Compiling Haskell for Energy Efficiency

Empirical Analysis of Individual Transformations

Kirkeby, Maja Hanne; Santos, Bernardo; Fernandes, João Paulo; Pardo, Alberto

Published in:
39th Annual ACM Symposium on Applied Computing, SAC 2024

DOI:
10.1145/3605098.3635915

Publication date:
2024

Document Version
Peer reviewed version

Citation for published version (APA):
Kirkeby, M. H., Santos, B., Fernandes, J. P., & Pardo, A. (2024). Compiling Haskell for Energy Efficiency: Empirical Analysis of Individual Transformations. In *39th Annual ACM Symposium on Applied Computing, SAC 2024* (pp. 1104-1113). Association for Computing Machinery. https://doi.org/10.1145/3605098.3635915

This is an Accepted Manuscript of an article published by ACM in SAC '24: Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing on 29 May 2024, available at: https://doi.org/10.1145/3605098.3635915

Compiling Haskell for Energy Efficiency: Empirical Analysis of Individual Transformations

Bernardo Santos
up201706534@up.pt
University of Porto
Porto, Portugal

João Paulo Fernandes*
jpf9731@nyu.edu
New York University Abu Dhabi
Abu Dhabi, UAE

Maja H. Kirkeby†
kirkebym@acm.org
Roskilde University
Roskilde, Denmark

Alberto Pardo
pardo@fing.edu.uy
Instituto de Computación,
Universidad de la República
Montevideo, Uruguay

ABSTRACT

Energy efficiency is a growing concern for software developers. This empirical study investigates the impact of compiler optimizations on energy efficiency in Haskell programs compiled using the Glasgow Haskell Compiler (GHC). We focus on GHC's -O2 optimization series and explore the effects of selectively disabling individual optimizations using -fno-* flags, alongside variations in initial execution temperatures.

We examined 25 GHC optimizations across 18 benchmarks from the NoFib Haskell Benchmark Suite resulting in 468 combinations of benchmark and optimization-deactivations. Data was collected at three starting temperatures (45°C, 55°C, and 65°C), resulting in 40 samples per benchmark-optimization combination. Our key metrics included energy consumption and execution time.

Considering all combinations, for 24% of the individual optimizations provided, when disabled, a significant increase in energy consumption, i.e., enabling these optimizations resulted in more energy-efficient executables, whereas for 26% the optimization provided a significant increase in time, when disabled. However, only for 12% of all the combinations, the disabling increased both time and energy consumption significantly, and in 5% of all the combinations, we observed opposite impacts on time and energy.

We found that 10 optimizations produced equally or more energy-efficient executables for all the benchmarks, whereas only one compiler optimization produced a better or equally performing executable for all the benchmarks.

As a secondary finding, we explored the influence of initial temperatures on energy consumption. While programs that started at 45°C showed the least variance in terms of both energy consumption and wall time, those started at 55°C tended to exhibit lower energy consumption for the typical program compared to those started at 45°C or 65°C.

CCS CONCEPTS

• **Software and its engineering → Compilers**; **Software performance**; **Software design tradeoffs**; **General programming languages**.

KEYWORDS

Program Transformations, Energy Consumption, Compiler Optimizations, Programming Languages, Functional Programming

ACM Reference Format:
Bernardo Santos, João Paulo Fernandes, Maja H. Kirkeby, and Alberto Pardo. 2024. Compiling Haskell for Energy Efficiency: Empirical Analysis of Individual Transformations. In *The 39th ACM/SIGAPP Symposium on Applied Computing (SAC '24), April 8–12, 2024, Avila, Spain.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605098.3635915

1 INTRODUCTION

The world's energy consumption has been steadily rising for over seven decades, and the great majority of the energy we consume comes from non-renewable sources [28]. The Information and Communications Technology (ICT) sector was estimated to be responsible for over 7% of the energy consumed in 2020 [3], a figure that only under optimistic scenarios is not expected to grow in the upcoming years.

One approach to reducing software's energy consumption is for the software developers to design and develop more energy-efficient code. However, a study has shown that only 18% of software developers consider energy consumption when developing software, and even fewer (14%) consider it a requirement [23]. This lack of consideration can be attributed to programmers' inconsistent knowledge about the energy consumption of software and their difficulty in finding adequate answers to their questions about this topic [27].

An alternative approach is to create compilers that optimize for energy. This approach has a higher potential for scaling and reaching practical benefits, as it can centralize the knowledge of producing energy-efficient code at the compiler level, benefiting

*On a leave from the Faculty of Engineering of the University of Porto, Porto, Portugal
†This work is partly supported by the Independent Research Fund Denmark Project no. 2102-00281B.

SAC '24, April 8–12, 2024, Avila, Spain

ACM ISBN 979-8-4007-0243-3/24/04…$15.00
https://doi.org/10.1145/3605098.3635915

all its users. The first step towards this approach is to determine whether an existing compiler can already generate energy-efficient executables. In this paper, we focus on the Haskell programming language and the Glasgow Haskell Compiler (GHC) to investigate this possibility.

Haskell was chosen for its purity and laziness in functional languages, which have been proven to increase modularity and facilitate compile-time optimizations [4, 9, 12]. In addition, Haskell is currently the most popular purely functional lazy language[1] and is used in production by more than 200 companies [5, 11, 16]. In this study we chose GHC, the most used Haskell compiler; it is open-source, is being actively developed, and has an extensive catalog of individually accessible optimizations [8, 17]. If GHC is capable of generating energy-efficient code, developers can write software as usual and rely on the compiler to produce energy-optimized versions.

Ultimately, we address the following two main research questions:

RQ$_1$ How do GHC optimizations influence the energy efficiency of programs?

RQ$_2$ Do GHC optimizations influence the execution time of programs in the same way they influence energy consumption?

Contributions. This work contributes new evidence for how GHC's compilation optimizations influence the energy and time consumption of Haskell programs, both from a global perspective and individually for each optimization. The study contributes to the incorporation of software energy efficiency into compilers. We also provide a framework that can be used to automate similar studies, along with public data for further analysis.

The remainder of this paper is organized as follows. In Section 2, we describe the methodology we adopted. Section 3 presents and discusses the obtained results. Section 4 addresses threats to the validity of our findings. Section 5 discusses related works, and Section 6 concludes the paper and suggests future research directions.

2 METHOD

To properly answer the proposed research questions, we developed the following methodology, taking into consideration the reviewed literature that we describe in Section 5. Firstly, we selected the 25 optimizations, i.e., optimization flags, that compose GHC's `-O2` series as our study object and sampled a set of 18 benchmarks from the NoFib Haskell Benchmark Suite. We considered three different starting temperatures for the programs' execution (45ºC, 55ºC, and 65ºC), and for each, we collected 40 samples of each benchmark-flag combination, in batches of 20 samples. Lastly, we performed a rigorous statistical analysis of the collected metrics, namely energy consumption and execution time.

The framework developed to perform the experiment, the statistical tests, and all data collected are publicly available in a public repository[2].

[1] https://survey.stackoverflow.co/2022#section-most-loved-dreaded-and-wanted-programming-scripting-and-markup-languages, accessed 22/09/2023

[2] https://github.com/bernas670/ghc-energy

2.1 Selecting Compiler Optimizations

Much like other compilers, GHC [31] offers three predefined series of optimizations 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 optimizations that cannot make run time or space worse, even if it means significantly longer compile times

In our research, we deviate from the approach of enabling optimizations individually. Instead, we apply the `-O2` series of optimizations, while selectively deactivating specific optimizations using the `-fno-*` flags. We adopt this methodology, which we have discussed and validated with an expert on the GHC compiler, due to the significant impact of the order in which optimization flags are applied on the compilation results. Notably, the activation or deactivation of a particular optimization can potentially create optimization prospects for subsequent flags in the sequence. The `-funfolding-use-threshold=<n>` flag is the only exception, which sets a cut-off size for function unfolding (also known as inlining) during compilation, where functions smaller than the specified threshold will be unfolded at the call site, while larger functions will not be unfolded. For this option, we are considering the default threshold (80), double the value (160), and half of it (40). A comprehensive list of the considered flags is presented in Table 1.

Flag	Flag
-fno-case-merge	-fno-float-in
-fno-case-folding	-fno-full-laziness
-fno-call-arity	-fno-ignore-asserts
-fno-exitification	-fno-loopification
-fno-cmm-elim-common-blocks	-fno-specialise
-fno-cmm-sink	-fno-solve-constant-dicts
-fno-block-layout-cfg	-fno-stg-lift-lams
-fno-cpr-anal	-fno-strictness
-fno-cse	-fno-unbox-small-strict-fields
-fno-stg-cse	-fno-spec-constr
-fno-dmd-tx-dict-sel	-fno-liberate-case
-fno-do-eta-reduction	-funfolding-use-threshold=<n>
-fno-do-lambda-eta-expansion	

Table 1: GHC flags that were individually deactivated (starting with -fno) and activated (the remaining one)

2.2 Sampling Benchmark Programs

The programs selected for this study are a part of the NoFib Haskell Benchmark Suite[3]. This suite is composed of more than one hundred benchmarks which are divided into seven categories; they cover benchmarks from actual command-line applications to algorithms like Fast Fourier Transform, as well as benchmarks from The Computer Language Benchmark Game (CLBG)[4].

[3] https://gitlab.haskell.org/ghc/nofib, accessed 22/09/2023

[4] https://benchmarksgame-team.pages.debian.net/benchmarksgame/, accessed 22/09/2023

We used random sampling to select two benchmarks from each category, as well as all benchmarks from the `shootout` category that belong to CLBG, with the exception of `fasta` and `k-nucleotide` due to issues with their execution. Considering multiple categories of benchmarks allows for a comprehensive and balanced evaluation of the compiler's effectiveness in optimizing different code structures. Furthermore, the benchmarks from CLBG allow for a direct comparison of the compiler's performance against other languages and implementations. The 18 benchmarks under consideration are listed in Table 2.

Category	Benchmarks
gc	circsim, hash
imaginary	bernoulli, integrate
parallel	coins, queens
real	anna, fluid
shootout	binary-trees, fannkuch-redux, n-body, pidigits, reverse-complement, spectral-norm
smp	callback001, chan
spectral	power, treejoin

Table 2: Selected NoFib benchmarks

2.3 Equipment and Setup

The machine used to perform the experiment 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 GB 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 [31].

The checkout of NoFib used was `bca0196`. As some of NoFib's benchmarks require external packages, the packages used and respective versions are listed in Table 3.

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

Table 3: Packages used and respective versions

To reduce the energy consumption of the system and to reduce the noise in the readings, we first stop all non-essential services on the machine, as well as disable the WiFi.

Concerning the data collection, each benchmark is compiled without/with an optimization, and when the CPU baseline temperature is reached, either by cooling down or heating up the CPU, we execute the benchmark and measure its energy consumption and execution time. The compiled benchmarks are executed 20 consecutive times, and the order of execution was kept the same throughout the experiment.

We use the system file `/sys/class/thermal/thermal_zone0/temp` to read temperature data and the energy consumption of the programs was measured using Intel's Running Average Power Limit (RAPL), as it has been reported as having negligible overhead and providing accurate results [10, 29]. Furthermore RAPL has been intensively used in other research works with similar goals [6, 14, 21, 24].

2.4 Package vs. DRAM Energy

RAPL allows for the collection of both package and DRAM energy consumption. While package energy consumption represents the energy consumed by the entire processor socket, including CPU cores, memory controller, and other components, DRAM energy consumption refers only to the energy consumed by the dynamic random access memory modules.

In this paper, we consider the total energy consumption to be the sum of both package's and DRAM's energy consumption, but these two do not contribute to this metric in equal parts. Taking into consideration all of the collected samples we calculated that on average 4.23% of the total energy consumption is consumed by DRAM and the remaining 95.77% is consumed by the package. Therefore, optimizing the package energy consumption could yield significant energy savings compared to strategies that optimize DRAM consumption.

2.5 Reducing the temperature induced Noise Level

The CPU temperature has been proven to influence its energy consumption [32]. In this study we do not tamper with the temperature during execution, instead we force the execution to begin at specific starting temperatures: 45ºC, 55ºC and 65ºC.

For each temperature, we collected two batches of 20 samples of all programs with each optimization flag. To assess the noise level for each temperature, we calculated the batches' differences in execution time and energy consumption.

For every program-optimization combination, we calculated the minimum and maximum statistically significant increases in both energy consumption and execution time, Table 4. From these results, we are able to conclude that a starting temperature of 45ºC presented the smallest range in total energy and execution time.

Temperature (ºC)	Energy Inc. (%)	Time Inc (%)
45	[-2.89, 1.74]	[-4.06, 3.01]
55	[-4.60, 25.86]	[-18.96, 5.89]
65	[-4.94, 14.22]	[-16.23, 9.81]

Table 4: Minimum and maximum statistically significant percentage increases in total energy consumption and execution time, by starting temperature

In Figure 1, we can observe that varying the starting temperature affects the distribution of the percentage increases in total energy consumption. The batches with a starting temperature of

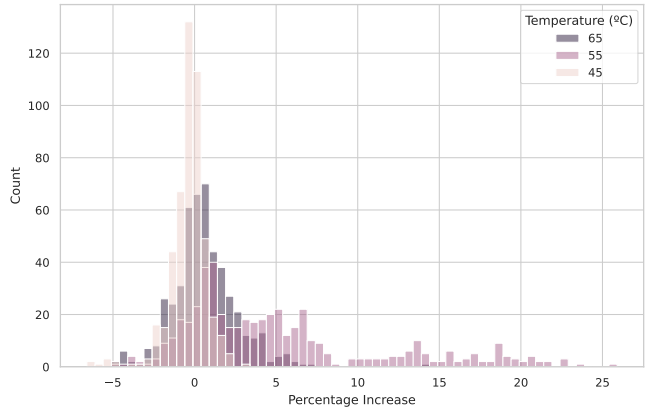

Figure 1: Percentage increase in total energy consumption between batches

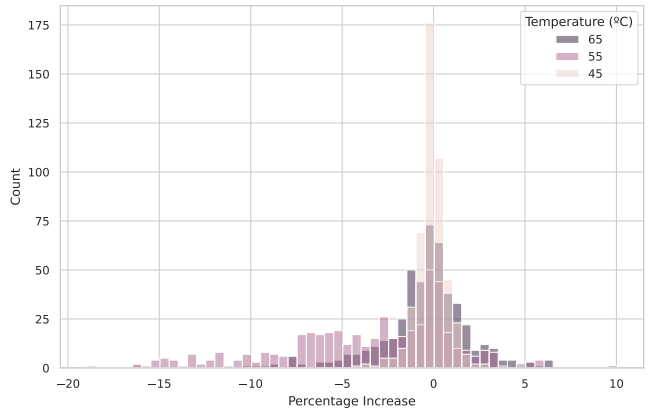

Figure 2: Percentage increase in execution time between batches

45ºC had the smallest variance in total energy, while the ones executed at 55ºC had the largest. The very same behavior is present when considering execution time, and can be observed in Figure 2. The minimum and maximum statistically significant percentage increases in both energy and execution time can be observed in Table 4.

As the measurements collected with a starting CPU temperature of 45ºC show the least variance when considering both total energy consumption and execution time, these samples are the ones taken into consideration when looking at the effects of the compiler optimizations.

2.6 Statistical analysis

To evaluate the impact of compiler optimizations on energy efficiency, we need to quantify the disparities between sample sets for each benchmark when compiled with and without each optimization. To assess potential differences in the means, we employ Welch's t-test [34], for which specific assumptions must be satisfied:

(1) all samples are drawn independently from each other
(2) no significant outliers are present
(3) data follows a normal distribution

The first assumption is met due to the manner in which data collection is conducted. To address the second assumption, quartile outliers were removed prior to any statistical analysis. And, for validation of data normality, we conducted Shapiro-Wilks [30], D'Agostino's K^2 [7] and Anderson-Darling [2] tests. Only combinations passing all three normality tests were considered as normally distributed. For combinations that respected all three assumptions, Welch's t-test was employed. For combinations not satisfying all assumptions, a random sampling[5] [33] approach with 100,000 iterations was utilized to determine whether differences in means between two groups were statistically significant or occurred by chance.

To analyze the correlation between the energy consumption and execution time, Spearman's correlation coefficient [35] was calculated.

At last, to compare the influence that the CPU's starting temperature has on the energy consumption of programs, we used the one-sided Mann Whitney U test [15]. For this version of the test, the alternative hypothesis is that the first distribution is stochastically larger than the second distribution. The test was performed twice for each pair of starting temperatures, with a confidence level of 0.05.

3 RESULTS ANALYSIS AND DISCUSSION

In this section, we present results we obtained following the adopted methodology and discuss the findings we derived from them.

3.1 Impact of Optimizations on Energy Consumption

Throughout this section, we will thoroughly discuss (percentage) increases in energy consumption. It is important to note that, since most often we are deactivating compiler optimizations (using `-fno-*`), an increase in energy consumption in these cases means that the corresponding optimizations would have saved energy if they had been activated. However, for the `-funfolding-use-threshold=<n>` compiler option, a positive percentage increase translates to an actual increase in energy consumption.

After removing the quartile outliers from the measurements, the percentage increase on average energy consumption of every benchmark compiled without/with each compiler option compared to the average energy consumption of that same benchmark when compiled with `-O2`, Figure 3.

In Figure 3, we depict the percentage increase, for all flags, for which the highest increase was observed; all other benchmarks are grouped in an `Others` category. Analysing the highlighted benchmarks in Figure 3, we verify that when some of the optimizations are disabled there is an increase in energy of over 100%, i.e. they optimize the energy consumption of a benchmark to less than half. The most extreme example of this is `spectral-norm` when compiled with the `-fno-strictness` option, with an increase of 371.37%.

Moreover, we can observe that when compiled with different options, a benchmark's energy consumption varies. Taking `integrate`

[5] Also referred to as A/B testing.

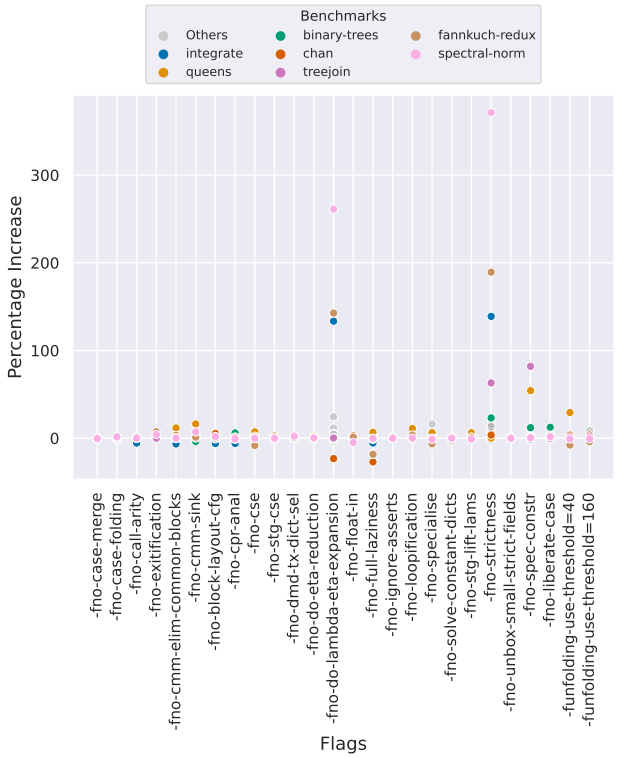

Figure 3: Percentage increase in total energy consumption per flag

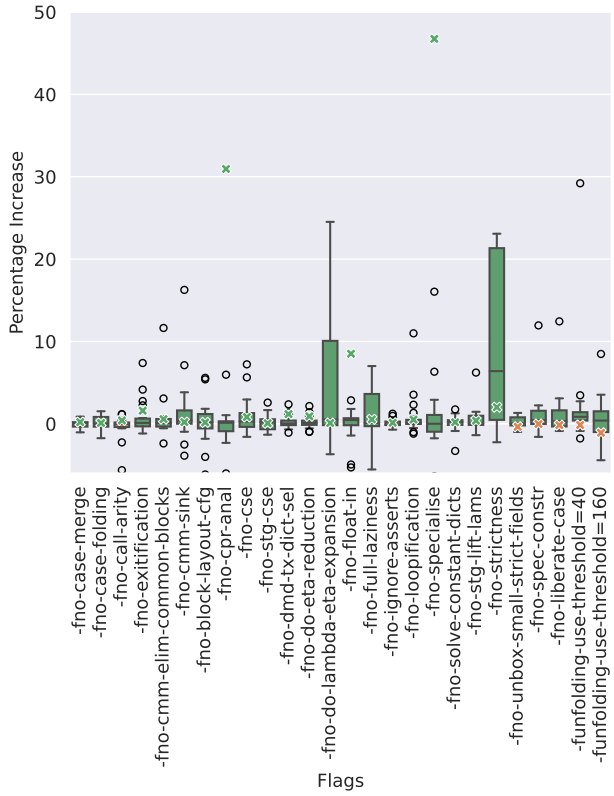

Figure 4: Energy improvement distribution per flag

as an example, when compiled with `-fno-do-lambda-eta-expansion` or `-fno-strictness`, its energy consumption increases around 130%, but other options seem to barely influence its consumption or may even decrease it, such as `-fno-specialize` which reduces consumption by 7.56%. This indicates that the same compiler option may influence different benchmarks in different ways, for example `fno-do-lambda-eta-expansion` increases the energy consumption of `fannkuch-redux`, `integrate` and `spectral-norm`, but `chan`'s energy consumption recorded a decrease. After the overview, we will investigate the energy consumption of 468 benchmark-compiler optimization pairs.

In Figure 4, the box plot provides an overview of the optimization flags' impact on the benchmarks' energy consumption. Each box plot illustrates the energy distribution of the average percentage increase across all programs. The data mirrors that in Figure 3, but the y-axis is cropped to $[-6.0, 50.0]$ for better readability, excluding 12 outliers. In the figure, green boxes have a positive median and red boxes a negative median, and the crosses indicate the flags average for all programs, where green means a positive average and red a negative average. According to this data 22 out of 24 `-fno-*` options increase the energy consumption of both the typical and the average programs, meaning that the optimization that is being disabled contributes to improving the energy efficiency of these programs. The remaining 2 `-fno-*` options, `-fno-call-arity` and `-fno-ignore-asserts`, reduce the energy consumption of the average program and increase the consumption of the typical program. In addition, the thresholds considered for the `-funfolding-use-threshold=<n>` option both increase consumption of the typical program but reduce the consumption of the average program.

The results of the statistical tests described in Section 2.6 are presented in Figure 5, and can be interpreted as follows: in green, the flag significantly increased the energy consumption of the program compared with `-O2`; in orange, the flag significantly reduced the energy consumption; and in blue, the flag's change in energy consumption was not statistically significant.

Out of all the 468 benchmark-compiler option combinations presented in Figure 5, 111 (24%) significantly increased the energy consumption, i.e., enabling the optimization produced a more energy-efficient executable, and 37 (8%) significantly decreased the energy consumption, i.e., enabling the optimization produced a less energy-efficient executable, while the remaining 321 (69%) did not record a statistically significant difference in energy consumption, meaning any difference in consumption was most likely due to chance.

Analysing this data, we are able to once again verify how the same optimization influences energy consumption of different programs in different ways, now with the support of statistical tests. For example, `-fno-call-arity` increases the total energy consumption of `circsim` and `fannkuch-redux`, but reduces `integrate`'s consumption. Furthermore, it becomes evident that a benchmark's

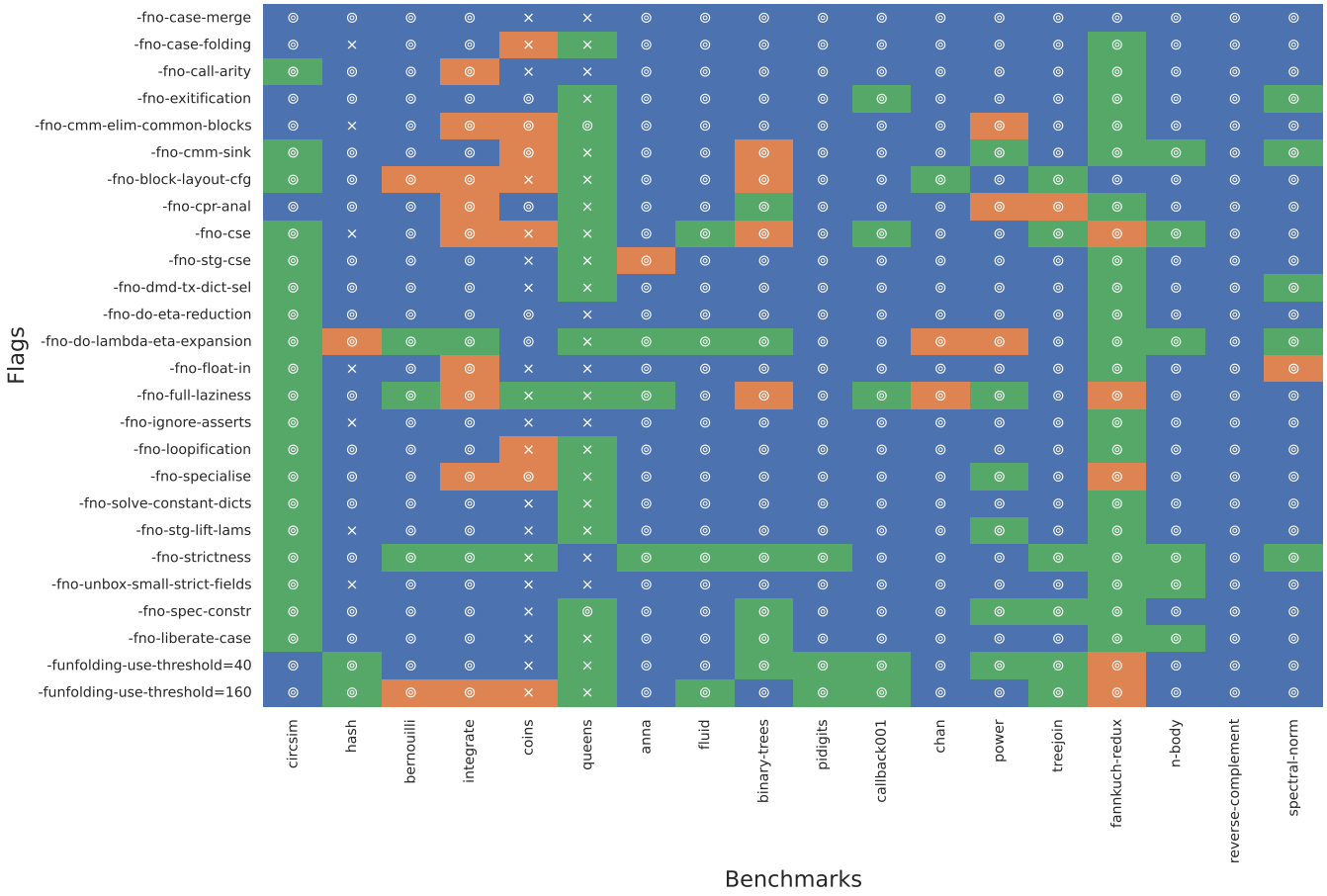

Figure 5: Energy-efficiency results for each flag and benchmark combination. The "×" label denotes significance tested with Welch's t-test, and "◎" with A/B testing. *Orange* indicates a significant energy reduction, *blue* denotes no significant difference, and *green* denotes a significant increase.

energy consumption may vary based on applied optimizations. Consider `coins`, which experienced both increases and decreases in consumption with different options, and mostly no significant difference with others.

Considering how the `-fno-*` options impacted the benchmarks: 1 option, the `-fno-case-merge`, showed no significant impact on the energy efficiency of the benchmarks, 10 options either increased the energy consumption or had no significant impact on the benchmarks, and 14 options sometimes increased, sometimes decreased, and sometimes had no impact on the energy consumption.

The optimization that significantly impacted the most benchmarks was `-fno-do-lambda-eta-expansion`, increasing the consumption of 10 benchmarks and decreasing the consumption of 3. And the option which impacted the least was `-fno-case-merge`, which did not significantly impact the consumption of any benchmark. For 18 out of the 24 `-fno-*` options considered in this study, the number of benchmarks which recorded an increase in consumption was higher than the ones which recorded a decrease, with `-fno-strictness` option being the most successful, increasing the consumption of 12 benchmarks without decreasing any.

When it comes to the `-funfolding-use-threshold=<n>` options, both of the tested thresholds had a mostly negative effect on the benchmarks, as the number of benchmarks which suffered an increase is larger than those which recorded a decrease.

RQ$_1$: How do GHC optimizations influence the energy efficiency of programs?

From our analysis, we are able to confirm that compiler optimizations do have an effect on a program's energy efficiency. While there is no compiler transformation that in general improves all programs, 10 improved or caused no negative effects for all the investigated benchmarks. In addition, considering that for 28% of the cases enabling optimizations produced a more energy-efficient executable there is great potential in using compiler transformations as a mean to reach energy savings. For example, the `-fno-strictness` option increases the consumption of `spectral-norm` by 371.37%, i.e, the transformation that is disabled reduces the consumption of the program by 78%.

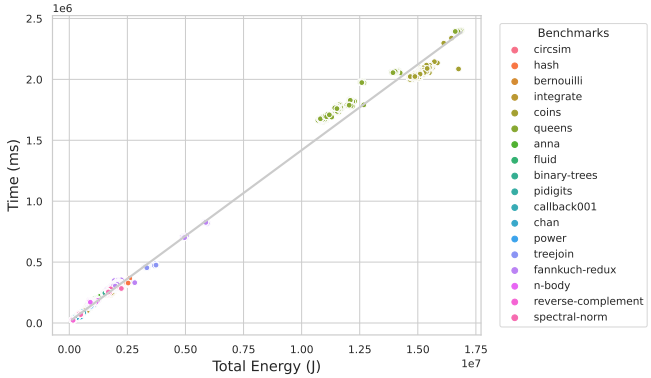

Figure 7: Energy consumption and execution time by benchmark

3.2 Execution Time & Energy

To evaluate if GHC optimizations affect program execution time similarly to energy consumption, we analyze the correlation between execution time and energy consumption for all executables generated by various `-fno-` option and benchmark combinations. Additionally, we explore how `-fno-` options impact time and whether their influence on time aligns with their influence on energy.

Correlation between execution time and energy consumption. To evaluate the correlation between execution time and energy consumption in our collected samples, we plotted them in Figure 7. The chart indicates a trend suggesting that higher execution time corresponds to higher energy consumption, and vice versa.

To quantitatively assess the correlation between these metrics, we employed Spearman's correlation coefficient. The calculated coefficient yielded a value of 0.9856, indicating a strong positive monotonic relation, i.e, as one variable increases so does the other. To corroborate this result, the associated p-value was determined

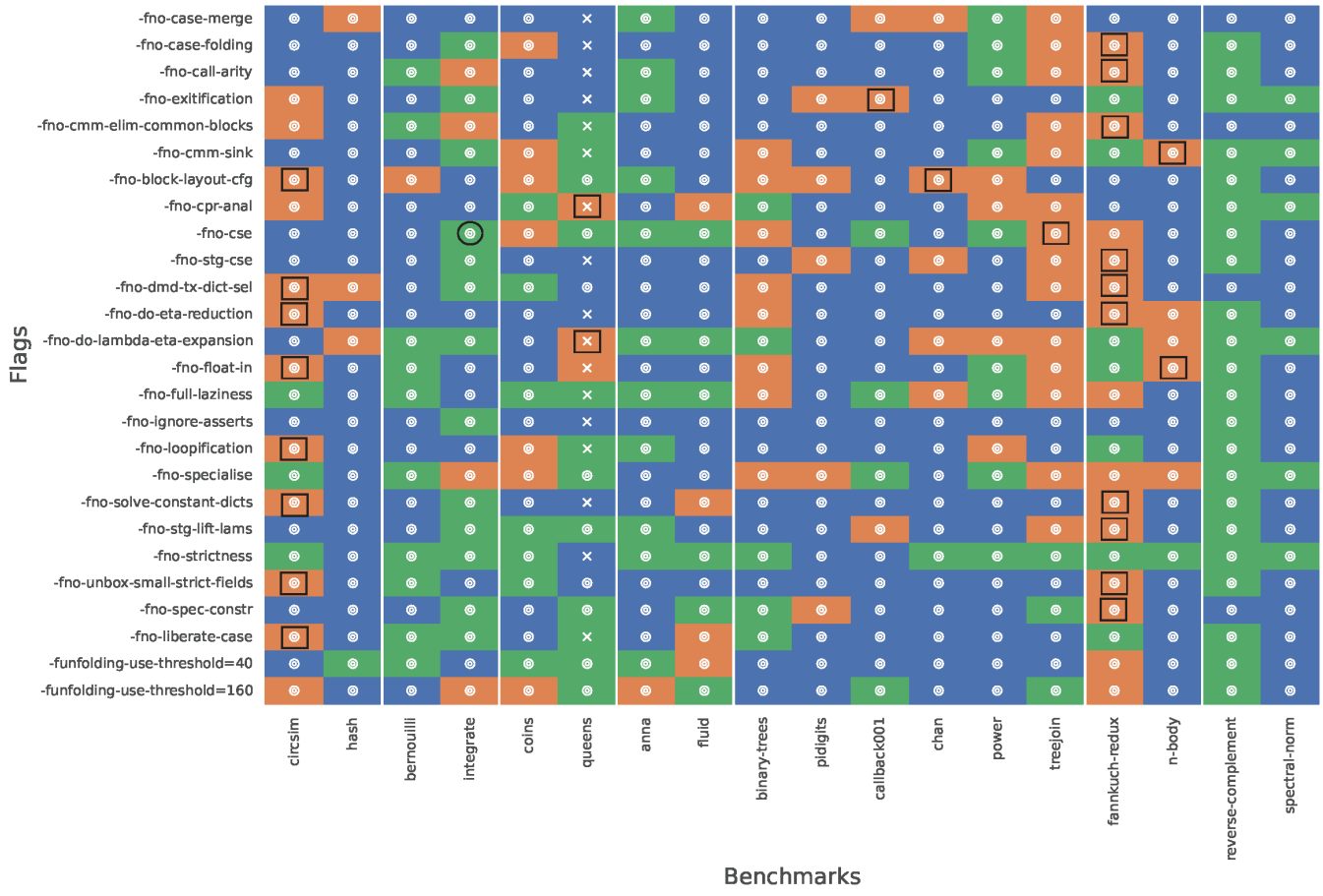

Figure 6: Time-efficiency results for each flag and benchmark combination. The "×" label denotes significance tested with Welch's t-test, and "◎" with A/B testing. *Orange* indicates a significant time reduction, *blue* denotes no significant difference, and *green* signifies a significant time increase. A black circle indicates increased time but decreased energy, while black squares indicate decreased time but increased energy.

to be approximated to 0.0, signifying an exceedingly low likelihood of this correlation occurring by random chance.

Optimizations impact on time and energy. Equivalent to the energy-efficiency impact shown in Figure 5, we depict the effect of `-fno-*` options on performance in Figure 6. The results of the statistical tests detailed in Section 2.6 are illustrated in Figure 6 as follows: green signifies a significant increase in wall time compared with `-O2`; orange indicates a significant reduction, and blue denotes no statistically significant change in performance.

Among the 468 benchmark-compiler option combinations in Figure 6, 121 (26%) exhibited a significant increase in wall time, indicating optimization led to a more time-efficient executable. Conversely, 95 (20%) showed a significant decrease in wall time, suggesting optimization resulted in a more time-consuming executable. The remaining 252 (54%) did not exhibit a statistically significant difference in performance.

Regarding the impact of `-fno-*` options on benchmarks, only one option, `-fno-strictness`, either increased wall time or had no significant impact, implying a positive or neutral effect of enabling the strictness optimization. The remaining 24 options displayed diverse effects on energy consumption.

Looking at the individual benchmark-compiler options, we see that the options often, i.e., 83 (18%) combinations, affect performance and energy-efficiency –significantly– in the same way, and for 213 (46%) combinations, there was no significant effect on neither energy nor time. More interestingly, sometimes the impact is inverse; in 26 (6%) combinations we recorded a significant impact on both the energy and time, but in opposite ways, i.e., one was improved and one was worsened.

In Table 5, we report the combinations impacting both energy and time. Since energy is defined as time multiplied by power dissipation, we anticipated similar impacts on both. This expectation aligns with the results. Specifically, 56 (12%) combinations showed a significant increase in both time and energy with the `-fno-*` option, while 27 (6%) combinations exhibited a significant decrease in both. In 25 (5%) combinations, the option led to a notable increase in energy and a decrease in time, implying optimization improved energy consumption but worsened wall-time. These combinations are marked in Figure 6 with black squares, mainly observed in `circsim` and `frankuch-redux`. A single combination, `-fno-cse` on the `integrate` benchmark, recorded a significant decrease in energy consumption and a simultaneous increase in time, signifying improved performance but worsened energy consumption; this is indicated with a black circle in Figure 6.

RQ_2: Do GHC optimizations influence execution time of programs in the same way they influence energy consumption?

The analysis of the collected data robustly confirms a strong correlation (95% confidence level) between software execution time and energy consumption.

The `-fno-*` options increases time in 26% of the combinations and increases energy in 24% of the combinations, while only 12% increases both time and energy at the same time, thus, implying that enabling optimizations has a positive effect on execution time and energy consumption in 12% of the investigated cases. Because energy is defined as time multiplied by power dissipation, we expected that most cases had similar impact on time and energy. This is well supported by the results, since 6% of the combinations have opposite and significant impact on energy and time. For a single case enabling the optimization would produce a more energy-efficient executable with lower performance and for 25 cases (5%) enabling the optimization would produce a less energy-efficient executable with better performance. When counting no significant impact on energy and time as similar behaviour, we find that for 63% of the tested combinations, the `-fno-*` options has similar impact on energy and time; however, in 46% of the cases we have recorded no significant impact on neither energy nor time.

3.3 Secondary Findings

Previous research shows that CPU temperature affects the energy consumption of software [32], in this study we assess whether or not there is a significant difference in energy consumption derived from the temperature at the beginning of a program's execution.

The average energy consumption increases along with the increase of the starting temperature, as can be observed in Table 6. However, according to the medians, a program started at 55°C tends to have a smaller consumption than one started at 45°C or 65°C. The one-sided Mann-Whitney U test results, presented in Table 7, corroborate this observation. According to the results of this test programs started at 45°C tend to consume more energy than those started with 55°C, and less than those started with 65°C.

4 THREATS TO VALIDITY

In this section, we discuss potential threats that may affect the validity of our findings, or their generalization.

We concentrate on the programming language Haskell and GHC, thus our study's results may not apply to other languages, even with similar optimizations. Different optimizations or compilers could also yield different conclusions.

The fact that we used a single hardware configuration to perform our experiments helps with the consistency of the results, as the hardware itself influences energy consumption. While the chosen system is representative of commonly used hardware, replicating the experiment with other hardware may yield different results.

An aspect that has been proven to influence energy consumption is the data. Altering data, such as input size, can change the execution path and subsequently affect energy consumption. More interestingly, even with the same execution path, different data may lead to varying energy consumption [13, 22]. In our experiment, we maintained consistent results by not modifying the handled data. However, replicating the experiment with different input data may yield varied results.

5 RELATED WORK

Over the years, more and more researchers have taken interest in software energy efficiency, and have explored many different aspects in order to determine their influence on energy consumption.

Some works have already studied the impact that compiler optimizations have on programs. In [1], the authors studied the impact of GCC's "packages" of optimizations, namely `-O1`, `-O2` and `-O3`, on C

	Energy increase	Energy decrease	No sign. Energy change	Total
Time increase	56 (12%)	1 (0%)	64 (14%)	121 (26%)
Time decrease	25 (5%)	27 (6%)	43 (9%)	95 (20%)
No sign. Time change	30 (6%)	9 (2%)	213 (46%)	252 (54%)
Total	111 (24%)	37 (8%)	321 (69%)	468 (100%)

Table 5: The count and percentages of `-fno-*` options and benchmark combinations with significantly increasing, decreasing, or nonsignificant impact.

Temperature (°C)	Mean (J)	Median (J)
45	2485690.28	1005308.00
55	2502110.99	984983.00
65	2588643.32	1045437.50

Table 6: Mean and median energy consumption based on starting temperature.

Hypothesis	$\rho - value$
45 > 55	0.0221
45 < 65	8.3406e-14
55 < 65	4.3186e-20

Table 7: Confidence levels for Mann-Whitney U tests on energy consumption differences based on starting temperature.

programs' energy efficiency. They found that the `-O2` and `-O3` flags were the ones that produce the most energy-efficient programs. Similarly, in [18], a comparison of GHC's `-O0`, `-O1` and `-O2` is performed. For some implementations, the non-optimized executables, compiled with `-O0`, performed better, both in terms of execution time and energy consumption, than the executables compiled with `-O1` and `-O2`. In the GHC User's Guide [31], the GHC authors themselves warn users about the lack of consistency of the `-O*` flags. According to [18], these optimizations can either improve or worsen the software's performance. In [20], the authors evaluate the impact of compiler phase ordering on energy consumption of C programs in comparison with the standard compiler phase orders, i.e. `-O*` flags. According to their experiments with Clang and LLVM, standard optimization levels (`-O1`, `-O2` and `-O3`) always result in better performing executables that consume less energy, when compared with executables generated without optimization. It is also reported that `-O2` and `-O3`, tend to improve energy consumption and performance over `-O1`. When compiling with phased orders specialized for improving energy efficiency, the authors report, in some cases, higher energy savings than if optimizing only for performance.

Another aspect that researchers have considered when investigating software energy efficiency is the programming language used when developing software. In [6], the authors set out to establish a ranking of 10 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]. One of the limitations identified by the authors was the fact that only CPU energy consumption was monitored, leaving out other components. This is addressed in [24], where they focus on relating energy consumption, execution time, and memory usage. The 27 languages studied were ranked according to four different objectives: Time & Memory, Energy & Time, Energy & Memory, and Energy & Time & Memory. The results of this study were later validated in [25].

Apart from programming languages and compile-time optimizations, there have also been studies on the influence that other implementation aspects have on energy consumption. Some studies have focused on the use of different data structures [14, 18, 26]. Notably, [14, 18] are also in the context of Haskell, where the effects of strictness and concurrency on energy consumption has also been studied [14, 19]. Finally, other studies have focused on the impact of the input data [13, 22].

6 CONCLUSIONS AND FUTURE WORK

This work intended to understand how individual optimizations applied by GHC impacts Haskell programs' energy efficiency.

We examined 18 Haskell programs and 25 GHC optimizations, compiling each program 26 times —once with all optimizations enabled and once with each optimization individually disabled. We gathered data on energy consumption and execution time. Through a statistical analysis of the data, we were able to conclude that a program's energy consumption and execution time can be significantly affected by the choice of compiler optimization flags.

In general, it is difficult to make definite conclusions about optimizations. Equivalent to time, the same optimization can increase energy for one benchmark and decrease it for another. However, the results show that for 22 out of 24 -fno-* options, enabling the optimization reduces the energy consumption of programs; this holds true both when considering the median and the average energy consumption of the programs. In addition, enabling individual compiler optimizations resulted in 10 producing equivalent or more energy-efficient executables for all benchmarks, while only 1 produced better-performing or equivalent executables for all benchmarks. Therefore, 10 optimizations are energy-safe, while only 1 is time-safe for this benchmark suite.

Considering all optimization and benchmark combinations, for 24% enabling the optimizations provided more energy efficient executables, and for 26% enabling the optimizations provided better performing executables. However, when considering both aspects at once, only for 12% provided both significantly better performing and significantly more energy-efficient programs. In 5% of all the combinations (26 out of 468), we observed opposite impacts on time and and energy. An investigation of these may provide new insights on the trade-offs between optimizing for time and for energy.

A secondary finding revealed that starting a program when the CPU is 55°C tends to result in lower energy consumption compared to starting at 45°C or 65°C. However, executing at 45°C shows

smaller variance in time and energy measurements than at 55ºC or 65ºC. The starting temperature influences waiting times; a higher tolerable starting temperature leads to shorter waiting times. Thus, identifying 55ºC as a more energy-efficient starting temperature allows us to reduce both waiting times and energy consumption.

In the future, it would be essential to extend this study to more benchmarks and combinations of flags to perhaps confirm the current trends. It would be interesting to investigate the optimizations with opposite impact on energy and time to identify the reasons for this and, thus, revealing possible time and energy trade-offs within the GHC. In addition, it would be exciting to identify the commonalities of the safe optimizations to inspire new similar transformations or perhaps compose a new sequence of energy-optimizing flags.

ACKNOWLEDGMENTS

This article is based upon work from COST Action CERCIRAS CA19135, supported by COST (European Cooperation in Science and Technology). We acknowledge the feedback of Aadil Chasmawala on earlier versions of this paper.

REFERENCES

[1] Sarah Abdulsalam, Donna Lakomski, Qijun Gu, Tongdan Jin, and Ziliang Zong. 2014. Program energy efficiency: The impact of language, compiler and implementation choices. In *International Green Computing Conference*. IEEE, Dallas, TX, USA, 1–6. https://doi.org/10.1109/IGCC.2014.7039169

[2] T. W. Anderson and D. A. Darling. 1952. Asymptotic Theory of Certain "Goodness of Fit" Criteria Based on Stochastic Processes. *The Annals of Mathematical Statistics* 23, 2 (1952), 193 – 212. https://doi.org/10.1214/aoms/1177729437 Publisher: Institute of Mathematical Statistics.

[3] Anders Andrae. 2020. New perspectives on internet electricity use in 2030. *Engineering and Applied Science Letters* 3 (Jun 2020), 19–31. https://doi.org/10.30538/psrp-easl2020.0038

[4] R. S. Bird. 1984. Using circular programs to eliminate multiple traversals of data. *Acta Informatica* 21, 3 (1984), 239–250. https://doi.org/10.1007/BF00264249

[5] Sergey Bushnyak. 2021. Companies that use Haskell in production. Available at: https://haskellcosm.com/. https://haskellcosm.com/

[6] Marco Couto, Rui Pereira, Francisco Ribeiro, Rui Rua, and João Saraiva. 2017. Towards a Green Ranking for Programming Languages. In *Proceedings of the 21st Brazilian Symposium on Programming Languages* (Fortaleza, CE, Brazil) *(SBLP 2017)*. Association for Computing Machinery, New York, NY, USA, Article 7, 8 pages. https://doi.org/10.1145/3125374.3125382

[7] Ralph D'Agostino and E. S. Pearson. 1973. Tests for Departure from Normality. Empirical Results for the Distributions of b2 and $\sqrt{b1}$. *Biometrika* 60, 3 (1973), 613–622.

[8] Taylor Fausak. 2021. 2021 State of Haskell Survey Results. Available at: https://taylor.fausak.me/2021/11/16/haskell-survey-results/. https://taylor.fausak.me/2021/11/16/haskell-survey-results/

[9] João Paulo Fernandes, Alberto Pardo, and João Saraiva. 2007. A shortcut fusion rule for circular program calculation. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*, Gabriele Keller (Ed.). ACM, 95–106. https://doi.org/10.1145/1291201.1291216

[10] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. 2012. Measuring Energy Consumption for Short Code Paths Using RAPL. *SIGMETRICS Perform. Eval. Rev.* 40, 3 (Jan 2012), 13–17. https://doi.org/10.1145/2425248.2425252

[11] HaskellWiki. 2020. Haskell in industry. Available at: https://wiki.haskell.org/index.php?title=Haskell_in_industry&oldid=63468. https://wiki.haskell.org/index.php?title=Haskell_in_industry&oldid=63468

[12] J. Hughes. 1989. Why Functional Programming Matters. *Comput. J.* 32, 2 (01 1989), 98–107.

[13] Steve Kerrison and Kerstin Eder. 2015. Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor. *ACM Trans. Embed. Comput. Syst.* 14, 3, Article 56 (Apr 2015), 25 pages. https://doi.org/10.1145/2700104

[14] Luís Gabriel Lima, Francisco Soares-Neto, Paulo Lieuthier, Fernando Castor, Gilberto Melfe, and João Paulo Fernandes. 2016. Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, Osaka, Japan, 517–528. https://doi.org/10.1109/SANER.2016.85

[15] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50 – 60. https://doi.org/10.1214/aoms/1177730491

[16] Simon Marlow. 2018. Fighting spam with Haskell. Available at: https://engineering.fb.com/2015/06/26/security/fighting-spam-with-haskell/. https://engineering.fb.com/2015/06/26/security/fighting-spam-with-haskell/

[17] Simon Marlow and Simon Peyton Jones. 2012. *The Glasgow Haskell Compiler* (the architecture of open source applications, volume 2 ed.). Lulu, Chapter 5, 67–88.

[18] Gilberto Melfe, Alcides Fonseca, and João Paulo Fernandes. 2018. Helping Developers Write Energy Efficient Haskell through a Data-Structure Evaluation. In *Proceedings of the 6th International Workshop on Green and Sustainable Software* (Gothenburg, Sweden) *(GREENS '18)*. Association for Computing Machinery, New York, NY, USA, 9–15. https://doi.org/10.1145/3194078.3194080

[19] Gilberto Melfe, Alcides Fonseca, and João Paulo Fernandes. 2018. Evaluation of the impact on energy consumption of lazy versus strict evaluation of Haskell data-structures. In *Proceedings of the XXII Brazilian Symposium on Programming Languages, SBLP 2018, Sao Carlos, Brazil, September 20-21, 2018*, Carlos Camarão and Martin Sulzmann (Eds.). ACM, 83–89. https://doi.org/10.1145/3264637.3264648

[20] Ricardo Nobre, Luís Reis, and João M. P. Cardoso. 2018. Compiler Phase Ordering as an Orthogonal Approach for Reducing Energy Consumption. https://doi.org/10.48550/ARXIV.1807.00638

[21] Ana Oprescu, Sander Misdorp, and Koen van Elsen. 2022. Energy cost and accuracy impact of k-anonymity. In *International Conference on ICT for Sustainability, ICT4S 2022, Plovdiv, Bulgaria, June 13-17, 2022*. IEEE, 65–76. https://doi.org/10.1109/ICT4S55073.2022.00018

[22] James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. 2017. Data dependent energy modeling for worst case energy consumption analysis. In *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems* (Sankt Goar Germany). ACM, New York, NY, USA, 51–59.

[23] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E. Hassan. 2016. What Do Programmers Know about Software Energy Consumption? *IEEE Software* 33, 3 (2016), 83–89. https://doi.org/10.1109/MS.2015.83

[24] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. 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* (Vancouver, BC, Canada) *(SLE 2017)*. Association for Computing Machinery, New York, NY, USA, 256–267. https://doi.org/10.1145/3136014.3136031

[25] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2021. Ranking programming languages by energy efficiency. *Science of Computer Programming* 205 (2021), 102609. https://doi.org/10.1016/j.scico.2021.102609

[26] Rui Pereira, Marco Couto, João Saraiva, Jácome Cunha, and João Paulo Fernandes. 2016. The Influence of the Java Collection Framework on Overall Energy Consumption. In *Proceedings of the 5th International Workshop on Green and Sustainable Software* (Austin, Texas) *(GREENS '16)*. Association for Computing Machinery, New York, NY, USA, 15–21. https://doi.org/10.1145/2896967.2896968

[27] Gustavo Pinto, Fernando Castor, and Yu David Liu. 2014. Mining Questions about Software Energy Consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 22–31. https://doi.org/10.1145/2597073.2597110

[28] Hannah Ritchie and Max Roser. 2020. Energy. *Our World in Data* (2020). Available at: https://ourworldindata.org/energy.

[29] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. 2012. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. *IEEE Micro* 32, 2 (2012), 20–27. https://doi.org/10.1109/MM.2012.12

[30] S. S. Shapiro and M. B. Wilk. 1965. An Analysis of Variance Test for Normality (Complete Samples). *Biometrika* 52, 3/4 (1965), 591–611.

[31] GHC Team. 2020. GHC User's Guide Documentation. Available at: https://downloads.haskell.org/~ghc/8.8.4/docs/users_guide.pdf, accessed 22/09/2023. https://downloads.haskell.org/~ghc/8.8.4/docs/users_guide.pdf

[32] Yewan Wang, D. Nortershauser, Stephane Masson, and Jean-Marc Menaud. 2018. Potential effects on server power metering and modeling. *Wireless Networks* (11 2018), 1–12. https://doi.org/10.1007/s11276-018-1882-1

[33] N.A. Weiss, P.T. Holmes, and M. Hardy. 2005. *A Course in Probability*. Pearson Addison Wesley, Upper Saddle River, NJ. 112–114 pages.

[34] B. L. Welch. 1947. The Generalization of 'Student's' Problem when Several Different Population Variances are Involved. *Biometrika* 34, 1/2 (1947), 28–35. http://www.jstor.org/stable/2332510

[35] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. 2012. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, Berlin, Germany.