

Does GCC Compiler Development Pay for Itself? GCC Versions' Impact on Energy Consumption

Brage Isak Keiserås  

Department of Informatics, University of Oslo, Norway

Bo Markussen  

Department of Mathematics, University of Copenhagen, Denmark

Ingrid Chieh Yu  

Department of Informatics, University of Oslo, Norway

Ken Friis Larsen  

Department of Computer Science, University of Copenhagen, Denmark

Maja H. Kirkeby  

Department of People and Technology, Roskilde University, Denmark

Michael Kirkedal Thomsen  

Department of Informatics, University of Oslo, Norway

Department of Computer Science, University of Copenhagen, Denmark

Abstract

Background. Compilers are actively developed programs, with newer versions implicitly promising better performance.

Aim. In this paper, we evaluate whether the cost of the expected improvements in optimisations in newer versions of the GCC C compiler pays off, both in terms of execution time and energy consumption.

Method. To do so, we select 10 major versions of GCC (spanning GCC-4 to GCC-13), and four different optimization flags (*-Og*, *-O2*, *-O3*, and *-Os*) to compile and run benchmarks written in C from the SPEC CPU 2017 benchmarking suite, measuring both the energy and time of compilation and running of the benchmarks. A linear mixed-effects model mapping the GCC major version number to a numerical variable is then fitted to the data.

Result. We find a statistically significant trend of decreasing running costs when optimized with *-O2* and *-O3*, with an overall decrease in costs (for both energy and time) amounting to 13% for *-O2* and 9% for *-O3* overall (from GCC-4 to GCC-13). However, this is not without a cost, as there is also a statistically significant trend of increasing compilation cost for the optimization levels tested except *-Og*, with an increase amounting to 17% in energy and 15% in time with *-O3* enabled, and 18% in energy and 16% in time with *-O2* enabled. Some benchmarks benefit more than others, which we hypothesize is due to more advanced vectorization in newer compiler versions.

Conclusion. GCC is producing lower-cost programs, at the expense of higher compilation cost. Still, the trade-off is well worth it as a single additional execution of the compiled program offsets the increased cost of compilation for both the run-time and energy cost.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Runtime environments

Keywords and phrases Compilers, Compiler Optimisation, GCC, Energy Consumption, Energy Measurement

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.38

Category Technical Track Paper

Supplementary Material *Software*: <https://doi.org/10.6084/m9.figshare.32316525>



© Brage Isak Keiserås, Bo Markussen, Ingrid Chieh Yu, Ken Friis Larsen, Maja H. Kirkeby, and Michael Kirkedal Thomsen;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 38; pp. 38:1–38:19



Leibniz International Proceedings in Informatics
LIPIcs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Funding *Maja H. Kirkeby*: Independent Research Fund Denmark, Project no. 2102-00281B

Acknowledgements We thank Lars Husted Vadgaard for the initial work with experimental setup and Kerstin Eder for inputs to the work.

1 Introduction

One of the most important types of program is the optimising compiler. The job of a compiler is to translate text to a runnable program, while performing optimisations that can improve running time, possibly on specific machines or architectures. For final production code, we generally want the fastest program possible, thus compilers often receive updates with new promising optimizations, resulting in hopefully better performing programs. With these improvements of optimizations, we expect to see the cost of running the compiled programs decrease, though, at the possible cost of increased compile time as we perform more advanced optimizations.

However, does this expected reduction in running costs actually occur, and is it really worth it? The types of programs that are written every day differ greatly, and not every optimization will optimize every program equally. Indeed, some may harm the performance. If the cost of every single compilation increases, surely, this added cost will compensate for itself by the cost of the compiled program decreasing as well. Programs are only compiled a few times, and hopefully run many times. Stated somewhat cheekily “Compiler optimizations should pay for themselves” [4]. But do they?

It is possible to interpret how compiler optimizations may pay for themselves in several different ways. The most common metric to consider is the execution time, which often gives a good overall assessment of the programs. There also exist more machine-specific metrics: memory consumption, cache behaviour, etc. A more recent metric is energy consumption; either measured externally as wall-plug power or on the machine with, for example, RAPL [9] or an integrated power management circuit. In this work, we have chosen to use the time and energy cost as metrics. Execution time gives the classical cost as a basis for comparison, while energy cost (measured with RAPL) gives an alternative contemporary metric that is receiving increased focus.

There is an inherent trade-off between more compiler optimizations and faster programs produced. The more optimization the compiler does, the faster the compiled program will generally become. Because of this trade-off, compilers will usually have a hierarchy of coarse-grained flags that you can quickly apply to get a faster program, at the cost of higher compilation costs. For GCC, this hierarchy is `-O0`, `-O1`, `-O2`, and `-O3`, with other flags (`-Og`, `-Os`, and `-Oz`) also being available.

To test this, we need a set of benchmarks that we can compile and measure performance. As a representative selection of programs, we use all the benchmarks of the SPEC CPU 2017 suite [21] written in C. The SPEC CPU 2017 benchmark suite is an industry standard consisting of several real-world programs with carefully selected input aiming at stress testing the system’s processor, memory, and compiler.

In this paper, we address the research question: “Is there a downward trend in compilation costs between increasing versions of the GNU C Compiler (GCC) and the performance of the compiled programs?” We consider both time and energy costs of compilation and execution, and examine how coarse-grained optimization flags evolve across GCC versions. Our main experimental factors are GCC major version, optimization flag (`-Os`, `-Og`, `-O2`, `-O3`), and SPEC CPU 2017 C benchmarks. As dependent variables, we measure compilation time and energy, execution time and energy, and the size of the produced benchmark binaries.

The SPEC CPU 2017 C benchmarks serve as our experimental objects. We analyze data with linear mixed effects models in the form of ANOVA and ANCOVA to quantify trends in GCC versions and assess the statistical significance of the observed effects.

2 Related Work

Prior work has studied software energy consumption across programming languages, compilers, compiler versions, and optimization configurations. We organize this work from broader language-level comparisons toward studies most closely related to our focus on GCC version evolution and optimization flag groups.

Several studies have compared the energy consumption of programs written in different programming languages. Pereira et al. [19] compare the energy consumption of a range of programming languages using RAPL-based energy measurements, while Gordillo et al. [6] later conduct a similar comparison using physical energy measurements. Both give similar results and, notably, show that C is always at the top of energy efficiency for the studied benchmarks.

Kempen et al. [12] extend this line of work by further examining the relationship between the programming language and the with implementation of the language. Using RAPL, they show that, when factors such as the number of available cores and the CPU clock frequency are held constant, all language implementations exhibit a constant power draw. Furthermore, they show that different implementations of a language can play a role for their energy usage, which they demonstrate for CPython and PyPy, and LuaJit and Lua.

Concerning C compilers, a study by Hebbar et al. [8] uses SPEC and an Intel machine to determine that the Intel C compiler produces (significantly) faster programs than the GCC compiler due to better utilization of Intel's specialized vector instructions. Another study compares the GCC and the Intel C compiler and determines that the GCC compiler is more energy efficient, although for a different set of benchmarks [10]. However, when measuring execution time, they get more varying results. At some clock frequencies, the GCC outperforms the Intel compiler, and at others, the Intel compiler outperforms the GCC compiler.

There has been some recent work on how a compiler evolves over time. Most similar to our work is Jiménez et al. [11] studying the impact of choosing a newer versions of the compiler/interpreter on energy and time for programs in C, Python, and Java, using a benchmark suite of algorithms taken from the Computer Language Benchmarks Game (CLGB). Their result shows no clear trend; that a newer version means less energy. Ournani et al. [17] study the energy usage of different versions over a variety of Java Virtual Machine (JVM) distributions. They concluded that newer versions generally use less energy than older ones and that the different distributions form clusters of similar energy consumption. They also note that C has the highest correlation between execution time and energy consumption, and Java has the weakest one. In addition, two master's theses [2,16] both look at what effect the compiler/interpreter version has for energy usage. They both use RAPL to measure energy, and both use the CGLB as benchmarks. Olvera compares different versions of Python, C++ (using GNU g++), Java, and JavaScript, and comes to the conclusion that a newer version greatly impacts energy usage for Python and JavaScript. But for Java and especially C++, the version number has little impact. Azevedo compares different versions of C (LLVM's clang), C++ (LLVM's clang), Java, and Python, arriving at a similar conclusion that selecting a newer version generally will slightly improve energy consumption, but that the newest version need not be the optimal. The author notes that this tendency is the weakest, or non-existent, for C (LLVM) depending on the benchmark.

Script 1 GCC benchmarking.

```

foreach GCC in GCC versions do download and build GCC with -O2;
create configurations: (GCC version, optimization flag, benchmark, repetitionn);
randomize configuration order;
compile once to warm up;
foreach configuration in configurations do
  | wait until 45°C; measure as we compile the configuration;
foreach configuration in configurations do
  | wait until 45°C; measure as we run the configuration;
  
```

The effect of different optimization flags on the energy usage of a C or C++ program has been studied by Absdulsalam et al. [1], where they find using *-O2* and *-O3* over *-O1* will reduce the energy usage, and also finding that C, C++, and Java consistently outperform Python energy-wise.

The standard optimization flags: *-O1*, *-O2*, and *-O3*, are convenient, however they are not always optimal. It is possible to select specific, fine-grained optimization flags that outperform them for specific benchmarks and specific machines. Pallister et al. [18] find specific GCC flags for embedded hardware that reduce the energy consumption, and give some evidence that there is no universally good set of flags for any benchmark and any hardware. Similarly, Georgiou et al. [5] find specific (ordered) subsets of the standard flags giving more energy efficient programs for embedded hardware using LLVM. There has also been work on other language optimization flags. For example Kirkeby et al. [14], look at Haskell + GHC's different optimizations at a granular level. The authors come to the conclusion that the optimization flags do statistically affect the energy usage and execution time, and in general show that for 24% of the tested combinations, the energy usage went down enabling optimizations.

Taken together, prior work has investigated either the effect of compiler versions on energy consumption or the effect of compiler optimization configurations. Our study combines these perspectives in a longitudinal analysis focused specifically on GCC. We examine 10 systematically selected GCC versions using four standard optimization levels and SPEC CPU 2017, and measure both compilation and execution costs in terms of time and energy. The most similar work will be further compared to our work in Section 5.

3 Methodology

To test how compilation costs and compiled program's running cost change throughout the evolution of GCCs we design Script 1. We wait until a fixed temperature of 45°C is reached to reduce any impact temperature-variation has on the energy usage.¹ This temperature has been experimentally shown to yield the smallest energy consumption using a similar setup [14]. The experiment was conducted in a temperature-controlled room, on a laptop always connected to the power. The laptop is running no other user programs, and we kept the background processes to a minimum.

¹ Temperature is managed by reading from `/sys/class/thermal/thermal_zone3/temp`.

3.1 The Machine

In order to compile the wide array of GCC versions, we use a “chroot”-environment with Debian Jessie. This should not meaningfully affect performance, the only difference is the usage of different libraries (such as LibC – the C standard library). But we are aiming to measure the impact of the *compiler*, not libraries.

We have the following hardware + OS configuration:

Laptop	LENOVO_MT_20F6_BU_Think_FM_ThinkPad X260
CPU	Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz, <i>Skylake</i>
Host OS	5.15.0-171-generic, Ubuntu 22.04.5 LTS
Chroot LibC	Debian GLIBC 2.19-18+deb8u10

We use `-march=native`, allowing for specific architecture instruction adaptations. Skylake was first supported in GCC version 6; before that (for us version 4 and 5) GCC defaults to using Broadwell instructions.

3.2 On RAPL

Intel’s Running Average Power Limit is used as a tool for measuring energy usage. Its accuracy has been demonstrated in several studies [7, 13, 20]. In particular, Khan et al. [13] note that the accuracy of RAPL has been improved in Haswell, and even further in Skylake, which is the microprocessor of our computer. Moreover, RAPL has been widely used to compare energy usage [2, 14, 16, 17, 19].

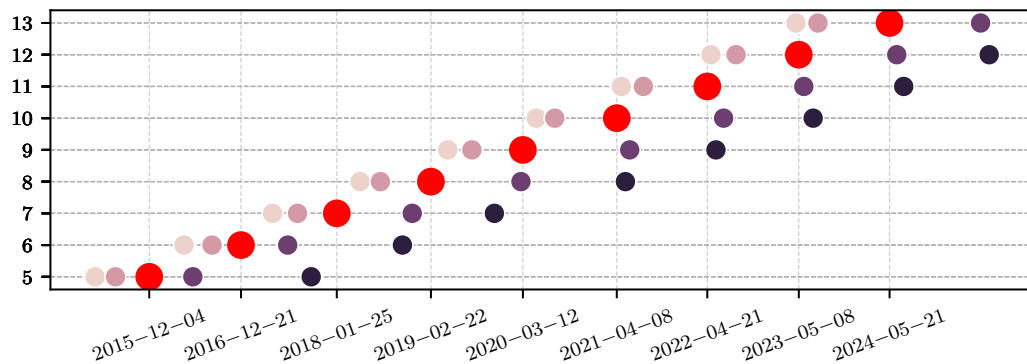
RAPL gives us 4 different readings: PP0, PP1, DRAM, and PKG. PP0 is included into PKG, and PP1 is “Uncore” devices such as the GPU and is not included in our computer. To obtain an overall energy usage, we take the sum of PKG and DRAM; this is in alignment with previous studies (for example in the works of Pereira et al. [19]).

The RAPL registers can occasionally overflow. This manifests itself in huge negative numbers, and these results are then filtered out. In total, 294 rows were filtered, of which 28 were for compilation and 266 for running. However, for each benchmark run, at most 3 (out of 30) for each group were filtered out.

3.3 Which GCC Versions To Pick?

GCC is an actively developed and widely used program. With this comes many mainline releases, but also changes to previous mainline releases. Changes to previous mainline releases can also happen after a new mainline release. In order to pick the versions to compare against each other, we follow three maxims. Firstly, (1) we choose versions of GCC with strictly increasing release dates. Moreover, (2) the major version number must be different for each version. And finally, (3) we want our versions to be more or less equally spaced in terms of the amount of development done to them – which we measure by their release date.

The strategy of always picking the 3rd iteration of the major release for each version aligns well with our maxims. Figure 1 shows all releases of GCC from version 5 to version 15 with the 3rd iteration of each version highlighted. Each version’s release date is more-or-less evenly spaced by one year. As we would like to benchmark a total of 10 versions of GCC, we obtain the following versions: 4.9.3, 5.3.0, 6.3.0, 7.3.0, 8.3.0, 9.3.0, 10.3.0, 11.3.0, 12.2.0, and 13.2.0. In the remainder of this work, we will refer to the different versions by their major version, i.e., the first number in the version triplet (4 for 4.9.3).



■ **Figure 1** GCC versions release timeline. GCC major versions from 5 to 13 over release dates. Each 3rd minor version is highlighted in red, with its release date marked by the ticks.

3.4 Which Optimization Flags to Pick?

While it would be ideal to include all the optimization flags of GCC, this is not feasible. Instead we pick the following four flags: `-Og`, `-O2`, `-O3`, and `-Os`, aiming to get a good mix of the different types of optimizations. `-Og` optimizes for the best debugging experience. It performs a few optimizations and, thus, creates a good baseline as the first optimization flag a developer would use. `-Os` is used for optimizing for the lowest space of the binaries compiled. `-O2` and `-O3` are the primary optimisation flags for production code. These two are expected to take longer to compile, but they generate the fastest code with `-O3` aiming to produce the fastest programs. Taken together, these four flags strike a good balance between having a variety of different optimization targets, and also keeping the time for the experiments to run manageable.

3.5 Which Benchmarks to Pick?

We decided to use the “SPEC CPU® 2017” benchmark suite for this study [21], henceforth referred to as SPEC. SPEC is designed to be an industry-standardized, CPU-intensive suite for measuring and comparing compute intensive performance and striking a good balance of being both compilation-intensive and execution-intensive. As we are measuring the evolution of the GCC C compiler specifically, we choose the subset of the SPEC benchmarks that are only written in C.

SPEC distinguishes between *speed* and *rate* benchmarks, where *speed* measures the score for a single copy and *rate* measures the score for multiple (a user-defined amount) concurrent copies. *speed* is compiled with OpenMP support, and *rate* is not. Hence, each concurrent copy in *rate* runs in a “single-threaded” manner. To best measure how the compiler evolves, we have chosen to disable OpenMP support² by using *rate* with only a single concurrent copy. The scoring metric *rate* uses is not relevant as we are measuring our costs (energy and time) independent of SPEC.

We are then left with the following eight benchmarks: `500.perlbench_r`, `502.gcc_r`, `505.mcf_r`, `519.lbm_r`, `525.x264_r`, `538.imagick_r`, `544.nab_r`, and `557.xz_r`. Table 1 provides a brief overview of what each benchmark does.

² Disabling OpenMP does not disable instruction-level parallelism, such as vectorization, which GCC attempts to optimise for – possibly to great effect, as we will see later.

■ **Table 1** Overview of benchmarks used, adapted from SPEC documentation [21], which classifies benchmarks by predominantly integer or floating-point operations.

Benchmark	Based on	Type
<i>500.perlbench_r</i>	Perl (v5.22.1), interpreter for a popular scripting language.	Integer
<i>502.gcc_r</i>	GCC (4.5.0) with many optimization flags, IA32 as target.	Integer
<i>505.mcf_r</i>	MCF, a single-depot vehicle scheduling.	Integer
<i>519.lbm_r</i>	3D incompressible fluids simulation.	Floating Point
<i>525.x264_r</i>	H.264/AVC video encoder.	Integer
<i>538.imagick_r</i>	ImageMagick, an image manipulation tool.	Floating Point
<i>544.nab_r</i>	Nucleic Acid Builder, a molecular modelling application.	Floating Point
<i>557.xz_r</i>	XZ Utils (5.0.5), in-memory file (de)compression.	Integer

We note that *502.gcc_r* and *525.x264_r* have some compatibility issues, that were fixed by following the recommended solutions described in the benchmark. Namely using *-fcommon* for *525.x264_r* and using *-fgnu89-inline* for *502.gcc_r*. *500.perlbench_r* also had an issue which we were not able to remedy for GCC version 4; therefore, that benchmark is not included and has no data points for this version.

Finally, note that SPEC CPU 2026 has been released, with SPEC CPU 2017 therefore set for retirement. However, SPEC CPU 2017 remains appropriate for our study, as our compilers span from 2014 to 2024.

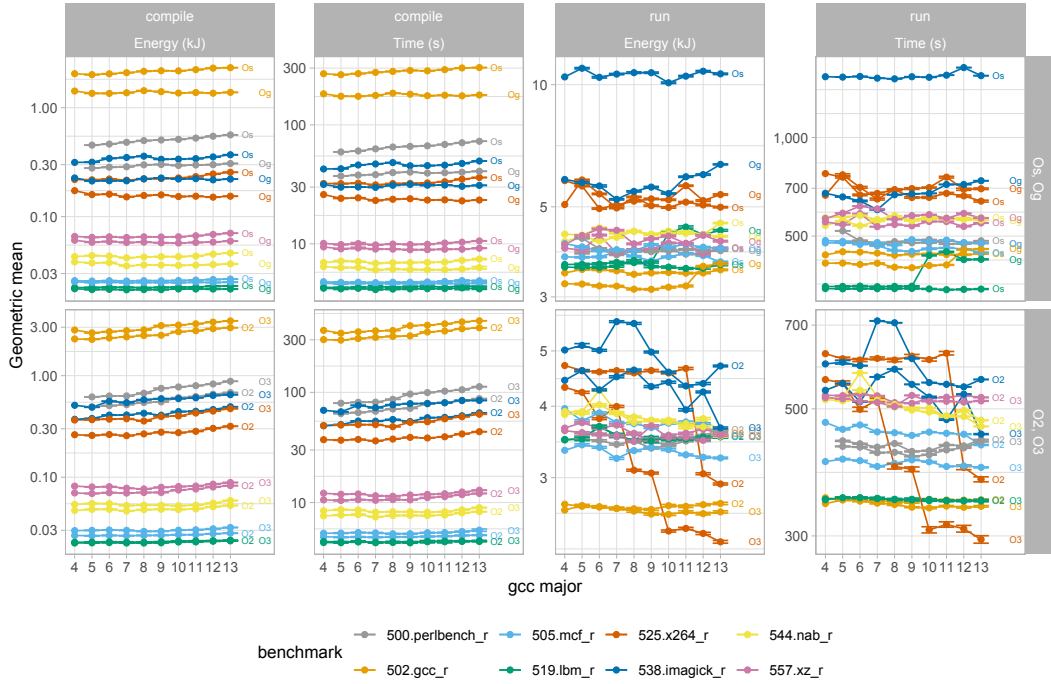
4 Results and Statistical Evaluation

The dataset is stratified into measurements of ENERGY and TIME within COMPILE ($N = 9452$) and RUN ($N = 9216$). Each strata consists of a full factorial design with approximately 30 repetitions for each combination of GCC_VERSION (10 levels: *gcc-04.09.03*, ..., *gcc-13.03.00*), OPTIMIZATION (4 levels: *-Os*, *-Og*, *-O2*, *-O3*) and BENCHMARK (8 levels); except the benchmark *500.perlbench_r* that could not run with GCC-4.

The four strata are analysed in separate linear mixed effects models with fixed effects of BENCHMARK, OPTIMIZATION and GCC_VERSION, and the interactions OPTIMIZATION:BENCHMARK and OPTIMIZATION:GCC_VERSION, and random effects of OPTIMIZATION within combinations of BENCHMARK and GCC_VERSION. This was done for a model (below referred to as ANOVA) where GCC_VERSION in the fixed effects part was used as a categorical variable, and a model (below referred to as ANCOVA) where it was used as a numerical variable GCC_MAJOR giving the major version from 4 to 13. In the models, ENERGY and TIME were log-transformed and GCC_MAJOR was also log-transformed in the ANCOVA model. Thus, after back-transformation by the exponential function, the ANCOVA model corresponds to a power relation between the response and GCC major, and contrasts may be interpreted as relative changes.

In brief, we fitted eight statistical models (ANOVA and ANCOVA for the four strata) with response either $Y = \text{ENERGY}$ or $Y = \text{TIME}$, and where $(A_{Os}, A_{Og}, A_{O2}, A_{O3}) \sim \mathcal{N}_4(0, \Sigma)$ are independent across BENCHMARK and GCC_VERSION, and $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$ are independent across observation index i :

$$\begin{aligned}
 \text{ANOVA: } \log(Y_i) = & \alpha(\text{OPTIMIZATION}_i, \text{BENCHMARK}_i) \\
 & + \beta(\text{OPTIMIZATION}_i, \text{GCC_VERSION}) \\
 & + A_{\text{OPTIMIZATION}_i}(\text{BENCHMARK}_i, \text{GCC_VERSION}_i) + \epsilon_i
 \end{aligned}$$



■ **Figure 2** Raw geometric means (points) with associated 95% confidence intervals (error bars) of the approximately 30 repetitions for each combination of BENCHMARK, OPTIMIZATION, and GCC_VERSION within the 4 strata. The individual combinations of benchmark and optimization are connected by lines across the GCC version. Note that the y-axes are logarithmic and different in the eight panels.

$$\begin{aligned} \text{ANCOVA: } \log(Y_i) = & \alpha(\text{OPTIMIZATION}_i, \text{BENCHMARK}_i) \\ & + \gamma(\text{OPTIMIZATION}_i) \cdot \log(\text{GCC_MAJOR}) \\ & + A_{\text{OPTIMIZATION}_i}(\text{BENCHMARK}_i, \text{GCC_VERSION}_i) + \epsilon_i \end{aligned}$$

The models were fitted in the statistical software R using the package `lme4` (v. 2.0-1), and estimates are given as estimated marginal means using the package `emmeans` (v. 2.0.2).

4.1 Changes Over GCC Version

Figure 2 shows the raw geometric means (i.e., back-transformed raw means in the logarithmic scale) for the 4 strata within combinations of BENCHMARK, OPTIMIZATION, and GCC_VERSION. The benchmarks are connected by lines across the GCC versions. We can see that there is a large difference in the scale of time and energy consumption for the benchmarks, and that some benchmarks experience dramatic reductions in the running cost for energy and time, in particular for `-O2` and `-O3`. This justifies the modelling on the logarithmic scale (otherwise the estimates would be dominated by the benchmarks with the largest consumptions) as well as the addition of the random effects (to model particular behaviour of the benchmarks across optimization and GCC version).

Figure 3 shows the estimated development in energy and time consumption for the four optimizations across the population of benchmarks as a function of the GCC version. The dots with error bars show that estimates and associated 95% confidence intervals from the ANOVA models, and the smooth curves and regions show the same for the ANCOVA

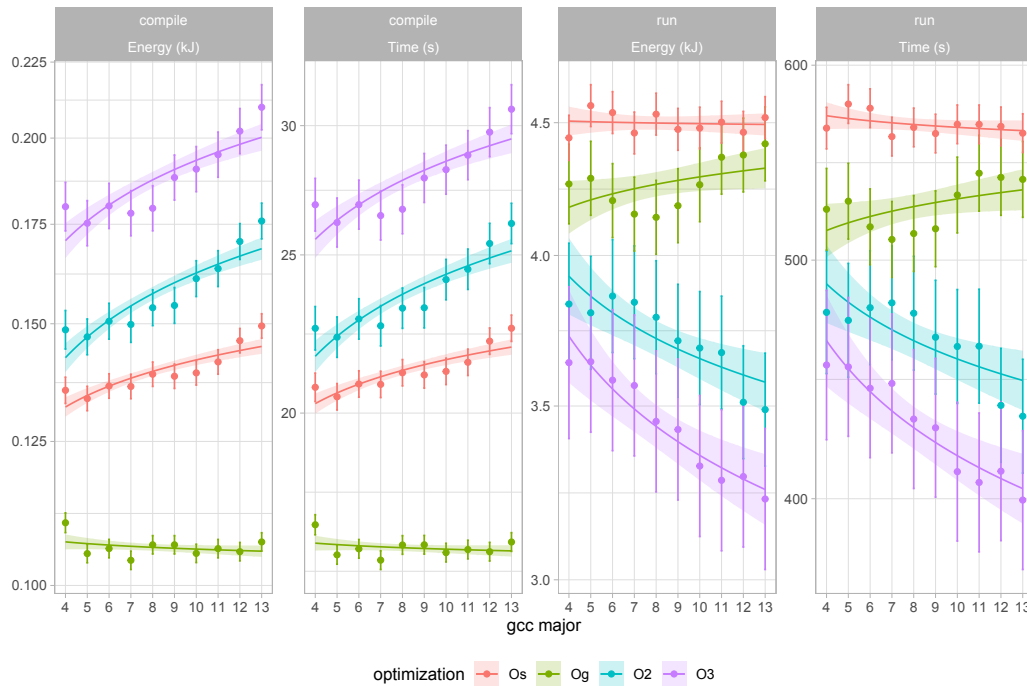


Figure 3 Estimates (points for ANOVA, lines for ANCOVA) and associated 95% confidence intervals (error bars for ANOVA, regions for ANCOVA) for energy and time consumption. Note that the y-axes are logarithmic and different in the four panels.

models. Since the ANOVA models can be understood as the fully non-linear models, whereas the ANCOVA models stipulate a particular functional form (namely a power relation after the back-transformation), we see that the ANCOVA models do not capture the non-linear behaviour. In particular, the ANCOVA model does not capture the dip for the optimization flag *-Og* in RUN ENERGY and RUN TIME around GCC-8. However, in all cases the ANCOVA models may be used to describe the overall trends from GCC-4 to GCC-13.

Table 2 shows the estimated slopes on the logarithmic scale for the ANCOVA models. After back-transformation, the slopes correspond to the exponents in the power relation between the response and the GCC major version. For the optimization flags *-O2* and *-O3*, we see statistically highly significant changes across major with a decrease in run energy and time, and an increase in compile energy and time. For the optimization flag *-Os* we see a highly significant increase in compilation energy and time. For the other six slopes with optimization flags *-Os* and *-Og* we have either non-significant or borderline significant changes. However, the ANCOVA models do not capture the dip for GCC-8 for energy and time consumption for *-Og*. Furthermore, the ANOVA models show a clear decrease until around GCC-8 followed by a clear increase afterwards.

Table 3 shows the estimated ratios (using the ANOVA models) between consecutive GCC major versions within energy and time in combinations with the four optimization flags. After back-transformation the contrasts are interpreted as ratios giving the relative change. In total there are 144 such ratios: 9 consecutive comparisons for each of the 16 combinations of measuring (RUN, COMPILE), response (ENERGY, TIME) and the different optimization flags. Hence, we will only show the ratios that are statistically significantly different from 1.

■ **Table 2** Estimated slopes in ANCOVA models with 95% confidence intervals. P-values are for the null hypotheses of no trend, i.e. slope equal to 0. Note that the slopes are on the log vs log scale, and hence correspond to the power exponents after back-transformation. Note also that P-values and confidence intervals are not adjusted for multiple testing.

Measuring	Response	Optimization	Estimate	P-value
RUN	ENERGY	Os	-0.003 (-0.019 ; 0.014)	0.7607
RUN	ENERGY	Og	0.029 (-0.000 ; 0.059)	0.0523
RUN	ENERGY	O2	-0.080 (-0.123 ; -0.037)	0.0003
RUN	ENERGY	O3	-0.115 (-0.168 ; -0.063)	0.0001
RUN	TIME	Os	-0.012 (-0.028 ; 0.004)	0.1444
RUN	TIME	Og	0.032 (0.001 ; 0.064)	0.0461
RUN	TIME	O2	-0.077 (-0.123 ; -0.030)	0.0012
RUN	TIME	O3	-0.117 (-0.172 ; -0.063)	0.0001
COMPILE	ENERGY	Os	0.080 (0.061 ; 0.099)	0.0001
COMPILE	ENERGY	Og	-0.012 (-0.027 ; 0.002)	0.1015
COMPILE	ENERGY	O2	0.143 (0.115 ; 0.172)	0.0001
COMPILE	ENERGY	O3	0.136 (0.101 ; 0.170)	0.0001
COMPILE	TIME	Os	0.068 (0.050 ; 0.086)	0.0001
COMPILE	TIME	Og	-0.009 (-0.023 ; 0.004)	0.1710
COMPILE	TIME	O2	0.126 (0.098 ; 0.154)	0.0001
COMPILE	TIME	O3	0.120 (0.087 ; 0.154)	0.0001

4.2 Effect of Optimization

Table 4 shows the estimated ratios between the four optimization flags within the four combinations of COMPILE and RUN, and energy and time consumption. We see that all these ratios are highly significantly different from 1. The contrasts are chosen to give estimated ratios above 1. Thus, for run energy we have shown the ratio $-O2/-O3$ instead of the ratio $-O3/-O2$, and we see that optimization $-O2$ is estimated to use 8.1% more run energy than $-O3$. Overall $-O3$ has the highest consumption for COMPILE and the lowest consumption for RUN. This is also evident from Figure 3. Table 4 gives the estimated ratios: for each optimization flags it shows how much more efficient one optimization flag is over another.

4.3 Changes for Individual Benchmark Programs

As seen in Figure 2 the individual programs have different developments across GCC versions. In particular for RUN ENERGY and RUN TIME, where programs like *538.imagick_r* have almost constant consumption from GCC-4 to GCC-13, whereas programs like *525.x264_r* experience huge efficiency gains.

In this section, we examine how the individual programs deviate from the overall changes captured by the ANOVA models. This may be done by inspecting the predicted random effects. Figure 4 shows the exponential function of the difference of the predicted random effects between two consecutive GCC versions within BENCHMARK and OPTIMIZATION. The interpretation of these values is the relative change in consumption in relation to the overall changes in consumption. Note that the y-axes are on a logarithmic scale and different in the 32 panels. For instance, as visualized by the red dot at gcc-12 in panel in the 5'th row and 3'rd column of Figure 4, we see that the RUN ENERGY for *525.x264_r* in optimization $-O2$ decreased more than 30% more than the general change from GCC-11 to GCC-12. This example is particularly pronounced as RUN ENERGY for *525.x264_r* in optimization $-O2$ otherwise follows the general pattern in consumption across the GCC versions (i.e. the other ratios are close to 1).

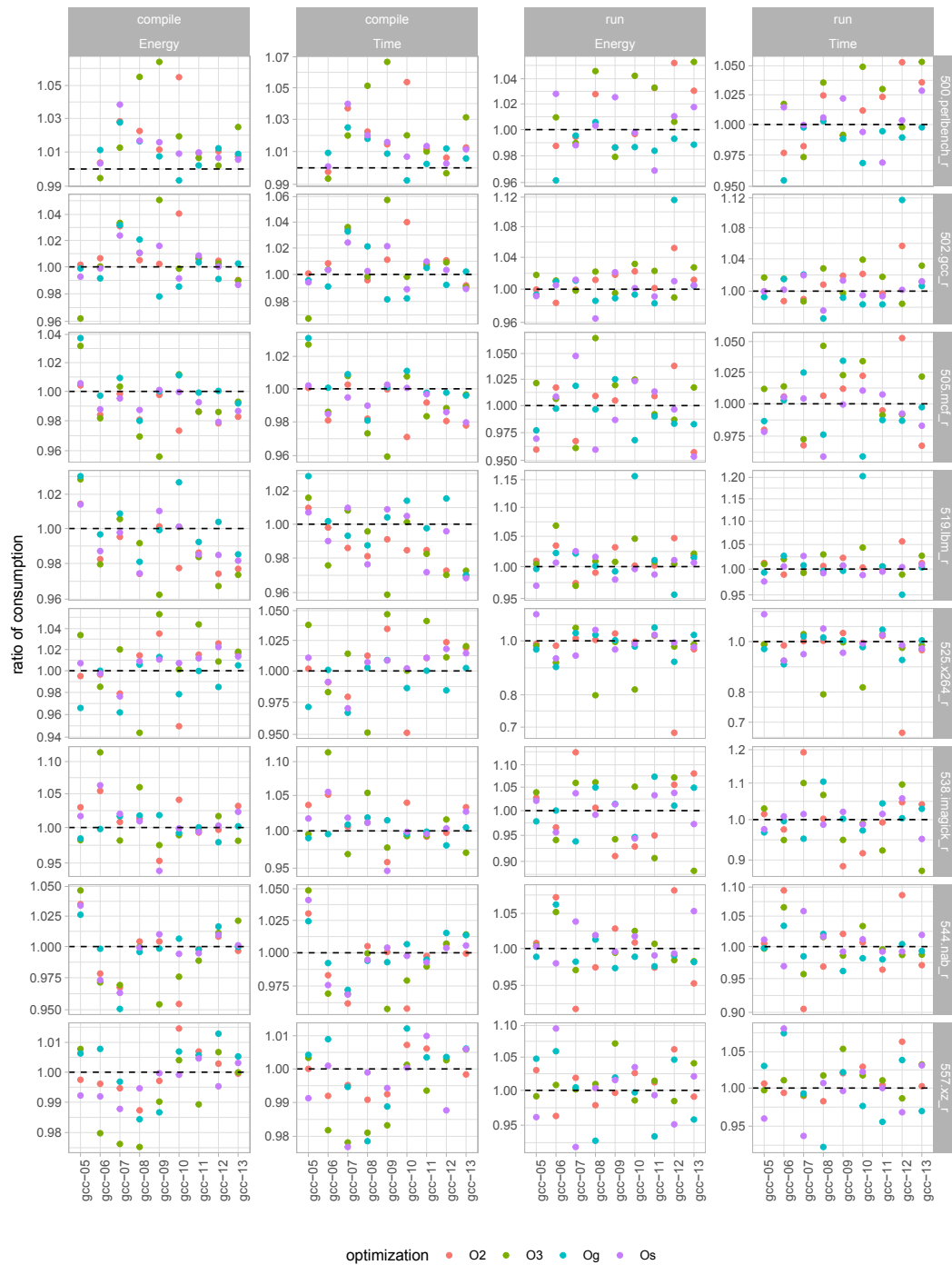


Figure 4 Predicted ratios of relative changes between GCC versions in comparison to the overall behaviour. Note that the y-axes are on a logarithmic scale and different in the 32 panels.

■ **Table 3** Changes of estimated marginal means between GCC version within ANOVA models. Only changes significant on the 5% level are shown. Note that P-values and confidence intervals are not adjusted for multiple testing.

Measuring	Response	Optimization	GCC	Estimated ratio	P-value
RUN	ENERGY	Os	4 to 5	1.029 (1.001 ; 1.057)	0.0402
RUN	TIME	Os	6 to 7	0.974 (0.949 ; 0.999)	0.0422
COMPILE	ENERGY	O2	9 to 10	1.042 (1.002 ; 1.084)	0.0383
COMPILE	ENERGY	O2	11 to 12	1.043 (1.003 ; 1.085)	0.0361
COMPILE	ENERGY	Og	4 to 5	0.953 (0.934 ; 0.974)	0.0001
COMPILE	ENERGY	Og	7 to 8	1.024 (1.004 ; 1.045)	0.0198
COMPILE	ENERGY	Os	11 to 12	1.033 (1.006 ; 1.062)	0.0160
COMPILE	TIME	Og	4 to 5	0.958 (0.940 ; 0.977)	0.0001
COMPILE	TIME	Og	7 to 8	1.022 (1.003 ; 1.041)	0.0246
COMPILE	TIME	Os	11 to 12	1.030 (1.003 ; 1.058)	0.0268

■ **Table 4** Estimated ratios and associated 95% confidence intervals between select optimizations within the ANOVA models. Note that P-values and confidence intervals are not adjusted for multiple testing.

Measuring	Response	Contrast	Estimated ratio	P-value
RUN	ENERGY	O2 / O3	1.081 (1.061 ; 1.101)	0.0001
RUN	ENERGY	Og / O2	1.147 (1.127 ; 1.167)	0.0001
RUN	ENERGY	Og / O3	1.240 (1.214 ; 1.266)	0.0001
RUN	ENERGY	Os / O2	1.210 (1.191 ; 1.230)	0.0001
RUN	TIME	O2 / O3	1.085 (1.065 ; 1.105)	0.0001
RUN	TIME	Og / O2	1.134 (1.113 ; 1.156)	0.0001
RUN	TIME	Og / O3	1.230 (1.203 ; 1.257)	0.0001
RUN	TIME	Os / O2	1.224 (1.203 ; 1.246)	0.0001
COMPILE	ENERGY	O2 / Og	1.481 (1.468 ; 1.494)	<0.0001
COMPILE	ENERGY	O2 / Os	1.128 (1.123 ; 1.134)	<0.0001
COMPILE	ENERGY	O3 / Og	1.767 (1.748 ; 1.786)	<0.0001
COMPILE	ENERGY	O3 / O2	1.193 (1.186 ; 1.200)	<0.0001
COMPILE	TIME	O2 / Og	1.429 (1.417 ; 1.442)	<0.0001
COMPILE	TIME	O2 / Os	1.113 (1.108 ; 1.119)	<0.0001
COMPILE	TIME	O3 / Og	1.679 (1.661 ; 1.697)	<0.0001
COMPILE	TIME	O3 / O2	1.175 (1.168 ; 1.181)	<0.0001

Table 5 shows the most pronounced relative changes. These are ranked according to the absolute value of the relative log ratio, which is an ad hoc measure defined by

$$\frac{\text{log ratio}}{\sqrt{\text{mean}[(\text{log ratio within measuring, response, optimization, benchmark})^2]}}$$

Thus, the relative log ratio targets changes that stick out the most in Figure 4.

4.4 Variance Parameters

Table 6 shows an example of the estimated variance parameters, specifically in the ANOVA model for RUN ENERGY. Based on this we can make estimates for the standard deviation of parameter estimates in future studies. Thus, suppose that we make a new study with n benchmarks and m replications within each optimization. Then the estimated standard deviation of the estimate for the difference $\beta(-O3, k) - \beta(-O3, j)$ for GCC versions $k \neq j$, say,

■ **Table 5** Most pronounced changes between consecutive GCC versions in comparison to the overall consumptions as ranked by the absolute value of the relative log ratio measure. Only changes with $|\log(\text{ratio})| > 0.1$ and $|\text{relative log ratio}| > 2.5$ are shown.

Measuring	Response	Optimi- zation	Benchmark	GCC version	Ratio	Relative log ratio
RUN	ENERGY	O2	525.x264_r	11 to 12	0.684	-18.65
RUN	TIME	O2	525.x264_r	11 to 12	0.668	-18.01
RUN	ENERGY	Og	502.gcc_r	11 to 12	1.116	10.15
RUN	TIME	Og	519.lbm_r	9 to 10	1.203	8.90
RUN	ENERGY	Og	519.lbm_r	9 to 10	1.157	7.11
RUN	TIME	Og	502.gcc_r	11 to 12	1.117	6.38
COMPILE	ENERGY	O3	538.imagick_r	5 to 6	1.115	4.14
COMPILE	TIME	O3	538.imagick_r	5 to 6	1.114	4.10
RUN	TIME	O2	538.imagick_r	6 to 7	1.192	3.02
RUN	TIME	O3	525.x264_r	7 to 8	0.792	-2.91
RUN	ENERGY	O3	525.x264_r	7 to 8	0.799	-2.76
RUN	ENERGY	Os	525.x264_r	4 to 5	1.116	2.65
RUN	TIME	Og	525.x264_r	5 to 6	0.904	-2.60
RUN	TIME	Os	525.x264_r	4 to 5	1.129	2.55
RUN	ENERGY	Og	525.x264_r	5 to 6	0.898	-2.53

■ **Table 6** Estimated standard deviations and correlations for random terms in ANOVA model for run energy.

Random effect	Std. Dev.	Correlation		
A_{O2}	0.0723			
A_{O3}	0.0903	0.50		
A_{Og}	0.0473	0.16	0.13	
A_{Os}	0.0267	0.13	0.30	0.06
Residual ϵ_i	0.0112			

is given by

$$\widehat{\text{sd}}(\hat{\beta}(-O3, k) - \hat{\beta}(-O3, j)) = \sqrt{\frac{2 \cdot \widehat{\text{sd}}(A_{O3})^2}{n} + \frac{2 \cdot \widehat{\text{sd}}(\epsilon_i)^2}{n \cdot m}} \approx \frac{\sqrt{2} \cdot 0.0903}{\sqrt{n}}$$

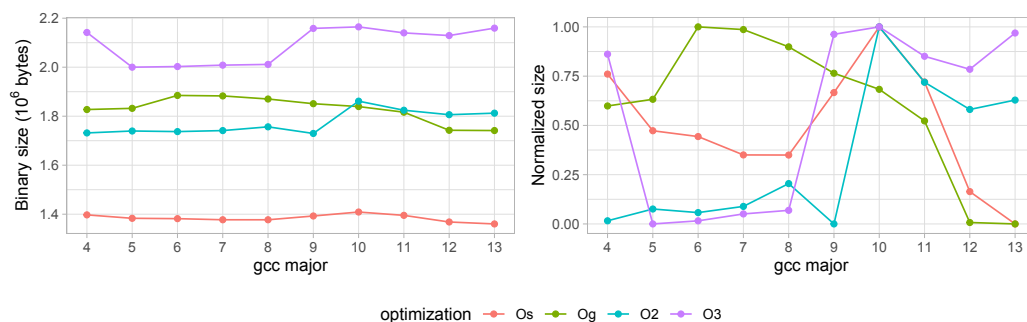
We see that this estimated standard deviation is mostly dependent on the number of benchmark programs.

4.5 Size of Compiled Benchmarks

As we can see in Figure 5, *-Os* consistently produces the smallest – in terms of binary size – resulting program, and *-O3* consistently produces the largest. *-O2* and *-Og* have similar sizes, with *-O2* starting as the smaller of the two, but ending up as the bigger of the two. Moreover, we can see that there is no clear trend towards increasing binary sizes nor decreasing binary sizes with newer GCC versions. *-Og* ends up smaller, *-O2* ends up bigger, *-O3* ends up roughly where it started, and *-Os* ends up smaller, but all of them with bumps.

4.6 Summary of Results

There is a statistically significant trend ($P = 0.0001$) of compilation getting more expensive in both time and energy, when compiling with the optimization flags *-O2*, *-O3*, and *-Os*. The trend of compilation becoming more expensive over time is coupled with *-O2* and *-O3*



■ **Figure 5** Sum of the binary sizes produced by compiling the benchmarks. Left panel shows the absolute values. Right panel shows each optimization flag normalized to a 0 – 1 range.

producing programs that use less energy and less time, with a statistically significant trend ($P \leq 0.0012$ for $-O2$ and $P = 0.0001$ for $-O3$) of running costs becoming cheaper; the effect being more drastic for $-O3$.

We can calculate the predicted relative change between two GCC versions v_1 and v_2 by taking $e^{s \cdot \log(\frac{v_2}{v_1})} - 1$, inserting the desired s from Table 2. As an example, the relative change of energy costs between version 4 and 13 for $-O3$ is $e^{-0.115 \cdot \log(\frac{13}{4})} - 1 \approx -12.68\%$, or in other words a 12.68% decrease in energy consumption.

Using this calculation, throughout all the GCC versions, comparing the version GCC-4 to the version GCC-13, and integrating values from Table 2 we obtain the following results:

Metric	Flag	Estimate	95% C.I.
RUN ENERGY	$-O3$	-12.68%	(-17.96% , -7.16%)
RUN TIME	$-O3$	-12.88%	(-18.35% , -7.16%)
COMPILE ENERGY	$-O3$	17.39%	(12.64% , 22.19%)
COMPILE TIME	$-O3$	15.19%	(10.80% , 19.90%)
RUN ENERGY	$-O2$	-9.00%	(-13.50% , -4.27%)
RUN TIME	$-O2$	-8.68%	(-13.50% , -3.47%)
COMPILE ENERGY	$-O2$	18.36%	(14.52% , 22.47%)
COMPILE TIME	$-O2$	16.01%	(12.24% , 19.90%)

In summary, using the newest version made the benchmarks improve by $\approx 13\%$ for $-O3$ and by $\approx 9\%$ for $-O2$. This was at the cost of making the compilation around $\approx 16\%$ more expensive for $-O3$ and around $\approx 17\%$ more expensive for $-O2$.

5 Discussion

For the optimization flags $-O2$ and $-O3$ the energy and time costs are decreasing with newer versions of GCC, and the compilation costs are increasing. To assess whether compiler development “pays for itself”, we must compare how much the reduced running costs are in comparison to the increased compilation costs. This is rather straightforward: for every benchmark and optimization flag, compiling is cheaper (in both time and energy) than a single run of the corresponding benchmark. Even the most expensive compilation (*502.gcc_r* with $-O3$, around 3.5kJ and 45s) is comparable to the cheapest run (*525.x264_r* with $-O3$, around 2.2kJ and 27s). Consequently, the extra compilation cost of newer GCC versions is “paid off” by the second execution. This supports the advice of using the newest GCC version, as long as $-O2$ or $-O3$ is employed.

By contrast, *-Os* and *-Og* do not exhibit clear trends towards lower execution costs, which is unsurprising given that they prioritize, respectively, binary size and debugging experience. As shown in Figure 5, newer GCC versions do not necessarily produce smaller binaries overall, but *-Os* consistently yields the smallest binaries among the optimization levels.

5.1 Benchmarks Differ – No Silver Bullet for Optimization

Certain benchmarks seem to benefit more from newer versions of GCC than others. In particular *525.x264_r* starts out being one of the benchmarks with the highest running cost, to one of the ones with the lowest running costs when optimized with *-O2* or *-O3*. A lot of these improvements happen in sudden jumps towards lower running costs between consecutive versions, implying a certain version of GCC made an optimization that improved a particular benchmark.

From Table 5 we see that *525.x264_r* has two large additional drops in running cost compared to other benchmarks, between GCC-11 and GCC-12 for *-O2*, and between GCC-7 and GCC-8 for *-O3*. Both of these releases coincide with changes to GCC’s vectorization, including reworked cost metrics in GCC-8, and auto-vectorization being enabled for *-O2* in GCC-12.³ A further jump for *525.x264_r* between GCC-9 and GCC-10 for *-O3* is visible in Figure 2. We suspect this is related to several reported missing optimizations (bug IDs: 64745, 89653, 88915, 57169, and 89386) that were flagged as fixed in GCC-10 relating to vectorization (one of them even explicitly mentions *525.x264_r*).

We therefore hypothesise that the sudden improvements observed for *525.x264_r* are largely due to enhanced vectorization in newer GCC versions. This could be tested by repeating our experiments with vectorization disabled.

There are other benchmarks behaving differently than the mean, but all of them less significantly than *525.x264_r*. In principle all programs can benefit from improved vectorization; *525.x264_r* stands out as it is particularly amiable to it. Since SPEC is designed to include a series of representative benchmarks, we do not expect the improvements we see to be restricted to any single workload, such as *525.x264_r*.

There are some consecutive versions where compilation became statistically significantly more expensive for the compilation of all benchmarks. Between GCC-9 and GCC-10, and between GCC-11 and GCC-12, the *-O2* compilations became $\approx 4\%$ more expensive, energy-wise. These versions coincide with function inlining and vectorization becoming enabled for *-O2*, so we hypothesize the increased costs is at least partially due to this.

5.2 Result Comparison with Prior Findings

Previous studies have investigated how compiler or interpreter versions affect energy consumption across different programming languages [2, 11, 16, 17]. Among these, the works of Jiménez et al. [11] and Azevedo [2] are most directly comparable to our work because they include C and study the effect of compiler version on energy consumption. In contrast to our study, neither of them identifies a general downward trend in energy usage for GCC [11] nor Clang [2].

Their experimental designs differ considerably from ours. Both studies use the Computer Language Benchmarks Game (CLBG) and adopt benchmark-specific compiler configurations prescribed by CLBG. While Azevedo does not report the exact flags, the replication package provided by Jiménez et al. shows that, across GCC versions 7–11, all nine C benchmarks are compiled with *-O3*, and eight of the nine use OpenMP.

³ See <https://gcc.gnu.org/gcc-8/changes.html> and <https://gcc.gnu.org/gcc-12/changes.html>.

These differences in experimental design provide several possible explanations for the different trends observed. One possible explanation for the difference in observed trends concerns the experimental coverage. Jiménez et al. characterize GCC evolution over a narrower range of versions (7–11) and under a single highly optimized configuration, whereas we study GCC 4–13 and systematically evaluate several optimization levels. Our results show that the longitudinal trend depends on the optimization level, suggesting that broader version coverage and multiple optimization configurations may reveal effects that are less apparent over a shorter interval or under a single configuration.

Benchmark selection may also contribute to the difference. Both Azevedo and Jiménez et al. use CLBG, whereas we use SPEC CPU 2017. Our results show substantial variation across SPEC CPU 2017 benchmarks in how they benefit from newer GCC versions, suggesting that benchmark selection may influence whether a general version-dependent trend is observed. We hypothesize that the CLBG benchmarks may already be sufficiently hand-optimized that they leave less opportunity for newer compiler optimizations. The presence of manually written vectorization code, including uses of `immintrin.h`, `emmintrin.h`, and `v2dt`, in Jiménez et al.’s replication package [11] supports this interpretation. This is particularly relevant because our results suggest that improvements in auto-vectorization account for a substantial part of the performance gains observed for SPEC CPU 2017. If the code is already manually vectorized, there is less opportunity for the compiler’s auto-vectorization to provide additional performance gains [15]. In contrast to CLBG benchmarks, SPEC CPU 2017 was designed with portability across architectures as an explicit goal, the benchmarks were carefully selected to have the ability to have their platform or architecture-dependent code disabled [3].

A further difference concerns parallelism. Most C benchmarks used by Jiménez et al. [11] employ OpenMP-based parallelism, whereas we deliberately disable OpenMP-level parallelism to reduce variation unrelated to compiler evolution, while retaining instruction-level parallelism (such as vectorization), which *is* affected by compiler evolution. Parallel execution introduces additional sources of variation, including thread scheduling, synchronization overhead, and workload distribution, which may increase benchmark-specific variability and make an underlying compiler-version trend less apparent.

Beyond the overall version trend, prior work also helps interpret some benchmark-specific changes observed in our results. In Hebbar et al. [8], GCC version 5.5.0 was outperformed by Intel’s compilers on the SPEC 2017 benchmarks. The authors attribute this difference in part to more effective use of advanced vectorization. In particular, for *525.x264_r* the authors find that Intel outperforms GCC by 155%, and for our specific selection of benchmarks, Intel generally outperforms GCC by more than 40%, except for *500.perlbench_r* and *502.gcc_r* where GCC performs slightly better. Together with our hypothesis for the drastic *525.x264_r* improvement, this suggests that newer GCC versions may be reducing the performance gap previously observed between GCC and Intel’s compilers in this area – at least for *525.x264_r*.

5.3 Threats to Validity, Limitations, and Future Work

Our results come from a single computer with an x86-64 Skylake microprocessor, and how other architecture evolves is not clear. Since many of the improvements we see might be linked to improved vectorization, other architectures, particularly ones without vectorization support, might benefit less from newer versions. We would expect either no link between newer versions or a weaker one. Other machines with same architecture and setup may also slightly different results, but the trend is expected to be unchanged.

Given that we see so much variation in how the different benchmarks behave, we cannot necessarily rule out the possibility that different benchmarks would give different results. However, as SPEC is a trusted industry-standard benchmark measuring the performance of real-world programs and the chosen subset covers a wide range of different program profiles, we would hypothesise that our results hold for similar real-world programs written in C.

GCC is a collection of multiple compilers and SPEC is a suite of benchmarks written in several programming languages. A natural future work would be to compare our results to the other two languages SPEC and GCC have in common, namely Fortran and C++. However, since they share a back-end the difference would probably not be drastic.

The selection of the different versions of GCC was done with the expectation that performance between minor versions within the same major version of GCC would have few changes. Whether this is true or not is left for future work.

We choose to disable OpenMP in our benchmarks, effectively disabling multi-threading. Parallelism of all sorts is especially interesting for energy consumption. However, we believe that it was correct to disable it, as it better isolates the impact of the change in version on performance. It does leave the question of whether we get the same results, for time but especially energy, if OpenMP was enabled.

6 Conclusion

Does GCC C compiler development pay for itself? That is, do newer versions of GCC reduce the time and energy costs of running compiled programs, at a low-enough increased costs of compiling the programs? Using SPEC CPU 2017 on a x86-64 Skylake machine, with 10 different GCC versions spanning from *gcc-04.09.03* to *gcc-13.03.00*, we find that it does for the optimisations levels *-O2* and *-O3*.

There is a statistically significant trend for both energy and time costs decreasing when compiled with *-O2* and *-O3*, at the costs of a statistically significant increasing trend of compilation costs (energy and time) for the same optimizations flags, all with $P \leq 0.0012$. Throughout all the versions tested, benchmarks improve by $\approx 13\%$ when compiled with *-O3*, and $\approx 9\%$ when compiled with *-O2*. This is at the additional cost of compilation costs increasing by $\approx 16\%$ and $\approx 17\%$ for *-O3* and *-O2* respectively.

In practice, a program is compiled only a few times, but runs many times. We note that the cost of compiling a benchmark was always lower than the cost of running the corresponding benchmark. Therefore, the additional costs of using a newer version of GCC are paid off by the second running of the benchmark.

No statistically significant trend of improvement was found for the benchmarks compiled with *-Os* and *-Og*. This is not surprising, as neither of them has a goal of optimization. Rather, they optimize for the debugging experience and for space usage.

Certain benchmarks benefit more than others from newer versions of GCC, notably *525.x264_r*, which starts as one of the most expensive to run and becomes one of the cheapest when optimized with *-O2* or *-O3*. We hypothesize this substantial improvement is in large part due to improvements in auto-vectorization; the GCC versions where we observe the largest performance gains coincide with changes in GCC's auto-vectorization feature. This suggests adapting performance-critical code to better match patterns targeted by auto-vectorization, or explicitly writing vectorization code oneself, can yield large performance benefits. More broadly, it highlights that how much a compiler can optimize any given program is dependent on the type of program being compiled.

Although SPEC benchmarks are indeed just benchmarks, they consist of many real-world programs and are designed to measure the performance of a compiler (together with the processor and the memory subsystem). Therefore, we believe that the results carry over to other C programs. Continuing, we cannot say for sure whether versions after GCC-13 continue reducing costs, but given newer CPUs with newer instructions for GCC to support, we suspect it would.

Data Availability Statement

All the data together with analysis scripts and scripts to replicate the experiment is available on: <https://doi.org/10.6084/m9.figshare.32316525>. The link has documentation for the data-set, and there is a `README.md` file for how to read, and do the experiment together with the analysis.

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. IEEE, 2014. doi:10.1109/IGCC.2014.7039169.
- 2 Pedro Miguel Oliveira Azevedo. Ranking compiler versions by energy efficiency. Master’s thesis, Universidade do Porto (Portugal), 2023.
- 3 James Bucek, Klaus-Dieter Lange, and J  akim v. Kistowski. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE ’18, pages 41–42, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3185768.3185771.
- 4 Michael Franz. Compiler optimizations should pay for themselves. In *Proceedings of the Joint Modular Languages Conference*, pages 111–121, 1994.
- 5 Kyriakos Georgiou, Craig Blackmore, Samuel Xavier-de Souza, and Kerstin Eder. Less is more: Exploiting the standard compiler optimization levels for better performance and energy consumption. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, pages 35–42, 2018. doi:10.1145/3207719.3207727.
- 6 Alberto Gordillo, Coral Calero, M   Angeles Moraga, F  lix Garc  a, Jo  o Paulo Fernandes, Rui Abreu, and Jo  o Saraiva. Programming languages ranking based on energy measurements. *Software Quality Journal*, 32(4):1539–1580, 2024. doi:10.1007/S11219-024-09690-4.
- 7 Marcus H  hnel, Bj  rn D  bel, Marcus V  lp, and Hermann H  rtig. Measuring energy consumption for short code paths using RAPL. *ACM SIGMETRICS Performance Evaluation Review*, 40(3):13–17, 2012. doi:10.1145/2425248.2425252.
- 8 Ranjan Hebbar SR and Aleksandar Milenkovi  . SPEC CPU2017: Performance, event, and energy characterization on the core i7-8700K. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 111–118, 2019. doi:10.1145/3297663.3310314.
- 9 Intel Corporation. *Intel   64 and IA-32 Architectures Software Developer’s Manual, Volume 3B: System Programming Guide, Part 2*. Intel Corporation. Continues the coverage of system programming subjects begun in volume 3A. Volume 3B covers thermal and power management features, debugging, and performance monitoring. URL: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>.
- 10 Armin J  ger, Jan-Patrick Lehr, and Christian Bischof. The influence of two modern compiler infrastructures on the energy consumption of the HPCG benchmark. *SICS Software-Intensive Cyber-Physical Systems*, 34(1):53–60, 2019. doi:10.1007/S00450-018-0393-8.
- 11 Elisa Jim  nez, Alberto Gordillo, Coral Calero, M   Angeles Moraga, and F  lix Garc  a. Does the compiler or interpreter version influence the energy consumption of programming languages? *Science of Computer Programming*, 243:103270, 2025. doi:10.1016/j.scico.2025.103270.

- 12 Nicolas van Kempen, Hyuk-Je Kwon, Dung Tuan Nguyen, and Emery D. Berger. It's not easy being green: On the energy efficiency of programming languages. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1553–1565, 2025. doi:10.1109/ASE63991.2025.00131.
- 13 Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. RAPL in action: Experiences in using rapl for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 3(2):1–26, 2018. doi:10.1145/3177754.
- 14 Maja Hanne Kirkeby, Bernardo Santos, João Paulo Fernandes, and Alberto Pardo. Compiling haskell for energy efficiency: Empirical analysis of individual transformations. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, pages 1104–1113, 2024. doi:10.1145/3605098.3635915.
- 15 Charith Mendis, Ajay Jain, Paras Jain, and Saman Amarasinghe. Revec: program rejuvenation through revectorization. In *Proceedings of the 28th International Conference on Compiler Construction*, CC 2019, pages 29–41, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3302516.3307357.
- 16 César Olvera Espinosa. Evolving trends in software performance and energy consumption: An analysis of compilers and interpreters over time. Master's thesis, Aalto University, 2024.
- 17 Zakaria Ournani, Mohammed Chakib Belgaid, Romain Rouvoy, Pierre Rust, and Joël Penhoat. Evaluating the impact of java virtual machines on energy consumption. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2021. doi:10.1145/3475716.3475774.
- 18 James Pallister, Simon J Hollis, and Jeremy Bennett. Identifying compiler options to minimize energy consumption for embedded platforms. *The Computer Journal*, 58(1):95–109, 2015. doi:10.1093/COMJNL/BXT129.
- 19 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. doi:10.1016/j.scico.2021.102609.
- 20 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. doi:10.1109/MM.2012.12.
- 21 Standard Performance Evaluation Corporation (SPEC). SPEC CPU® 2017 Benchmark Suite. <https://www.spec.org/cpu2017/Docs/>, 2017. Accessed: 2026-05-04.