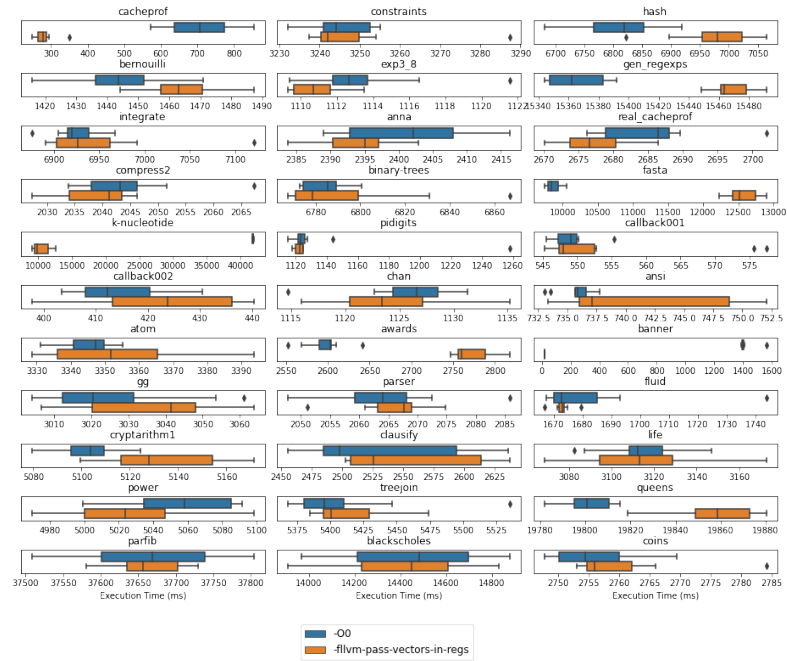


Figure A.44: Execution time of all benchmarks when compiled with `-O0` and `-fllvm-pass-vectors-in-regs`

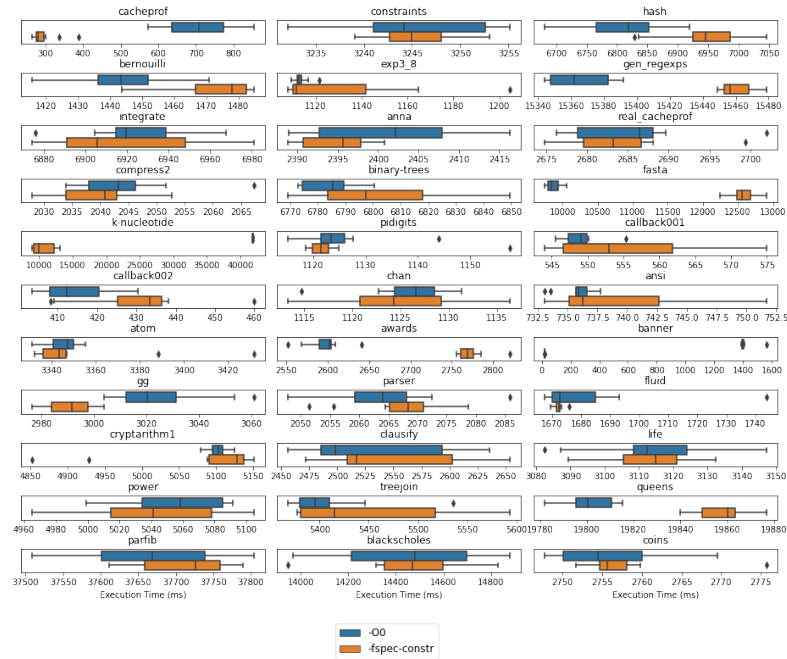
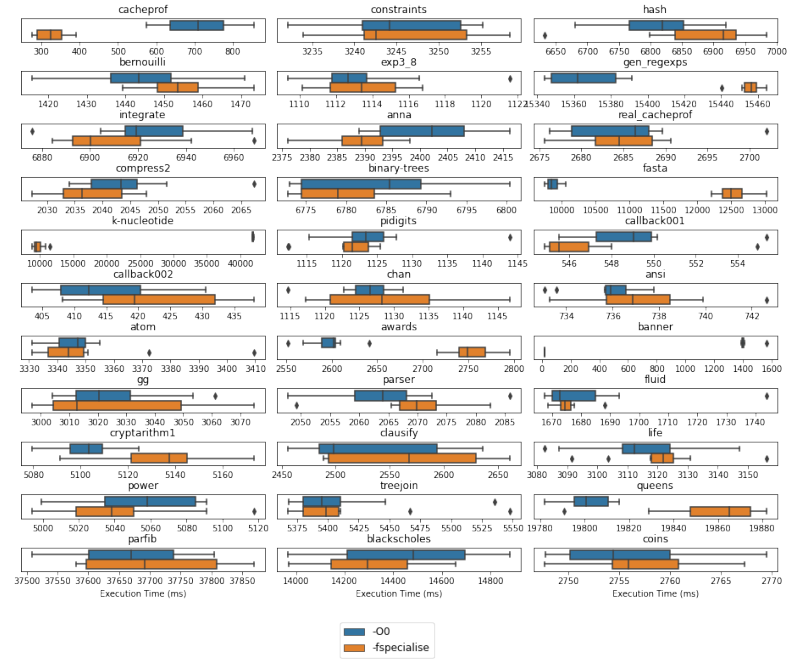
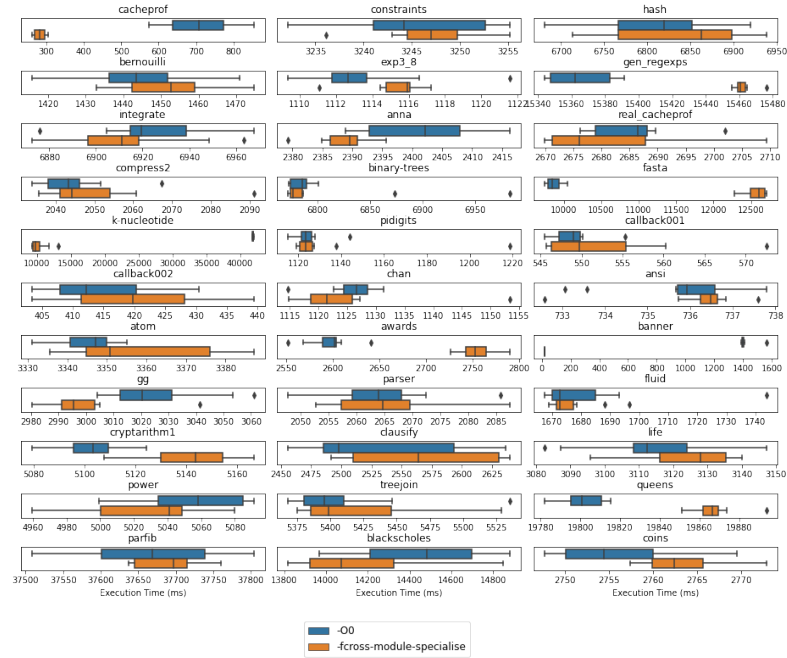


Figure A.45: Execution time of all benchmarks when compiled with `-O0` and `-fspec-constr`

Figure A.46: Execution time of all benchmarks when compiled with `-O0` and `-fspecialise`Figure A.47: Execution time of all benchmarks when compiled with `-O0` and `-fcross-module-specialise`

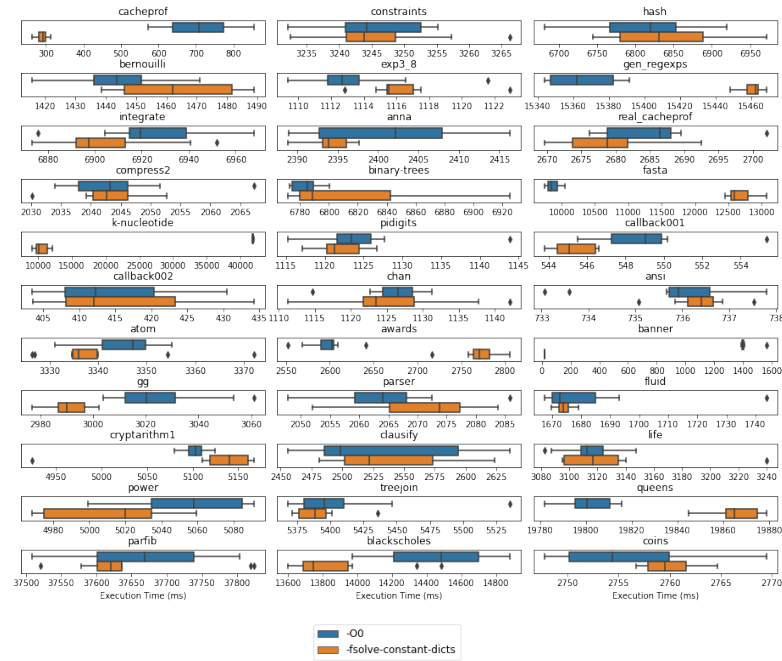


Figure A.48: Execution time of all benchmarks when compiled with `-O0` and `-fsolve-constant-dicts`

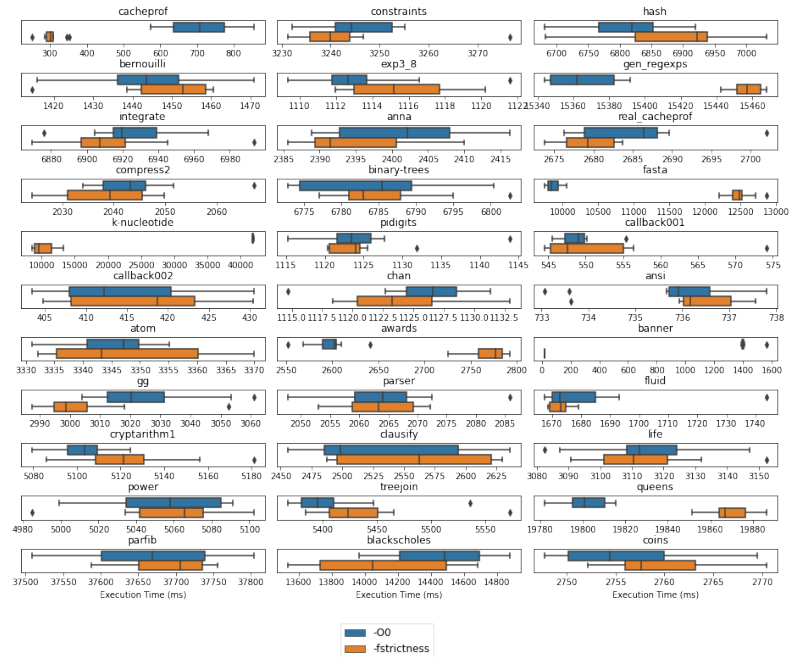


Figure A.49: Execution time of all benchmarks when compiled with `-O0` and `-fstrictness`

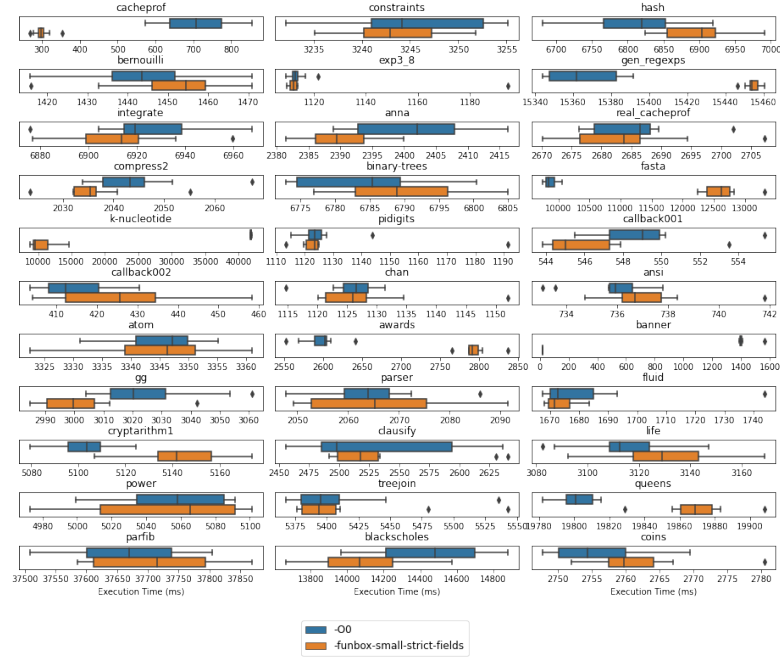


Figure A.50: Execution time of all benchmarks when compiled with `-O0` and `-funbox-small-strict-fields`

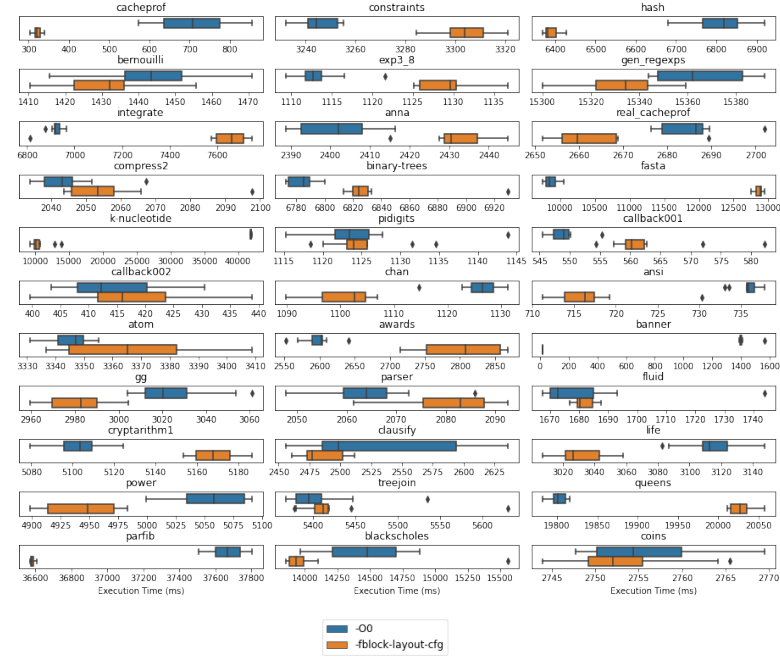
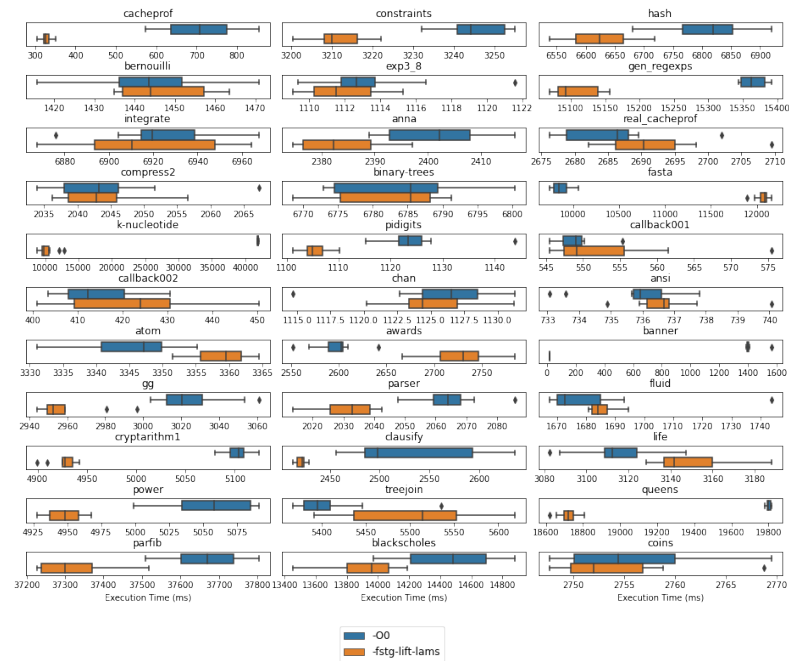


Figure A.51: Execution time of all benchmarks when compiled with `-O0` and `-fblock-layout-cfg`

Figure A.52: Execution time of all benchmarks when compiled with `-O0` and `-fstg-lift-lams`

A.2.2 Statistical Tests

Flag	$\rho < 0.05$		$\rho > 0.05$		Did not respect assumptions	
	Count	Programs	Count	Programs	Count	Programs
-fcase-merge	3	[fasta, awards, cryptarithm1]	25	[cacheprof, constraints, hash, bernouilli, exp3_8, gen_regexp, integrate, anna, real_cacheprof, compress2, binary-trees, pidigits, callback001, callback002, chan, gg, parser, life, power, queens, parfib, blackscholes, coins, fluid, clausify]	5	[k-nucleotide, ansi, atom, banner, treejoin]
-fcase-folding	2	[awards, cryptarithm1]	19	[cacheprof, constraints, hash, gen_regexp, integrate, anna, real_cacheprof, compress2, binary-trees, pidigits, callback002, ansi, atom, power, queens, parfib, blackscholes, coins, clausify]	12	[bernouilli, exp3_8, fasta, k-nucleotide, callback001, chan, banner, gg, parser, fluid, life, treejoin]

-fcall-arity	7	[fasta, ansi, awards, gg, parser, cryptarithm1, queens]	17	[cacheprof, constraints, hash, bernouilli, gen_regexp, integrate, anna, real_cacheprof, compress2, callback001, callback002, chan, atom, power, blackscholes, coins, fluid]	9	[exp3_8, binary-trees, k-nucleotide, pidigits, banner, clausify, life, treejoin, parfib]
-fexitification	8	[cacheprof, gen_regexp, anna, compress2, fasta, awards, gg, cryptarithm1]	18	[constraints, hash, bernouilli, exp3_8, integrate, real_cacheprof, binary-trees, pidigits, callback001, callback002, chan, ansi, life, power, queens, blackscholes, coins, fluid]	7	[k-nucleotide, atom, banner, parser, clausify, treejoin, parfib]
-fcmm-elim-common-blocks	14	[cacheprof, constraints, hash, exp3_8, integrate, fasta, callback001, callback002, ansi, gg, cryptarithm1, power, parfib, blackscholes]	9	[bernouilli, anna, compress2, binary-trees, pidigits, atom, life, queens, coins]	10	[gen_regexp, real_cacheprof, k-nucleotide, chan, awards, banner, parser, fluid, clausify, treejoin]

-fcmm-sink	22	[cacheprof, hash, bernouilli, exp3_8, gen_regexps, integrate, anna, real_cacheprof, binary-trees, fasta, pidigits, chan, ansi, atom, awards, life, power, queens, parfib, blackscholes, coins, clausify]	3	[constraints, callback002, gg]	8	[compress2, k-nucleotide, callback001, banner, parser, fluid, cryptarithm1, treejoin]
-fcpr-anal	6	[cacheprof, gen_regexps, anna, awards, gg, cryptarithm1]	17	[constraints, hash, bernouilli, exp3_8, integrate, real_cacheprof, binary-trees, pidigits, callback001, callback002, ansi, atom, parser, power, parfib, blackscholes, coins]	10	[compress2, fasta, k-nucleotide, chan, banner, fluid, clausify, life, treejoin, queens]
-fcse	7	[cacheprof, gen_regexps, anna, fasta, ansi, awards, gg]	15	[constraints, hash, bernouilli, integrate, pidigits, callback001, callback002, chan, parser, cryptarithm1, life, queens, blackscholes, coins, treejoin]	11	[exp3_8, real_cacheprof, compress2, binary-trees, k-nucleotide, atom, banner, fluid, clausify, power, parfib]

-fstg-cse	13	[cacheprof, constraints, gen_regexps, integrate, real_cacheprof, compress2, binary-trees, fasta, pidigits, awards, parser, cryptarithm1, power]	8	[hash, bernouilli, exp3_8, anna, callback002, atom, life, blackscholes]	12	[k-nucleotide, callback001, chan, ansi, banner, gg, fluid, clausify, treejoin, queens, parfib, coins]
-fdmd-tx-dict-sel	10	[cacheprof, hash, gen_regexps, integrate, compress2, fasta, chan, awards, gg, cryptarithm1]	19	[constraints, bernouilli, exp3_8, anna, real_cacheprof, binary-trees, pidigits, callback002, ansi, atom, parser, life, power, queens, parfib, blackscholes, coins, fluid, treejoin]	4	[k-nucleotide, callback001, banner, clausify]
-fdo-eta-reduction	7	[cacheprof, gen_regexps, awards, gg, cryptarithm1, parfib, coins]	17	[constraints, hash, bernouilli, integrate, anna, real_cacheprof, compress2, binary-trees, callback001, callback002, chan, atom, parser, life, power, queens, blacksholes]	9	[exp3_8, fasta, k-nucleotide, pidigits, ansi, banner, fluid, clausify, treejoin]

-fdo-lambda-eta-expansion	9	[cacheprof, hash, gen_regexps, anna, fasta, callback002, awards, gg, cryptarithm1]	13	[integrate, real_cacheprof, compress2, pidigits, chan, atom, parser, life, queens, parfib, blackscholes, coins, fluid]	11	[constraints, bernouilli, exp3_8, binary-trees, k-nucleotide, callback001, ansi, banner, clausify, power, treejoin]
-ffloat-in	8	[cacheprof, hash, gen_regexps, fasta, atom, awards, gg, cryptarithm1]	14	[constraints, integrate, anna, real_cacheprof, compress2, chan, ansi, parser, life, power, parfib, blacksc-holes, coins, treejoin]	11	[bernouilli, exp3_8, binary-trees, k-nucleotide, pidigits, callback001, callback002, banner, fluid, clausify, queens]
-ffull-laziness	11	[cacheprof, bernouilli, gen_regexps, anna, fasta, callback001, awards, parser, cryptarithm1, queens, parfib]	12	[constraints, integrate, real_cacheprof, compress2, pidigits, chan, ansi, gg, life, power, blacksc-holes, coins]	10	[hash, exp3_8, binary-trees, k-nucleotide, callback002, atom, banner, fluid, clausify, treejoin]
-fignore-asserts	10	[cacheprof, hash, bernouilli, gen_regexps, anna, fasta, awards, gg, cryptarithm1, parfib]	12	[exp3_8, integrate, real_cacheprof, chan, ansi, parser, life, power, queens, blackscholes, coins, fluid]	11	[constraints, compress2, binary-trees, k-nucleotide, pidigits, callback001, callback002, atom, banner, clausify, treejoin]

-fliberate-case	6	[cacheprof, hash, gen_regexps, awards, gg]	16	[constraints, integrate, real_cacheprof, compress2, binary-trees, callback001, callback002, chan, atom, parser, cryptarithm1, life, power, parfib, blackscholes, coins]	11	[bernouilli, exp3_8, fasta, k-nucleotide, pidigits, ansi, banner, fluid, clausify, treejoin, queens]
-floopification	17	[cacheprof, constraints, hash, bernouilli, exp3_8, gen_regexps, anna, real_cacheprof, fasta, atom, awards, gg, parser, cryptarithm1, power, queens, coins]	10	[compress2, callback001, callback002, chan, ansi, life, parfib, blackscholes, fluid, clausify]	6	[integrate, binary-trees, k-nucleotide, pidigits, banner, treejoin]
-flvm-pass-vectors-in-regs	9	[cacheprof, hash, bernouilli, gen_regexps, real_cacheprof, fasta, awards, cryptarithm1, queens]	16	[exp3_8, integrate, anna, compress2, pidigits, callback001, callback002, chan, atom, gg, parser, life, power, parfib, blacksholes, clausify]	8	[constraints, binary-trees, k-nucleotide, ansi, banner, fluid, treejoin, coins]

-fspec-constr	9	[cacheprof, hash, gen_regexps, fasta, callback002, awards, gg, cryptarithm1, queens]	14	[constraints, integrate, anna, real_cacheprof, compress2, binary-trees, callback001, chan, parser, life, power, parfib, blacksc-holes, treejoin]	10	[bernouilli, exp3_8, k-nucleotide, pidigits, ansi, atom, banner, fluid, clausify, coins]
-fspecialise	9	[cacheprof, hash, gen_regexps, anna, fasta, awards, parser, cryptarithm1, queens]	19	[constraints, bernouilli, exp3_8, integrate, real_cacheprof, compress2, binary-trees, pidigits, callback002, chan, ansi, atom, life, power, parfib, blackscholes, coins, clausify, treejoin]	5	[k-nucleotide, callback001, banner, gg, fluid]
-fcross-module-specialise	9	[cacheprof, anna, fasta, chan, awards, gg, cryptarithm1, queens, coins]	13	[constraints, hash, bernouilli, integrate, compress2, callback002, parser, life, power, parfib, blackscholes, fluid, treejoin]	11	[exp3_8, gen_regexps, real_cacheprof, binary-trees, k-nucleotide, pidigits, callback001, ansi, atom, banner, clausify]
-fsolve-constant-dicts	12	[cacheprof, bernouilli, gen_regexps, fasta, callback001, awards, gg, parser, cryptarithm1, power, queens, coins]	11	[constraints, hash, exp3_8, integrate, anna, real_cacheprof, compress2, pidigits, callback002, chan, ansi]	10	[binary-trees, k-nucleotide, atom, banner, fluid, clausify, life, treejoin, parfib, blackscholes]

-fstrictness	7	[cacheprof, gen_regexp, real_cacheprof, fasta, awards, queens, blacksholes]	17	[hash, exp3_8, integrate, anna, compress2, binary-trees, callback002, chan, atom, parser, cryptarithm1, life, power, parfib, coins, clausify, treejoin]	9	[constraints, bernouilli, k-nucleotide, pidigits, callback001, ansi, banner, gg, fluid]
-funbox-small-strict-fields	10	[cacheprof, hash, gen_regexp, anna, fasta, gg, cryptarithm1, queens, blacksholes, coins]	17	[constraints, bernouilli, exp3_8, integrate, real_cacheprof, compress2, binary-trees, callback002, chan, ansi, atom, parser, life, power, parfib, clausify, treejoin]	6	[k-nucleotide, pidigits, callback001, awards, banner, fluid]
-fblock-layout-cfg	19	[cacheprof, constraints, hash, exp3_8, gen_regexp, integrate, anna, real_cacheprof, fasta, ansi, atom, gg, parser, cryptarithm1, life, power, queens, parfib, blacksholes]	4	[bernouilli, pidigits, callback002, coins]	10	[compress2, binary-trees, k-nucleotide, callback001, chan, awards, banner, fluid, clausify, treejoin]

-fstg-lift-lams	16	[cacheprof, constraints, hash, gen_regexps, anna, fasta, pidigits, atom, awards, parser, cryptarithm1, life, power, queens, parfib, blacksc-holes]	10	[bernouilli, exp3_8, integrate, real_cacheprof, compress2, binary-trees, callback002, chan, ansi, coins]	7	[k-nucleotide, callback001, banner, gg, fluid, clausify, treejoin]
-----------------	----	--	----	--	---	--

Table A.3: Benchmarks that have a statistically significant difference in execution time when compared to their non-optimized versions ($\rho < 0.05$), that do not ($\rho > 0.05$), and that did not respect the assumptions of ANOVA, by optimization flag

Flag	Benchmark	Effect Size	Flag	Benchmark	Effect Size
-fcase-merge	fasta	21.6	-fignore-asserts	gen_regexps	6.37
-fcase-merge	awards	5.8	-fignore-asserts	anna	-1.08
-fcase-merge	cryptarithm1	1.1	-fignore-asserts	fasta	25.73
-fcase-folding	awards	7.8	-fignore-asserts	awards	8.03
-fcase-folding	cryptarithm1	0.9	-fignore-asserts	gg	-2.30
-fcall-arity	fasta	11.8	-fignore-asserts	cryptarithm1	2.44
-fcall-arity	ansi	0.9	-fignore-asserts	parfib	-1.70
-fcall-arity	awards	6.4	-fliberate-case	cacheprof	-5.62
-fcall-arity	gg	-1.2	-fliberate-case	hash	2.33
-fcall-arity	parser	1.1	-fliberate-case	gen_regexps	7.01
-fcall-arity	cryptarithm1	2.8	-fliberate-case	anna	-1.16

-fcall-arity	queens	-0.9	-fliberate-case	awards	8.74
-fexitification	cacheprof	-5.5	-fliberate-case	gg	-1.90
-fexitification	gen_regexps	6.6	-floopification	cacheprof	-5.55
-fexitification	anna	-0.9	-floopification	constraints	5.08
-fexitification	compress2	-0.9	-floopification	hash	-5.35
-fexitification	fasta	27.3	-floopification	bernouilli	1.12
-fexitification	awards	7.3	-floopification	exp3_8	1.46
-fexitification	gg	-2.2	-floopification	gen_regexps	7.28
-fexitification	cryptarithm1	1.0	-floopification	anna	-1.86
-fcmm-elim-common-blocks	cacheprof	-5.8	-floopification	real_cacheprof	3.60
-fcmm-elim-common-blocks	constraints	6.3	-floopification	fasta	19.18
-fcmm-elim-common-blocks	hash	-4.2	-floopification	atom	-1.66
-fcmm-elim-common-blocks	exp3_8	1.4	-floopification	awards	5.67
-fcmm-elim-common-blocks	integrate	16.0	-floopification	gg	-7.51
-fcmm-elim-common-blocks	fasta	21.3	-floopification	parser	-3.16
-fcmm-elim-common-blocks	callback001	3.0	-floopification	cryptarithm1	2.70
-fcmm-elim-common-blocks	callback002	1.1	-floopification	power	-0.96
-fcmm-elim-common-blocks	ansi	-13.0	-floopification	queens	16.02
-fcmm-elim-common-blocks	gg	-1.0	-floopification	coins	-2.20
-fcmm-elim-common-blocks	cryptarithm1	2.2	-flvm-pass-vectors-in-regs	cacheprof	-6.21
-fcmm-elim-common-blocks	power	-5.6	-flvm-pass-vectors-in-regs	hash	2.14
-fcmm-elim-common-blocks	parfib	-10.3	-flvm-pass-vectors-in-regs	bernouilli	1.24
-fcmm-elim-common-blocks	blackscholes	-1.6	-flvm-pass-vectors-in-regs	gen_regexps	6.78
-fcmm-sink	cacheprof	-6.0	-flvm-pass-vectors-in-regs	real_cacheprof	-1.01
-fcmm-sink	hash	-3.0	-flvm-pass-vectors-in-regs	fasta	14.86
-fcmm-sink	bernouilli	-1.6	-flvm-pass-vectors-in-regs	awards	7.00

-fcmm-sink	exp3_8	1.3	-flvm-pass-vectors-in-regs	cryptarithm1	1.53
-fcmm-sink	gen_regexps	39.7	-flvm-pass-vectors-in-regs	queens	3.22
-fcmm-sink	integrate	-8.4	-fspec-constr	cacheprof	-5.99
-fcmm-sink	anna	-1.3	-fspec-constr	hash	1.79
-fcmm-sink	real_cacheprof	-5.7	-fspec-constr	gen_regexps	6.22
-fcmm-sink	binary-trees	19.9	-fspec-constr	fasta	17.41
-fcmm-sink	fasta	43.4	-fspec-constr	callback002	1.21
-fcmm-sink	pidigits	-1.9	-fspec-constr	awards	8.71
-fcmm-sink	chan	-0.9	-fspec-constr	gg	-2.25
-fcmm-sink	ansi	-4.9	-fspec-constr	cryptarithm1	1.32
-fcmm-sink	atom	-1.3	-fspec-constr	queens	4.81
-fcmm-sink	awards	2.4	-fspecialise	cacheprof	-5.26
-fcmm-sink	life	1.4	-fspecialise	hash	1.33
-fcmm-sink	power	-1.7	-fspecialise	gen_regexps	6.70
-fcmm-sink	queens	-8.5	-fspecialise	anna	-1.30
-fcmm-sink	parfib	-19.9	-fspecialise	fasta	12.89
-fcmm-sink	blackscholes	-3.8	-fspecialise	awards	6.32
-fcmm-sink	coins	6.8	-fspecialise	parser	1.35
-fcmm-sink	clausify	-1.9	-fspecialise	cryptarithm1	1.55
-fcpr-anal	cacheprof	-6.2	-fspecialise	queens	2.44
-fcpr-anal	gen_regexps	6.9	-fcross-module-specialise	cacheprof	-6.25
-fcpr-anal	anna	-0.9	-fcross-module-specialise	anna	-1.49
-fcpr-anal	awards	7.5	-fcross-module-specialise	fasta	21.19
-fcpr-anal	gg	-2.0	-fcross-module-specialise	chan	-1.04
-fcpr-anal	cryptarithm1	3.3	-fcross-module-specialise	awards	6.92
-fcse	cacheprof	-5.3	-fcross-module-specialise	gg	-2.03

-fcse	gen_regexps	7.2	-fcross-module-specialise	cryptarithm1	2.57
-fcse	anna	-1.0	-fcross-module-specialise	queens	5.82
-fcse	fasta	17.8	-fcross-module-specialise	coins	1.62
-fcse	ansi	1.2	-fsolve-constant-dicts	cacheprof	-6.15
-fcse	awards	7.4	-fsolve-constant-dicts	bernouilli	1.03
-fcse	gg	-2.0	-fsolve-constant-dicts	gen_regexps	7.14
-fstg-cse	cacheprof	-5.3	-fsolve-constant-dicts	fasta	18.47
-fstg-cse	constraints	5.8	-fsolve-constant-dicts	callback001	-1.72
-fstg-cse	gen_regexps	7.4	-fsolve-constant-dicts	awards	8.47
-fstg-cse	integrate	4.6	-fsolve-constant-dicts	gg	-2.34
-fstg-cse	real_cacheprof	-5.1	-fsolve-constant-dicts	parser	0.95
-fstg-cse	compress2	-0.9	-fsolve-constant-dicts	cryptarithm1	2.20
-fstg-cse	binary-trees	-1.4	-fsolve-constant-dicts	power	-1.26
-fstg-cse	fasta	20.0	-fsolve-constant-dicts	queens	5.82
-fstg-cse	pidigits	-2.8	-fsolve-constant-dicts	coins	1.28
-fstg-cse	awards	7.2	-fstrictness	cacheprof	-5.75
-fstg-cse	parser	-14.7	-fstrictness	gen_regexps	6.42
-fstg-cse	cryptarithm1	9.7	-fstrictness	real_cacheprof	-0.93
-fstg-cse	power	-1.9	-fstrictness	fasta	15.91
-fdmd-tx-dict-sel	cacheprof	-6.0	-fstrictness	awards	7.39
-fdmd-tx-dict-sel	hash	1.1	-fstrictness	queens	5.95
-fdmd-tx-dict-sel	gen_regexps	7.1	-fstrictness	blackscholes	-0.93
-fdmd-tx-dict-sel	integrate	-0.9	-funbox-small-strict-fields	cacheprof	-5.33
-fdmd-tx-dict-sel	compress2	-1.0	-funbox-small-strict-fields	hash	1.27
-fdmd-tx-dict-sel	fasta	20.3	-funbox-small-strict-fields	gen_regexps	6.96
-fdmd-tx-dict-sel	chan	1.1	-funbox-small-strict-fields	anna	-1.19

-fdmd-tx-dict-sel	awards	7.7	-funbox-small-strict-fields	fasta	11.47
-fdmd-tx-dict-sel	gg	-2.7	-funbox-small-strict-fields	gg	-1.27
-fdmd-tx-dict-sel	cryptarithm1	2.1	-funbox-small-strict-fields	cryptarithm1	2.14
-fdo-eta-reduction	cacheprof	-5.8	-funbox-small-strict-fields	queens	3.96
-fdo-eta-reduction	gen_regexps	6.8	-funbox-small-strict-fields	blackscholes	-1.23
-fdo-eta-reduction	awards	6.3	-funbox-small-strict-fields	coins	0.99
-fdo-eta-reduction	gg	-2.3	-fblock-layout-cfg	cacheprof	-5.73
-fdo-eta-reduction	cryptarithm1	2.3	-fblock-layout-cfg	constraints	6.05
-fdo-eta-reduction	parfib	-1.0	-fblock-layout-cfg	hash	-7.33
-fdo-eta-reduction	coins	-1.1	-fblock-layout-cfg	exp3_8	4.27
-fdo-lambda-eta-expansion	cacheprof	-5.6	-fblock-layout-cfg	gen_regexps	-1.69
-fdo-lambda-eta-expansion	hash	1.3	-fblock-layout-cfg	integrate	15.83
-fdo-lambda-eta-expansion	gen_regexps	6.6	-fblock-layout-cfg	anna	3.32
-fdo-lambda-eta-expansion	anna	-1.1	-fblock-layout-cfg	real_cacheprof	-2.20
-fdo-lambda-eta-expansion	fasta	14.0	-fblock-layout-cfg	fasta	33.63
-fdo-lambda-eta-expansion	callback002	1.6	-fblock-layout-cfg	ansi	-9.72
-fdo-lambda-eta-expansion	awards	7.6	-fblock-layout-cfg	atom	1.00
-fdo-lambda-eta-expansion	gg	-2.0	-fblock-layout-cfg	gg	-2.56
-fdo-lambda-eta-expansion	cryptarithm1	2.2	-fblock-layout-cfg	parser	2.07
-ffloat-in	cacheprof	-5.1	-fblock-layout-cfg	cryptarithm1	5.33
-ffloat-in	hash	2.9	-fblock-layout-cfg	life	-4.69
-ffloat-in	gen_regexps	4.7	-fblock-layout-cfg	power	-3.34
-ffloat-in	fasta	19.5	-fblock-layout-cfg	queens	17.08
-ffloat-in	atom	-1.4	-fblock-layout-cfg	parfib	-13.62
-ffloat-in	awards	7.2	-fblock-layout-cfg	blackscholes	-2.16
-ffloat-in	gg	-2.4	-fstg-lift-lams	cacheprof	-5.64

-ffloat-in	cryptarithm1	1.9	-fstg-lift-lams	constraints	-4.38
-ffull-laziness	cacheprof	-5.8	-fstg-lift-lams	hash	-2.66
-ffull-laziness	bernouilli	2.1	-fstg-lift-lams	gen_regexps	-8.69
-ffull-laziness	gen_regexps	6.6	-fstg-lift-lams	anna	-1.84
-ffull-laziness	anna	-1.2	-fstg-lift-lams	fasta	25.91
-ffull-laziness	fasta	20.7	-fstg-lift-lams	pidigits	-5.26
-ffull-laziness	callback001	-1.0	-fstg-lift-lams	atom	2.10
-ffull-laziness	awards	8.7	-fstg-lift-lams	awards	4.47
-ffull-laziness	parser	1.4	-fstg-lift-lams	parser	-3.44
-ffull-laziness	cryptarithm1	1.5	-fstg-lift-lams	cryptarithm1	-13.27
-ffull-laziness	queens	-1.3	-fstg-lift-lams	life	1.52
-ffull-laziness	parfib	-1.2	-fstg-lift-lams	power	-3.97
-fignore-asserts	cacheprof	-5.5	-fstg-lift-lams	queens	-27.25
-fignore-asserts	hash	1.5	-fstg-lift-lams	parfib	-3.07
-fignore-asserts	bernouilli	0.9	-fstg-lift-lams	blackscholes	-1.91

Table A.4: Effect sizes of the execution time of all significantly different benchmark-flag combinations