

Where Does a Compiler Spend Its Time?

A Study of GCC

Meet R Patel
Enrollment No.: 241133107036

Chauhan Pratham
Enrollment No.: 231130107006

Parmar Kavya Kishorbhai
Enrollment No.: 231130107043

Satyam Tulsidas Dhiravani
Enrollment No.: 241133107011

Department of Computer Engineering
SAL College of Engineering, Ahmedabad, Gujarat, India

Abstract

Compilers are usually taught as a sequence of phases, yet textbooks say little about how much each phase costs inside a production compiler. This paper measures it. Using the built-in timing report of GCC 13.3, we group 139 internal timers into seven stages that map onto the classical phases and measure them on two generated code bases (14,004 and 12,807 lines) at five optimization levels, on a scaling series of up to 56,004 lines, and on eight benchmark programs. At -O0 the back end consumes 72% of compiler time and the front end 19%; at -O2 the middle end alone takes 53% and the front end only 4%. Front-end cost stays nearly constant (0.38– 0.46 s) while total compile time rises $5.0\times$ from -O0 to -O2. Compile time grows almost linearly with source size (fitted exponent 1.02 at -O0 and 0.95 at -O2, $R^2 > 0.999$) over a 16-fold range. Relative to -O0, -O2 gives a geometric-mean speedup of $3.05\times$ (range 1.64– $7.75\times$); -O3 adds only 3% more, while raising compile time by 2– 15% and program code size from $1.14\times$ to $1.30\times$. All 40 optimized builds produced identical program output. We discuss implications for teaching and for choosing optimization levels, and we release all scripts and raw data.

Keywords: Compiler phases, GCC, compilation cost, optimization levels, empirical study, intermediate representation, compile-time scaling

1. Introduction

The traditional description of a compiler as a pipeline of lexical analysis, syntax analysis, semantic analysis, intermediate code generation, optimization, and code generation is a cornerstone of compiler education [1], [2]. The model explains what each stage does, but it does not say how much each stage costs. Students therefore often assume that scanning and parsing, the first stages they study in detail, are where a compiler spends its effort.

Production compilers expose enough instrumentation to test this assumption. GCC can report the time spent in each of its internal activities, and its pass structure is documented [6]. This paper uses that instrumentation to

answer four research questions:

- **RQ1.** How is compiler CPU time distributed across the classical phases, and how does the distribution change with the optimization level?
- **RQ2.** How does compile time scale with source size?
- **RQ3.** What does each optimization level buy at run time, and at what cost in compile time and code size?
- **RQ4.** How does the size of the program representation change as it moves through the pipeline?

The contributions are: (i) a documented mapping from 139 GCC timers to seven stages aligned with the classical phases; (ii)

measurements for five optimization levels on two independent generated workloads; (iii) a scaling study over a 16-fold range of source size; (iv) a runtime, code-size and correctness comparison on eight benchmarks; and (v) a public release of all scripts and raw data so that the study can be repeated.

2. Background and Related Work

Phases. The classical model separates lexical, syntax and semantic analysis (the analysis side) from intermediate code generation, optimization and code generation (the synthesis side) [1], [2]. Optimization techniques are catalogued in detail by Muchnick [3]. Modern infrastructures such as LLVM organize compilation around a common intermediate representation so that front ends, optimizers and back ends can be reused independently [4].

GCC pipeline. GCC parses C into a front-end tree, lowers it to the GIMPLE intermediate representation, runs interprocedural and GIMPLE optimization passes, expands to the RTL representation, and then performs RTL optimization, register allocation, scheduling and assembly output [6]. Register allocation, which we measure separately, is classically formulated as graph coloring [5].

Optimization levels. The levels -O0 to -O3 and -Os bundle sets of optimization flags; their exact contents are specified in the GCC manual [7]. Choosing among them is a long-standing practical problem, and a large literature applies machine learning to select optimizations automatically [10]. Understanding the baseline cost of each level, as measured here, is a prerequisite for such work.

Correctness. Optimization must preserve program behavior. Testing has found miscompilation bugs in widely used C compilers [8], and CompCert demonstrates that a realistic

C compiler can be formally verified [9]. We include a simple behavior-preservation check on our benchmarks, but we do not claim it as a correctness proof.

3. Methodology

3.1 Environment

All measurements used GCC 13.3.0 (Ubuntu 13.3.0-6ubuntu2~24.04.1) on Ubuntu 24.04 in a Linux container with one virtual CPU (Intel Xeon, 2.10 GHz), 4 GB RAM and kernel 6.18. Programs were compiled with only the optimization level changed; no architecture-specific, link-time or profile-guided options were used. Because the machine has a single core, experiments were run strictly one at a time.

3.2 Mapping Timers to Phases

GCC's `-ftime-report` option prints CPU time (user + system) for each internal timer. Across all runs we observed 144 distinct timer names. Three are inclusive phase totals and two (callgraph functions expansion, callgraph IPA passes) are inclusive aggregates of other timers; we excluded these five to avoid double counting. The remaining 139 timers were assigned to seven groups (Fig. 1) using the names and roles documented in the GCC internals manual [6]. As a validation, the sum of our groups equals GCC's own reported total in every run (to within 0.01 s).

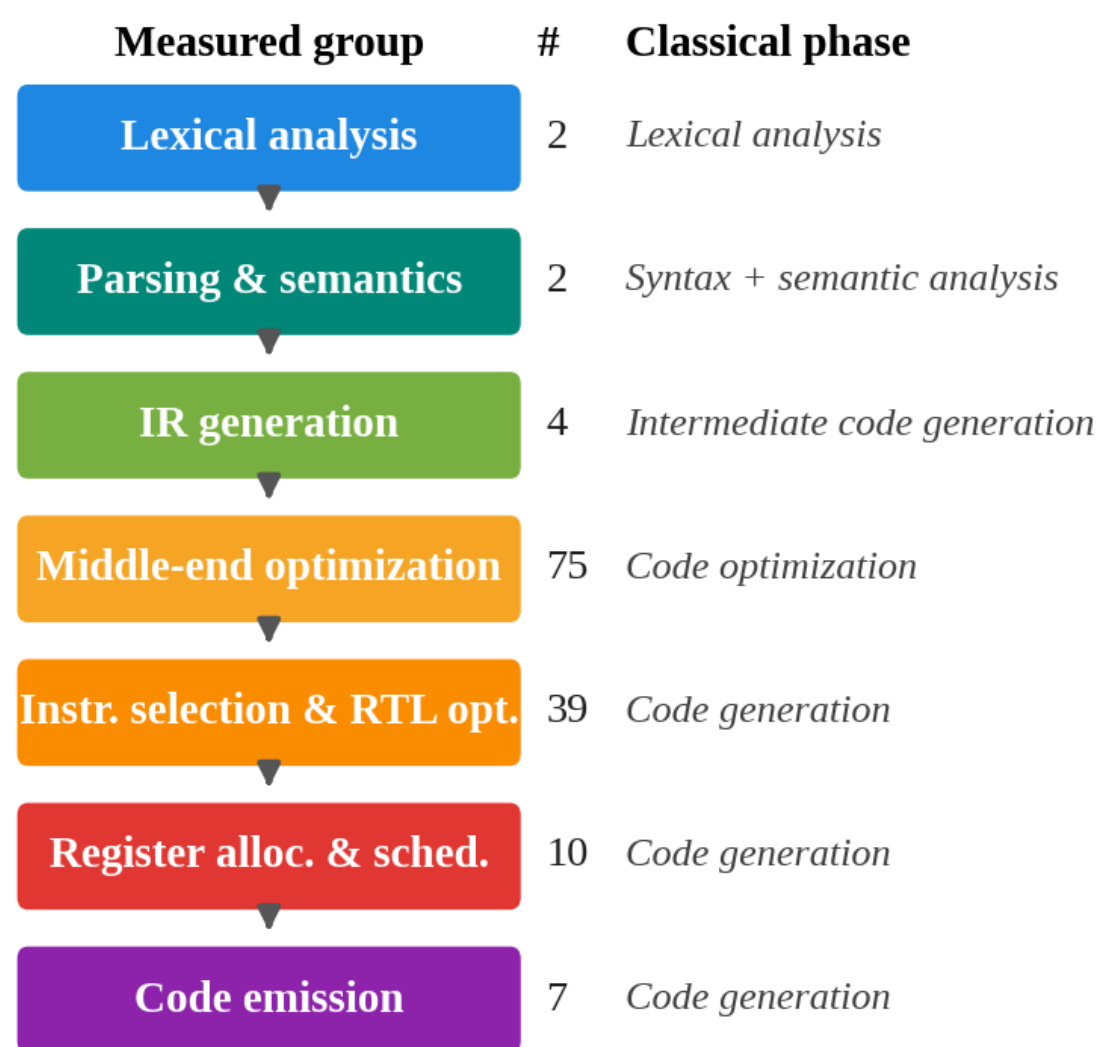


Fig. 1. Mapping of GCC timers to the seven measured groups and the classical phases

The mapping involves judgment. GCC's C front end interleaves parsing and semantic checking, so we report them as one group. Back-end work is split into instruction selection with RTL optimization, register allocation with scheduling, and code emission. A few shared timers (for example dominance computation) could belong to more than one group; we assigned them to the middle end.

3.3 Workloads

Table 1. Workloads

Name	Description	Size
W1	Generated C: arithmetic, loops, branches, small arrays	1,000 functions; 14,004 lines
W2	Generated C: structs, switch, array indexing	800 functions; 12,807 lines
S1- S5	W1 generator, N = 250 ... 4000 functions	3,504 ... 56,004 lines
B1- B8	fib, sieve, matmul, nbody, mandel, qsort, fnv, collatz	98- 451 tokens each

The generators use a fixed random seed. Generated functions are independent of one another, which isolates per-function costs but does not exercise whole-program effects; Section 6 returns to this. The eight benchmarks are small, self-contained C kernels that print a checksum so that results can be compared across builds.

3.4 Measurement Protocol

For RQ1 each workload was compiled to an object file five times per level and the median of each group is reported. For RQ2 each size was compiled three times at -O0 and -O2 and a power law $T = a \cdot N^k$ was fitted by least squares on log-log data. For RQ3 each benchmark was compiled and linked at each level, run five times, and the median wall-clock time reported; speedup is the -O0 time divided by the time at the given level, and means across programs are geometric. Program code size is the text segment reported by size minus that of an empty program (about 1.2 KB). For RQ4 we counted source tokens with a regular-expression tokenizer, GIMPLE statements in GCC's dump files (lines with an assignment, call, branch or return), and instructions in the generated assembly.

4. Results

4.1 Phase Distribution (RQ1)

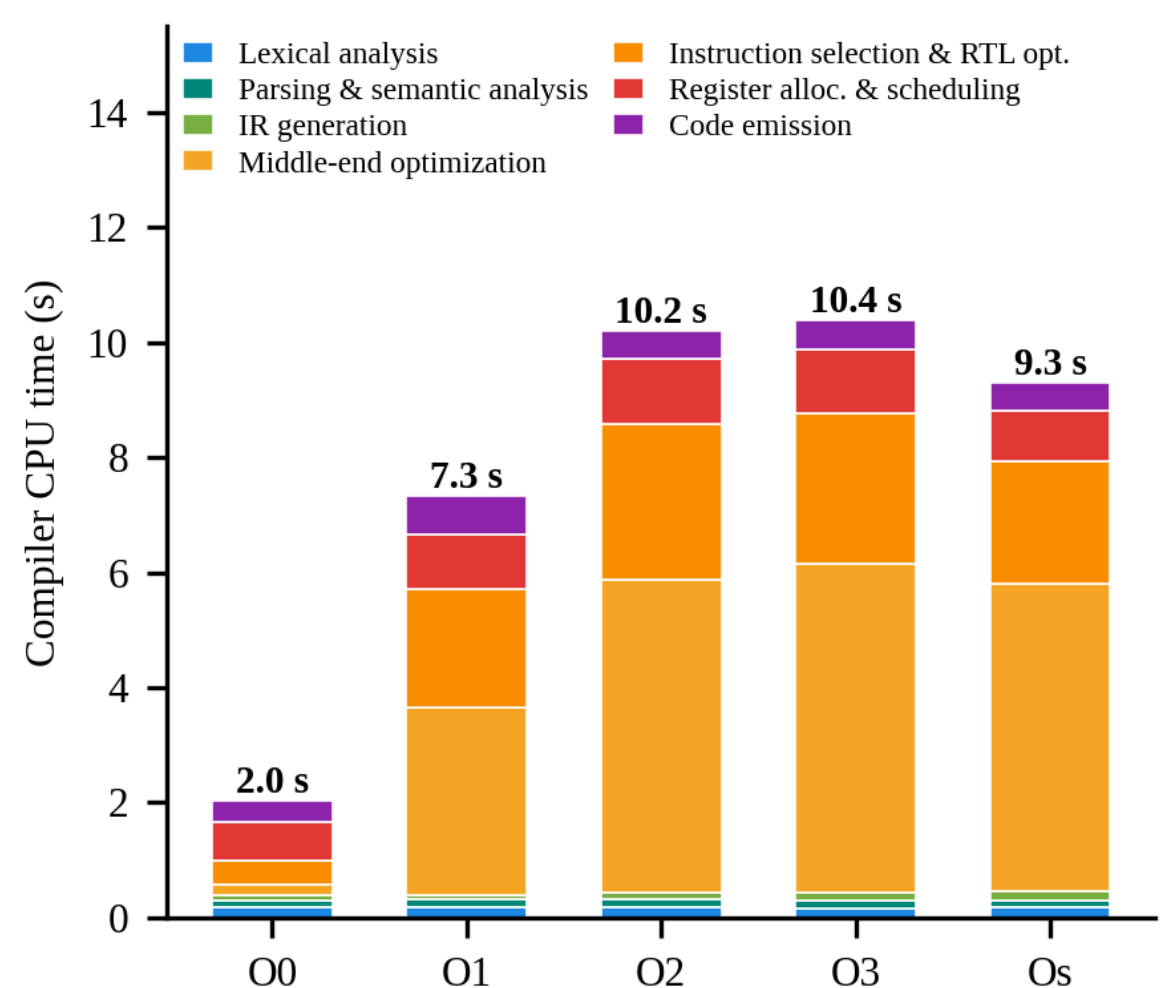


Fig. 2. Compiler CPU time by stage for workload W1

Fig. 2 and Table 2 show the result for W1. At -O0 the compiler spends only 18.7% of its time in the three front-end stages (lexical analysis, parsing and semantic analysis, IR generation) and 71.9% in the back end. Register allocation is the single largest stage (33.0%). At -O1 and above, the middle end dominates: it takes 44.3% at -O1 and 53.4% at -O2, while the front

end falls to 4.3%.

Table 2. Share of compiler CPU time per stage, W1 (%)

Stage	O0	O1	O2	O3	Os
Lexical analysis	8.4	2.3	1.7	1.5	1.9
Parsing & semantics	5.9	1.9	1.4	1.3	1.3
IR generation	4.4	1.2	1.3	1.3	1.7
Middle-end opt.	9.4	44.3	53.4	54.9	57.5
Instr. sel. & RTL	20.7	28.1	26.5	25.2	23.0
Reg. alloc. & sched.	33.0	13.0	11.2	10.7	9.5
Code emission	18.2	9.1	4.6	4.9	5.1
Total CPU time (s)	2.03	7.33	10.19	10.38	9.29
Cost vs. O0 (×)	1.00	3.61	5.02	5.11	4.58

The absolute front-end cost is almost independent of the optimization level: 0.38 s at -O0 and 0.44 s at -O2. Essentially all of the extra compile time at higher levels comes from optimization work. The back end also grows, from 1.46 s to 4.31 s (3.0×), so optimization makes register allocation and RTL passes more expensive as well as adding GIMPLE passes. Workload W2 shows the same pattern (front end 19.7% share of back end at -O0 vs. middle end 49.2% at -O2; total cost 4.65× at -O2), which suggests the result is not an artifact of one code generator.

4.2 Scaling (RQ2)

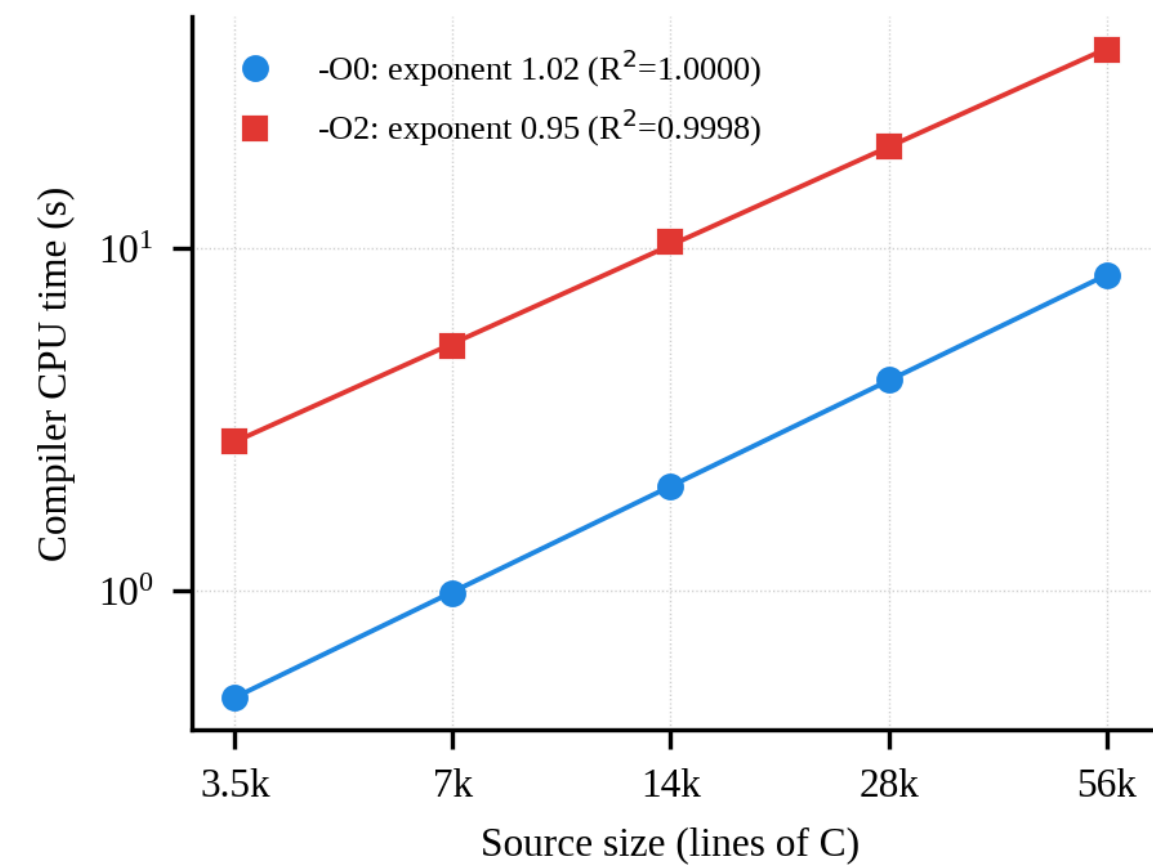


Fig. 3. Compile time versus source size (log- log)

Fig. 3 shows that compile time grew almost linearly with source size: the fitted exponent is 1.02 at -O0 and 0.95 at -O2, with R^2 above 0.999 in both cases. Throughput was 6.7– 7.2 thousand lines per second at -O0 and 1.3– 1.5

thousand lines per second at -O2, about five times lower. At -O2 throughput rose slightly with size (from 1,279 to 1,472 lines/s), which is consistent with a small fixed start-up cost being amortized, although we did not isolate it. Fits for the three largest groups at -O2 gave exponents of 0.98 (middle end), 0.98 (RTL) and 0.84 (register allocation); we do not interpret exponents of the smaller groups because their times are close to the 10 ms timer resolution.

4.3 Runtime, Size and Cost (RQ3)

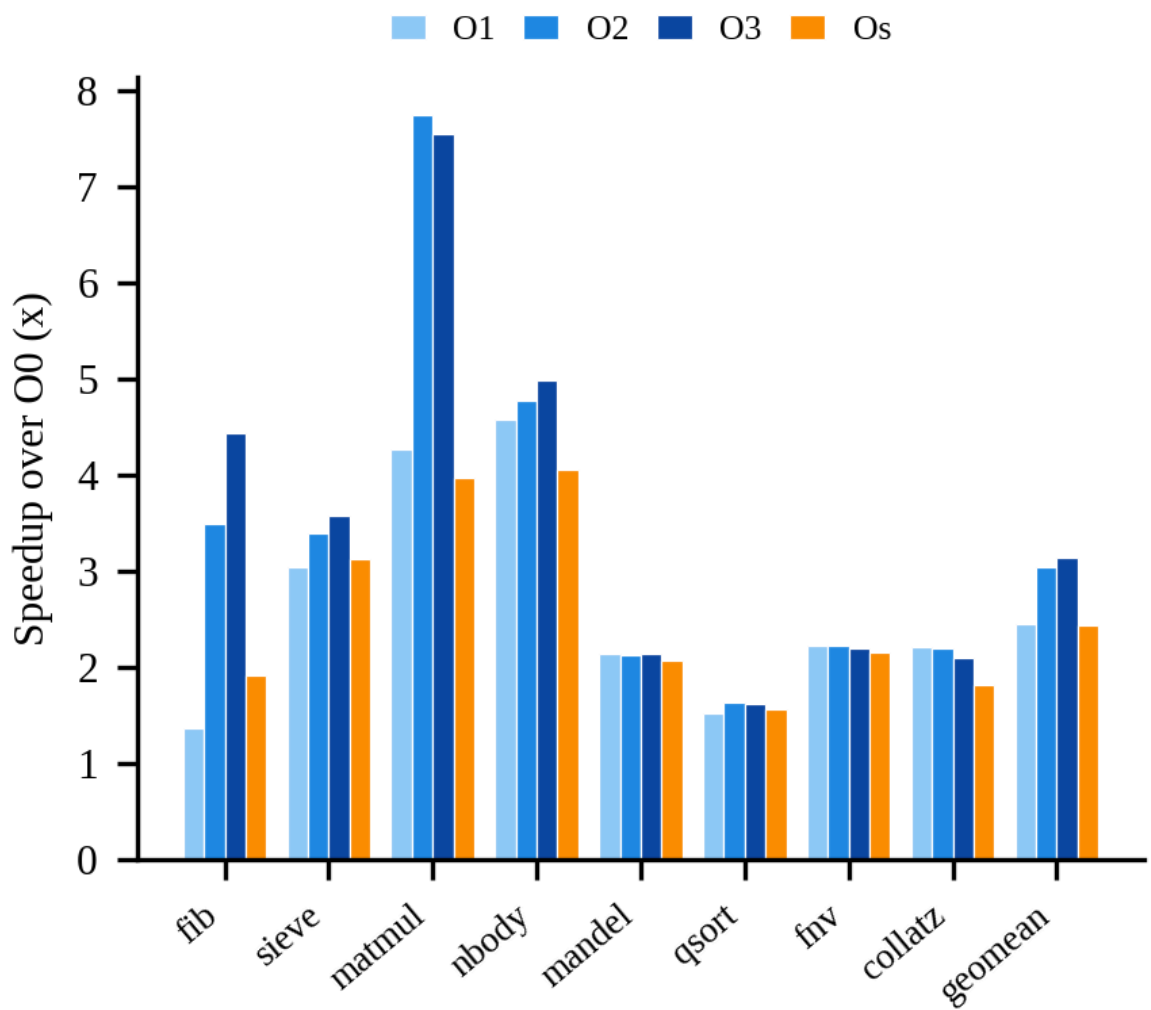


Fig. 4. Run-time speedup over -O0 for eight benchmarks

Fig. 4 shows run-time speedups. The geometric-mean speedup over -O0 is 2.45× at -O1, 3.05× at -O2, 3.14× at -O3 and 2.43× at -Os. At -O2 the speedup ranges from 1.64× (qsort) to 7.75× (matmul). Gains are not uniform across levels: fib improves from 3.49× at -O2 to 4.43× at -O3, whereas collatz slows slightly from 2.20× to 2.09×. We did not isolate which transformations are responsible.

Table 3. Cost and benefit of each level (relative to -O0)

Level	Compile cost W1 / W2 (×)	Speedup (geomean)	Program text (×)
-O1	3.61 / 3.34	2.45	0.83
-O2	5.02 / 4.65	3.05	1.14
-O3	5.11 / 5.34	3.14	1.30
-Os	4.58 / 4.08	2.43	0.77

Table 3 combines the three costs and benefits.

-O1 achieves about 80% of the -O2 speedup (on a logarithmic scale) at roughly 70% of its compile cost on W1. -O3 gives only 3% more speed than -O2 on average while program text grows from 1.14× to 1.30× and compile cost rises by 2% (W1) to 15% (W2). -Os gives a similar speedup to -O1 and the smallest code (0.77×).

Behavior preservation. All eight benchmarks printed identical output at all five levels (40 of 40 builds), including the two floating-point programs. This is a sanity check, not a proof, and it covers only these programs on one machine.

4.4 Representation Size (RQ4)

Table 4. Size of the representation at each stage

Program	Tokens	GIMPLE -O0	GIMPLE -O2	Asm -O0	Asm -O2
fib	50	16	114	36	327
sieve	163	54	21	90	54
matmul	239	53	67	161	168
nbody	451	94	69	398	106
mandel	148	45	22	87	55
qsort	290	92	70	191	192
fNV	134	33	20	63	37
collatz	98	26	12	48	38
Total	1573	413	395	1074	977

Table 4 follows the eight programs from source tokens to GIMPLE statements to assembly instructions. Over the whole suite, optimization reduces GIMPLE statements by 4% (413 to 395) and assembly instructions by 9% (1,074 to 977), but the per-program picture varies widely. The sieve shrinks by 61% at the GIMPLE level and nbody by 73% at the assembly level, while fib grows by a factor of 7.1 (GIMPLE) and 9.1 (assembly), which is consistent with inlining or unrolling of its recursive calls, although we did not verify the cause. A smaller representation therefore does not always follow optimization; it depends on the program.

5. Discussion

The phases students study first are the cheapest. In GCC the lexical, syntax and semantic stages together account for under 10% of compiler time at -O1 and above (about 6% at -O2), and under 15% at -O0. This does not make them unimportant, since they determine correctness and diagnostics, but it explains why compiler engineering effort concentrates on optimization and code generation.

Optimization cost is not only a middle-end matter. The middle end grows from 0.19 s to 5.44 s between -O0 and -O2, yet the back end also triples. The classical split into “ optimization” and “ code generation” therefore understates how much the quality of the intermediate form affects later stages.

Practical guidance. On this suite, -O2 captures nearly all of the benefit of -O3 at lower code size, and -O1 is an economical choice when build time dominates. These are measurements for one compiler and a small benchmark set, not universal rules; the per-program variation in Fig. 4 is a reminder that the best level depends on the program.

Teaching. The seven-group mapping in Fig. 1 can be reused in courses: students can run -ftime-report on their own programs and relate the measured stages to the phases they study. The per-phase measurements also give concrete numbers for discussing why compilers offer multiple optimization levels.

6. Threats to Validity

Single compiler and machine. All data come from GCC 13.3.0 on one single-core virtualized machine. Results for Clang/LLVM [4], other versions, other architectures or multi-core builds may differ. **Synthetic workloads.** The generated code consists of independent

functions. This favors linear scaling; programs with large call graphs or heavy interprocedural optimization may scale differently, and we did not test link-time optimization. **Small benchmarks.** The eight kernels run for 0.02–0.8 s and are not representative of large applications. Median run-to-run variation was 1% but reached 12.9% for one configuration (nbody at -O2). **Timer resolution.** GCC reports times in 10 ms steps, so small groups are coarse; we draw conclusions only from large groups and totals. **Classification.** Assigning timers to groups required judgment, and other reasonable assignments would shift a few percentage points between neighboring groups. **Metrics.** GIMPLE statement counts use a line-based heuristic, and assembly instruction counts include compiler-generated sequences.

7. Conclusion and Future Work

We measured where GCC 13.3 spends its time across the classical compilation phases. The front end is cheap and nearly constant in cost; optimization is what makes compilation expensive, and it does so across the middle and back ends. Compile time scaled almost linearly with source size on our workloads, -O2 gave a 3.05× average speedup at about five times the compile cost of -O0, and -O3 added little on average. The study also demonstrates that the textbook phase model can be tied directly to measurable quantities in a production compiler.

Future work includes repeating the study with Clang/LLVM, whose pass-timing facilities allow a similar mapping; adding programs with large call graphs and link-time optimization; measuring the effect of individual passes by toggling flags; and repeating the measurements on several machines to quantify hardware variation.

8. Reproducibility and Disclosure

All generators, benchmark sources, measurement scripts, raw timing data and the analysis code are provided in the supplementary archive (compiler_phase_study.zip). The study is repeated by running run_AB.py, run_CD.py and report.py in that order on a machine with GCC and Python 3; absolute times will differ by machine, and the proportions are the quantities of interest. **AI assistance:** the experiment scripts, analysis code and a draft of this manuscript were prepared with the assistance of an AI system (Claude, Anthropic); the measurements were run by that system in the environment described in Section 3.1. The author is responsible for the content.

References

- [1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, Compilers: Principles, Techniques, and Tools, 2nd ed. Boston, MA, USA: Pearson/Addison-Wesley, 2006.
- [2] K. D. Cooper and L. Torczon, Engineering a Compiler, 3rd ed. Cambridge, MA, USA: Morgan Kaufmann, 2022.
- [3] S. S. Muchnick, Advanced Compiler Design and Implementation. San Francisco, CA, USA: Morgan Kaufmann, 1997.
- [4] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in Proc. Int. Symp. Code Generation and Optimization (CGO), Palo Alto, CA, USA, Mar. 2004, pp. 75–86.
- [5] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, “Register allocation via coloring,” Computer Languages, vol. 6, no. 1, pp. 47–57, 1981.
- [6] “GNU Compiler Collection (GCC) Internals,” Free Software Foundation. [Online]. Available: <https://gcc.gnu.org/onlinedocs/gccint/>
- [7] “Options That Control Optimization,” in Using the GNU Compiler Collection (GCC), Free Software Foundation. [Online]. Available: <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>
- [8] X. Yang, Y. Chen, E. Eide, and J. Regehr, “Finding and understanding bugs in C compilers,” in Proc. 32nd ACM SIGPLAN Conf. Programming Language Design and

Implementation (PLDI), San Jose, CA, USA, Jun. 2011, pp. 283– 294.

- [9] X. Leroy, “ Formal verification of a realistic compiler,” Commun. ACM, vol. 52, no. 7, pp. 107– 115, Jul. 2009.
- [10] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, “ A survey on compiler autotuning using machine learning,” ACM Comput. Surv., vol. 51, no. 5, Art. no. 96, 2018.