

1 Multi-Language Benchmark Generation via L-Systems

2 VINICIUS FRANCISCO DA SILVA, UFMG, Brazil

3 HEITOR LEITE, UFMG, Brazil

4 FERNANDO MAGNO QUINTÃO PEREIRA, UFMG, Brazil

5 L-systems are a mathematical formalism proposed by biologist Aristid Lindenmayer with the aim of simulating
6 organic structures such as trees, snowflakes, flowers, and other branching phenomena. They are implemented
7 as a formal language that defines how patterns can be iteratively rewritten. This paper describes how such
8 a formalism can be used to create artificial programs written in programming languages such as C, C++,
9 Julia and Go. These programs, being large and complex, can be used to test the performance of compilers,
10 operating systems, and computer architectures. This paper demonstrates the usefulness of these benchmarks
11 through multiple case studies. These case studies include a comparison between clang and gcc; a comparison
12 between C, C++, Julia and Go; a study of the historical evolution of gcc in terms of code quality; a look into
13 the effects of profile guided optimizations in gcc; an analysis of the asymptotic behavior of the different phases
14 of clang's compilation pipeline; and a comparison between the many data structures available in the Gnome
15 Library (GLIB). These case studies demonstrate the benefits of the L-System approach to create benchmarks,
16 when compared with fuzzers such as CSMITH, which were designed to uncover bugs in compilers, rather than
17 evaluating their performance.

18
19 CCS Concepts: • Software and its engineering → Compilers.

20 Additional Key Words and Phrases: L-System, Benchmark, Synthesis

21 **ACM Reference Format:**

22 Vinicius Francisco da Silva, Heitor Leite, and Fernando Magno Quintão Pereira. 2025. Multi-Language Bench-
23 mark Generation via L-Systems . 1, 1 (October 2025), 25 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

25 1 Introduction

26 Language processing systems, such as compilers, interpreters and static analyzers are complex
27 tools whose validation depends on programs written in the target language. However, the number
28 of available benchmarks for any given compiler is typically limited [39]. To address this limitation,
29 several tools have been developed to automatically generate test programs [41]. This process,
30 known as *fuzzing* [26], is widely used to uncover bugs such as crashes and memory leaks. In the
31 compiler domain, fuzzers such as Csmith [41] and YARPGen [25] generate random C programs
32 to stress-test compilers and support static analysis. More recently, fuzzers like Fuzz4All [40] have
33 leveraged Large Language Models (LLMs) to generate test programs not only for compilers but
34 also for constraint solvers, interpreters, and other software systems with accessible APIs.
35

36 Despite the abundance of program generators [41], existing tools exhibit important limitations.
37 A key issue is the lack of control over the size of the generated programs. For example, Csmith,
38 the most widely used C compiler fuzzer, does not allow users to tune the output size. Instead, it

39 Authors' Contact Information: Vinicius Francisco da Silva, UFMG, Minas Gerais, Brazil, silva.vinicius@dcc.ufmg.br; Heitor
40 Leite, UFMG, Minas Gerais, Brazil, heitor.leite@dcc.ufmg.br; Fernando Magno Quintão Pereira, UFMG, Minas Gerais, Brazil,
41 fernando@dcc.ufmg.br.

42 Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee
43 provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the
44 full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored.
45 Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires
46 prior specific permission and/or a fee. Request permissions from permissions@acm.org.

47 © 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

48 ACM XXXX-XXXX/2025/10-ART

49 <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

50 produces programs whose sizes follow a normal distribution. When compiled with clang v9.0.1 at
 51 -O0, these programs contain on average 20,190 LLVM instructions, with a standard deviation of 3,650
 52 and a median of 19,161 instructions [12]. Other fuzzers, such as YARPGen [25], LDRGen [2], and
 53 Orange3 [23], show similar behavior. As a consequence, these tools are ill-suited for performance
 54 evaluation of compiler components such as parsing, semantic analysis, and code generation.

55 *Programs via L-Systems.* To overcome this limitation, this paper proposes a methodology for
 56 stress-testing compilers with a focus on *performance* rather than solely on *correctness*. Our approach
 57 enables the controlled generation of programs whose size can be precisely tuned by the user. This
 58 makes it possible to construct synthetic code of virtually arbitrary size, constrained only by practical
 59 resources such as generation time and storage space.

60 The central insight is that programs often exhibit recursive, self-similar structures, as discussed
 61 in Section 2. For instance, the branches of a control construct (e.g., if-then-else) are themselves
 62 programs that may recursively contain further control constructs. To exploit this property, we intro-
 63 duce a generation technique based on *L-systems* [24]. Originally developed by Aristid Lindenmayer
 64 to model the growth of biological organisms, L-systems provide a grammar-based framework that
 65 we extend to code generation. This paper shows how families of self-similar programs can be
 66 encoded as L-grammars. Programs are then generated by iterative rewriting, where each resulting
 67 string encodes the blueprint of a program organized around a central data structure such as an
 68 array or a list. As discussed in Section 3, this technique supports multiple programming languages,
 69 yields complex control-flow graphs, avoids undefined behavior, and enables the manipulation of
 70 different data structures.

71 *Summary of Contributions.* To demonstrate the benefits of representing program growth with
 72 L-systems, we introduce BENCHGEN, a multi-language benchmark generator. Section 4 evaluates
 73 BENCHGEN through seven case studies, including: a comparison between gcc and clang; a com-
 74 parison of Go, Julia, C, and C++; an asymptotic analysis of different components of clang; a
 75 longitudinal analysis of the evolution of gcc; an evaluation of profile-guided optimizations in
 76 clang; a comparison of data structures from GLIB; and a comparison between programs generated
 77 by BENCHGEN and by CSMITH. BENCHGEN is publicly available under the GPL 3.0 license, and has
 78 already been used to generate benchmarks for several languages beyond those discussed in this
 79 paper, including RUST and VALE.

82 2 L-Systems

83 An L-system (or *Lindenmayer system*) is a formal model based on rewriting rules, originally devised
 84 to describe the growth patterns of plants and other fractal-like structures. It comprises an alphabet
 85 of symbols, a set of production rules that define how symbols are transformed, and an initial *string*
 86 (the axiom) that serves as the starting point. At each iteration, the rules are recursively applied
 87 to the current string, producing increasingly complex sequences. Example 2.1 illustrates how this
 88 formalism operates.

89 *Example 2.1.* Figure 1 presents an example of an L-system. The rules used in this system generate
 90 geometric patterns through string rewriting. Starting from the axiom A, the productions specify
 91 how symbols evolve at each step: $A \rightarrow B - A - B$ and $B \rightarrow A + B + A$. Here, the symbols –
 92 and + represent rotations of 60 and 300 degrees, respectively. Repeated application of these rules
 93 generates sequences that, when interpreted graphically, produce intricate fractal curves, such as
 94 the well-known Sierpinski Triangle.

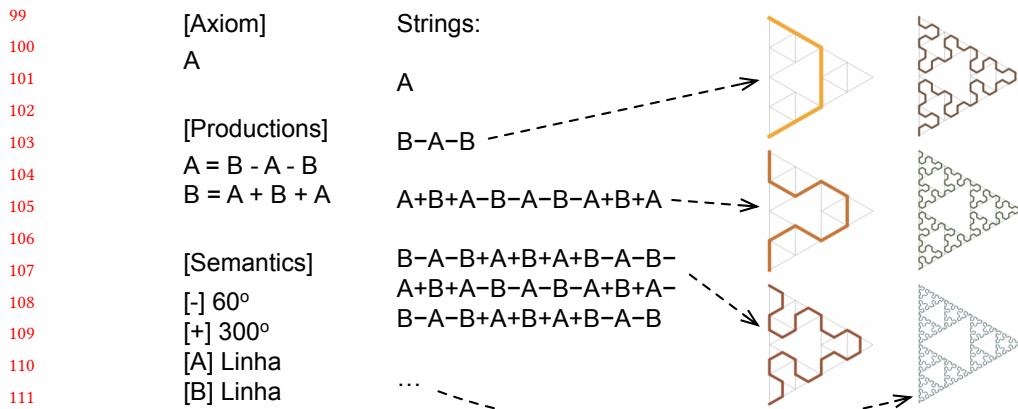


Fig. 1. L-system describing the Sierpinski Triangle.

2.1 Programs as Self-Similar Structures

L-systems exhibit a property known as *self-similarity*, meaning that structures contain smaller copies of themselves across different scales. In the context of L-systems, this feature arises naturally from the recursive application of rewriting rules, producing patterns that preserve the same shape at progressively finer levels of detail. This behavior is evident in fractals like the Sierpinski Triangle seen in Example 2.1, where each component is a scaled-down replica of the whole. Such hierarchical repetition is key to modeling phenomena like plant growth, coastlines, and tree branching.

Self-similarity also emerges in computer programs, which often embody recursive and hierarchical structures. Many programs are composed of smaller functions that may invoke themselves or be embedded within one another, as in if-then-else blocks or loops. Modularity and code reuse further reinforce this pattern: generic routines can be instantiated repeatedly across different abstraction levels, as Example 2.2 explains.

Example 2.2. Figure 2 illustrates the concept of self-similarity in code using a nested if-then-else block. Initially, a function $g(x)$ contains a single conditional. However, it can be recursively expanded to $g(x) = \text{if } g(x) \text{ then } g(x) \text{ else } g(x)$, forming a self-referential structure. Such recursive definitions naturally lead to self-similarity and are common in syntactical constructs that encode control-flow in programs.

Summary of Ideas This paper leverages the principle of self-similarity to generate C programs that are both well-defined and arbitrarily complex. The generation model is based on an L-grammar, akin to the one illustrated in Example 2.1, but instead of producing geometric patterns, it synthesizes C code constructions with executable semantics. Figure 3 shows an example of two versions of a program that BENCHGEN produces for C and Julia. This file is part of a 52-file benchmark that we use in Section 4.2 to compare the performance of different language processing systems.

3 Code Generation via L-Systems

The tool BENCHGEN, developed in this work, generates programs that manipulate data structures, according to the schema seen in Figure 4. Section 3.1 introduces the core building blocks used in program construction, while Section 3.2 describes semantics of these building blocks.

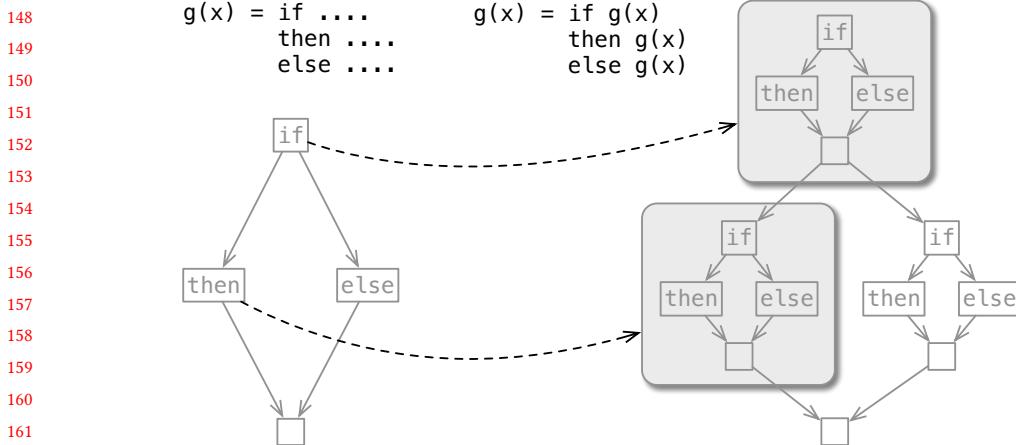


Fig. 2. The self-similar nature of computer code.

3.1 Syntactical Building Blocks

The L-systems described in this paper are built from two families of constructs:

- **Structure:** elements that define the control flow of a program, including IF, LOOP, and CALL.
- **Behavior:** operations that specify how data is manipulated, including new, insert, remove, and contains.

3.1.1 Structure Blocks. The control flow in programs generated by BENCHGEN arises from combining four types of code blocks, as specified by the grammar below:

```

175      b ::= IF(bcond, bthen)      ;; if_then
176      | IF(bcond, bthen, belse)  ;; if_then_else
177      | LOOP(bcond, bbody)       ;; while
178      | CALL(b)                   ;; function_call
179

```

Each of these constructs corresponds to a familiar programming construct: conditional branches (if-then, if-then-else), loops (while), and function calls. Example 3.1 provides an illustration.

Example 3.1. Figure 5 shows a grammar designed to synthesize programs. The right-hand side illustrates two derivation steps from this L-grammar, along with a corresponding (simplified) C program produced from the second derivation.

3.1.2 Behavior Blocks. The dynamic behavior of BENCHGEN programs emerges from interactions with data structures. All data structures within a program must share the same type. Currently, BENCHGEN supports arrays of integers and sorted linked lists of integers. Additionally, the C code generator also supports any of the twelve data structures available in the GNOME Library. Adding support for new containers involves extending four C++ classes. These classes define the behavior of four constructs in the L-grammar:

- **new:** Creates and initializes a data structure.
- **insert:** Adds an element to a data structure in scope.
- **remove:** Deletes an element from a data structure in scope.
- **contains:** Checks whether an element exists in a data structure in scope.

197	A function that is part of a C benchmark (52 .c files)
198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226	<pre>#include "program_c.h" array_t* func42 (array_t_param* vars, const unsigned long PATH0, int loopsFactor) { size_t pCounter = vars->size; if(PATH0 & 1) { <-- Control flow (Section 3.2) array_t_param params0; params0.size = 0; params0.data = (array_t**)malloc(params0.size*sizeof(array_t*)); array_t* array29 = func43(&params0, loopsFactor); DEBUG_RETURN(array29->id); free(params0.data); array29->refC--; if(array29->refC == 0) { free(array29->data); <-- Function call (Section 3.3) free(array29); DEBUG_FREE(array29->id); } } else { array_t_param params0; params0.size = 0; params0.data = (array_t**)malloc(params0.size*sizeof(array_t*)); array_t* array31 = func44(&params0, get_path(), loopsFactor); DEBUG_RETURN(array31->id); free(params0.data); array31->refC--; if(array31->refC == 0) { free(array31->data); free(array31); DEBUG_FREE(array31->id); } } array_t_param params0; params0.size = 0; params0.data = (array_t**)malloc(params0.size*sizeof(array_t*)); array_t* array35 = func45(&params0, loopsFactor); DEBUG_RETURN(array35->id); free(params0.data); return array35; }</pre>
	A function that is part of a similar Julia benchmark
	<pre>include("program_julia_head.jl") function func42 (vars::JArray_param, PATH0::UInt64, loopsFactor::Int)::JArray pCounter = length(vars.data) if (PATH0 & 1) != 0 params0 = JArray_param{Int}() <-- Structures & behavior (Section 3.1) params0.size = 0 array29 = func43(params0, loopsFactor); array29.refC = 1 if array29.refC == 0 empty!(array29.data) array29 = nothing end else params0 = JArray_param{Int}() params0.size = 0 array31 = func44(params0, get_path(), loopsFactor); array31.refC = 1 if array31.refC == 0 empty!(array31.data) array31 = nothing end end params0 = JArray_param{Int}() params0.size = 0 array35 = func45(params0, loopsFactor); return array35; end</pre>
	Multi-Language support (Section 3.5)

Fig. 3. Example of code that BENCHGEN produces for C and Julia that manipulate arrays.

Notice that these four behavior blocks can be customized by BENCHGEN users in any way. For instance, BENCHGEN provides a “*scalar mode*” to generate programs that manipulate integer variables. In this case, we have configured `insert` to increment variables; `remove` to decrement them; `new` to declare and initialize them with zero; and `contains` to test if they are zero. As a more concrete example, below we show how three such operations may be defined for programs working with arrays.

Example 3.2. Figure 6 shows C code generated to manipulate arrays. Although a program may handle many arrays, the example focuses on a specific variable, `array156`. The operations `new()`, `insert()`, and `contains()` must be customized by the user to define the behavior of the generated code.

3.2 Execution Flow

Programs generated by BENCHGEN are executable. Their control flow is governed by a variable named PATH, of type `unsigned long`, which determines the outcome of conditional branches. Algorithm BitPath in Figure 1 correlates the outcome of branches with this PATH variable. The

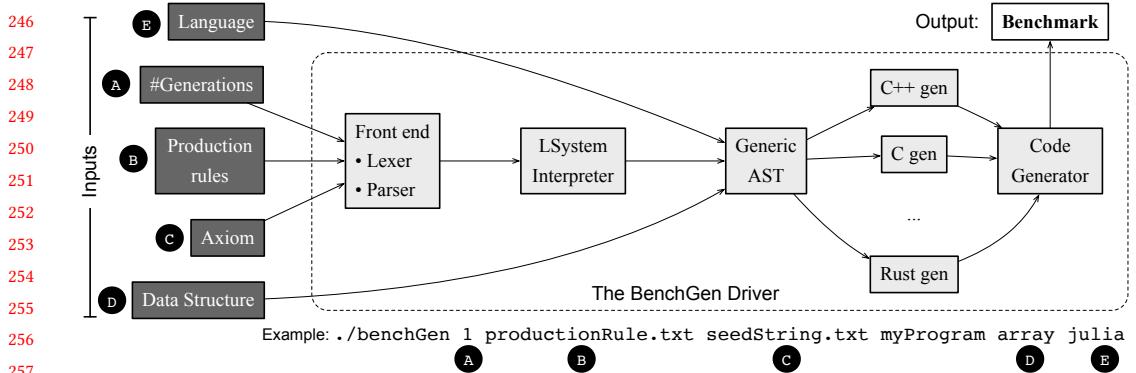


Fig. 4. BENCHGEN is a tool that generates programs in different programming languages. It receives as input the description of an L-System, which consists of a set of production rules, plus a starting symbol (the axiom). It interprets these rules, producing strings (the L-Strings), which guide the construction of generic ASTs. These trees can be converted to programs in different programming languages using different containers. A code generator then converts the language-specific tree to a program.

```

264 [Productions]
265 A = insert IF(B, LOOP(A)) remove
266 B = search IF(CALL(A), _)
267 [Axioma]
268 A
269
270 void f(struct S s, ulong path){
271     insert(3, s); <-
272     if (path & 1) { <-
273         B(s);
274     } else {
275         for (int i0 = 0; i0 < 5; i0++) {
276             A(s);
277         }
278     }
279     remove(6, s); <-
  
```

Iteration 1:
insert IF(B, LOOP(A)) remove

Iteration 3:
insert IF(search IF(CALL(A), _),
LOOP(insert IF(B, LOOP(A)) remove))
remove

Fig. 5. Example of an L-grammar used to define programs. The figure shows a simplified version of a C program. Figure 3 provides a more concrete view.

algorithm use a counter stack to ensure each path is uniquely identifiable via the PATH variable. It assigns a unique bitmask to each branch condition by:

- (1) **Tracking Nesting Depth:** Using a stack to manage counters for each level of nested branches.
- (2) **Resolving Joins:** Propagating the "maximum counter" upward when branches merge, ensuring no bitmask collisions.

The way Algorithm 1 determines how control-flow is taken lets us use BENCHGEN to test the effectiveness of profile-guided optimizations, as Section 4.5 will demonstrate. This mechanism is clarified in Example 3.3.

```

295     new
296         array_t* array156;
297         array156 = (array_t*)malloc(sizeof(array_t));
298         array156->size = 42;
299         array156->refC = 1;
300         array156->id = 156;
301         array156->data = (unsigned int*)malloc(array156->size*sizeof(unsigned int));
302         memset(array156->data, 0, array156->size*sizeof(unsigned int));
303         DEBUG_NEW(array156->id);
304
305     insert
306         unsigned int loop46 = 0;
307         unsigned int loopLimit46 = (rand()%loopsFactor)/2 + 1;
308         for(; loop46 < loopLimit46; loop46++) {
309             for (int i = 0; i < array156->size; i++) {
310                 array156->data[i]--;
311             }
312         }
313
314     contains
315         unsigned int loop46 = 0;
316         unsigned int loopLimit46 = (rand()%loopsFactor)/2 + 1;
317         for (int i = 0; i < array156->size; i++) {
318             if (array156->data[i] == 61) {
319                 break;
320             }
321         }
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343

```

Fig. 6. Examples of block definitions for new, insert, and contains.

Algorithm 1: BitPath assignment of unique bitmasks to branch conditions

Input: A function f **Output:** A mapping of each branch condition in f to a unique bitmask**Initialize the Stack:** push a single counter $Counter_0 = 1$, representing the least significant bit (LSB);**Assign Masks to Branches:;****Same Nesting Level:** use the current top-of-stack counter to generate the bitmask, e.g.,
PATH & ($1 \ll (\text{counter} - 1)$);**Nested Branch:** push a new counter, incremented by 1 from the parent counter,
ensuring fresh bits for deeper nesting;**Handle Join Points (after if-then-else):;**

Pop the current counter from the stack;

Update the parent counter (now on top of the stack) to the maximum of:;

- (a) the parent's original value;
- (b) the popped (child) counter;

This ensures that parent branches account for the deepest nesting level beneath them;

Independent Branches: after resolving a join, subsequent branches at the same level reuse
the updated parent counter;

Example 3.3. Figure 7 depicts the control flow graph of a program with the PATH variable
combined with masks created by the Algorithm 1. Notice that the counter used as the bitmask

is updated due to nesting or sequencing. Nesting causes the increment of the counter on the top of the counter stack. Whenever we analyze a branch, its counter will be the maximum of all the possible control-flows that reach it.

347

```

348     Code pattern:
349     if (PATH & 1) {
350         if (PATH & 2) {
351             a;
352         } else {
353             b;
354         }
355     } else {
356         if (PATH & 2) {
357             if (PATH & 4) {
358                 c;
359             } else {
360                 d;
361             }
362         } else {
363             if (PATH & 4) {
364                 e;
365             } else {
366                 f;
367             }
368         }
369     if (PATH & 8) {
370         g;
371     } else {
372         h;
373     }
374
375 }
```

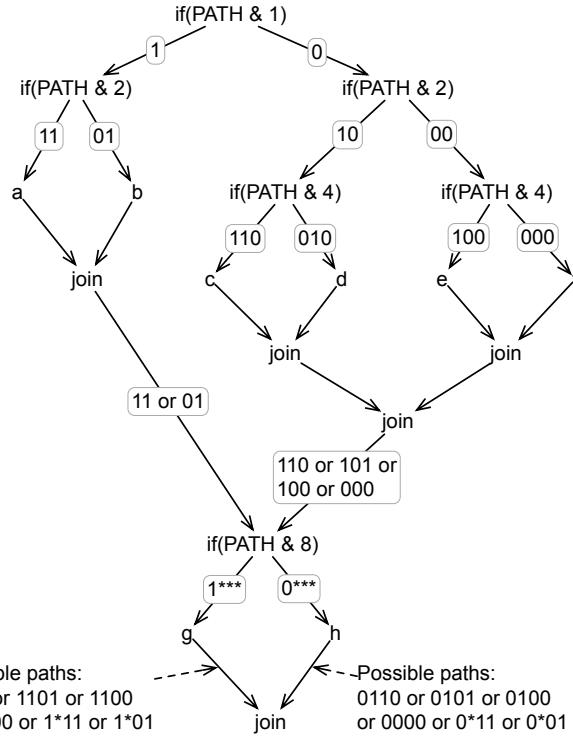


Fig. 7. The control flow of a synthetic program is determined by the path parameter.

372

3.3 Function Calls

BENCHGEN supports function calls through the CALL clause. When an L-string includes a construction of the form CALL(*e*), the entire substring *e* is extracted and defined as a separate function. The functions generated from a given L-specification can be either grouped into a single file or distributed across multiple files, depending on a configuration parameter in BENCHGEN. Example 3.4 illustrates how BENCHGEN generates interprocedural code.

Example 3.4. Figure 8 depicts the control flow produced by a CALL block. The enclosed string gives rise to a new function, which becomes part of the synthesized program. This new function is invoked at the point in the L-string where the CALL clause appears.

385

All occurrences of CALL(*b*) with identical strings *b* result in calls to the same function. To avoid redundancy, BENCHGEN maintains a table mapping strings to functions, ensuring that identical strings refer to the same function instance. Example 3.5 shows how the same function can be called multiple times. Notice that BENCHGEN does not support the creation of recursive function calls. If a function *f*₀ calls another function *f*₁, then *f*₀ is produced by a string that is strictly larger than the string that generates *f*₁.

392

```

393 insert
394 IF(
395   search
396   IF(
397     CALL(
398       insert
399         IF(
400           B,
401           LOOP(A)
402         )
403       remove
404     ),
405   )
406   remove
407   ),
408   —
409   ),
410   LOOP( ... )
411   )
412   remove
413
414
415
416
417
418

```

Fig. 8. Control flow of a program containing a CALL block.

Example 3.5. Figure 9 shows an L-System that generates a program with two functions, f_0 and f_1 . The latter is called twice, because it is produced by a string that appears two times in a given iteration of the expansion process.

Parameter Passing. Functions generated by BENCHGEN receive two parameters:

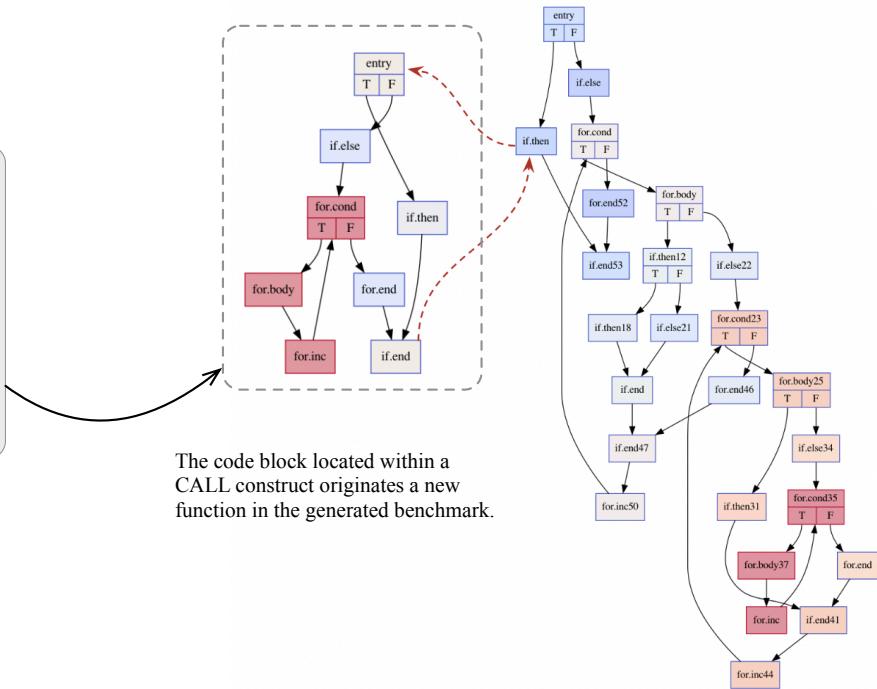
- **Data:** an array of data structures that enables sharing between caller and callee functions.
- **Path:** a control variable that governs execution flow, as described in Example 3.3.

The **Data** array is populated using a reaching definitions analysis, which determines which variables in the caller function are available to be passed as arguments at the call site. Example 3.6 illustrates this mechanism.

Example 3.6. Figure 10 illustrates parameter passing in a program generated by BENCHGEN. At the function call site, a reaching definitions analysis identifies three available variables: `array1`, `array156`, and `array157`. Pointers to these variables are inserted into the `param.data` structure, which is passed to function `func0`. Inside the callee, the passed variables are copied into new ones using the new construct. Each variable is copied exactly once. If no additional parameters are available for copying, subsequent new operations will create fresh variables, as seen in Example 3.3.

3.4 Memory Management

Programs generated by BENCHGEN do not suffer from memory leaks, despite frequently relying on heap allocation. To prevent leaks, BENCHGEN employs a reference-counting garbage collector. This approach tracks how many references (pointers) exist to each dynamically allocated object. When



```
442 Seed string:  
443 A  
444 Iteration 1:  
445 IF(CALL(C), CALL(LOOP(B)))  
446  
447 Iteration 2:  
448 IF(CALL(LOOP(insert)), CALL(LOOP(insert)))  
449  
450 void f0(array_t_param* s, ... path) {  
451     if (path & 1){  
452         array_t_param params1;  
453         // initialize the parameter  
454         f1(&param1, path);  
455     } else {  
456         array_t_param params2;  
457         // initialize the parameter  
458         f1(&param2, path);  
459     }  
460 }  
461  
462 Grammar:  
463 A = IF(CALL(C), CALL(LOOP(B)))  
464 B = insert  
465 C = LOOP(insert)  
466  
467 void f1(array_t_param* vars, ... path) {  
468     for (int i0 = 0; i0 < path & 0xFF; i0++) {  
469         // ... insert into array ...  
470     }  
471 }
```

Fig. 9. An L-System that would generate multiple calls to the same function.

461 Code of caller function, which implements the **CALL** construct

```
462     array_t_params1;
463     params1.size = 3;
464     params1.data = (array_t**)
465         malloc(params1.size*sizeof(array_t*));
466     params1.data[0] = array0;  ||
467     params1.data[1] = array156; || <-----|
468     params1.data[2] = array157; || |
469     array_t* array158 = func6(&params1, path);
```

The analysis of reaching definitions determines that three variables are available at the point of call. These variables are passed as parameters to the called function.

469 Code of callee function, which implements the **new** construct

```
470     array_t* func6(array_t_param* vars, const unsigned long PATH0) {  
471         size_t pCounter = vars->size;  
472         array_t* array1;  
473         if (pCounter > 0) { ←  
474             array1 = vars->data[--pCounter];  
475             array1->refC++;  
476         } else { ←  
477             array1 = (array_t*)malloc(sizeof(array_t));  
478             array1->size = 386;  
479             array1->refC = 1;  
480             array1->id = 1;  
481             array1->data = (unsigned int*)malloc(array1->size*sizeof(unsigned int));  
482             memset(array1->data, 0, array1->size*sizeof(unsigned int));  
483         } ... }  
484     }
```

If there are still paths available, the NEW copies one of these to the new variable.

Otherwise, a new variable is created, as seen.

-- If there are still parameters available, the NEW construct copies one of these parameters to the new variable.

Otherwise, a new data structure is created, as seen in Figure 4.

Fig. 10. Parameter passing in programs created by BENCHGEN.

484 a reference is created, the count is incremented; when it is removed, the count is decremented.
485 Once the count reaches zero, the object is no longer reachable and can be safely deallocated. To
486 support this mechanism, each structure created by BENCHGEN includes an additional field, `refC`,
487 which stores the current reference count. Example 3.7 shows how reference counting is used in
488 BENCHGEN to prevent memory leaks from happening.

491 *Example 3.7.* Figure 11 illustrates how BENCHGEN uses reference counting to manage memory.
 492 In this example, two variables, *x* and *y*, initially point to two separate heap-allocated structures.
 493 Each of these structures contains a *refC* field that holds the number of active references to the
 494 object. When the assignment *x* := *y* occurs, the reference count of the structure originally pointed
 495 to by *x* is decremented, as *x* no longer refers to it. If this decrement causes *refC* to reach zero, the
 496 structure is automatically deallocated. Meanwhile, the reference count of the structure pointed to
 497 by *y* is incremented to account for the new reference from *x*. After the assignment, both *x* and *y*
 498 point to the same object, whose *refC* now reflects two active references. This mechanism ensures
 499 that heap-allocated memory is reclaimed as soon as it is no longer reachable, preventing memory
 500 leaks in programs generated by BENCHGEN.
 501

502

503 **Definition of the Array Data Structure**

```
504
505     typedef struct {
506         unsigned int* data;
507         size_t size;
508         size_t refC; <----- Data structures created via LGen
509         int id;
510     } array_t;
```

511 **Variable assignment, e.g., as due to parameter passing**

```
512     array_t* arr1 = arr0;
513     arr0->refC++; <----- The new clause can cause a variable
514
515     { ... <----- assignment if there are parameters available for
516       arr0->refC--;
517       if (!arr0->refC) <----- assignment (see example 3.5).
518         free(arr0->data);
519         free(arr0);
520     }
```

521 **Variable definition leaves scope**

```
522
523     { ...
524       arr0->refC--;
525       if (!arr0->refC) <----- A variable goes out of scope when the block in
526         free(arr0->data); which it is defined ends. In this case, the
527         free(arr0); counter associated with that variable is
528     }
```

Fig. 11. Reference counter implementation.

525 3.5 Multilanguage Support

526 BenchGen is currently available in a beta version that supports the creation of benchmarks for
 527 multiple programming languages. Some of these languages are evaluated in Section 4.2. The
 528 benchmark generator itself is implemented in C++. To add support for a new language, users
 529 must implement new C++ classes that define the semantics of the structure and behavior blocks
 530 described in Section 3.1. This process also requires modifying the templates that generate code for
 531 the structure blocks (IF, LOOP, and CALL) and the behavior blocks (insert, remove, contains, and
 532 new).

533 Extending BENCHGEN therefore involves recompilation and is not yet fully automated. In addition
 534 to implementing new C++ classes, users must register the language module in the main BENCHGEN
 535 driver, which requires changing one line of code in the driver itself. In future work, we aim to fully
 536 automate this process and decouple the tool from the specific languages it supports. To this end,
 537 we are considering extending BENCHGEN with a domain-specific language that would allow users
 538 to specify new code-generation directives declaratively.
 539

540 4 Experimental Evaluation

541 The goal of this section is to demonstrate how BENCHGEN can be used in practice. To this end, we
 542 explore its usage in the following *Case Studies*:

- 543 • **CS1:** Comparison between gcc and clang in terms of execution speed, binary size, and
 compilation time.
- 544 • **CS2:** Comparison of the C and C++ compiler backends available in the GNU Compiler
 Collection.
- 545 • **CS3:** Analysis of the asymptotic behavior of different optimization levels in clang and gcc.
- 546 • **CS4:** Examination of the evolution of gcc from version 5 to 14, focusing on execution speed,
 binary size, and compilation time.
- 547 • **CS5:** Evaluation of the impact of profile-guided optimizations in clang.
- 548 • **CS6:** Comparison of different data structures in the GNOME LIBRARY (GLIB) with respect to
 insertion, search, and deletion times.
- 549 • **CS7:** Comparison between BENCHGEN and CSMITH, a compiler fuzzer.

550 **Experimental Setup:** All experiments were conducted on an Intel(R) Xeon(R) CPU E5-2680
 551 v2 running at 2.80 GHz, with Linux Ubuntu 5.15.0-139-generic. The specific compiler and library
 552 versions used are detailed in each case study.

553 **Checking for Undefined Behavior:** We analyzed the C programs evaluated in Section 4.1
 554 using four tools capable of detecting undefined behavior: UBSAN and ASAN from clang [36],
 555 Valgrind [31], the EVA plugin of Frama-C [4], and KCC [13]. In addition, we verified that every
 556 program evaluated in Section 4.2, when executed in debug mode, produces the same trace¹. While
 557 these measures do not formally prove that BENCHGEN always generates sound benchmarks, they
 558 provide strong evidence that this is highly probable.

559 4.1 CS1: gcc vs clang

560 The GNU C Compiler (gcc) and LLVM’s clang are the two most widely used C compilers, and
 561 comparisons between them are common in the literature. This case study contributes one more data
 562 point to this ongoing discussion by evaluating both compilers across three performance metrics:

- 563 • Execution time of the compiled program.
- 564 • Compilation time required to build the program.
- 565 • Binary size of the compiled program.

566 The comparison covers six optimization levels for gcc 14.2: -O0, -O1, -O2, -O3, -Os, and -Ofast;
 567 and seven for clang 21.0: -O0, -O1, -O2, -O3, -Oz, -Os, and -Ofast. Binary sizes were collected
 568 using the Linux `size` command, considering only the text section, which represents the size of
 569 the executable code in bytes. Compilation and execution times were measured with `hyperfine`,
 570 configured with three warm-up runs and at least ten benchmark runs. To compare the compilers, we
 571 have used four different L-Systems to generate four programs. We chose generations that ensured
 572 a mix of different execution times; hence, this custom benchmark collection contains programs
 573 that run within 10, 50 and 200 seconds once compiled with gcc -O3.

574 *Discussion.* Figure 12 summarizes the results of this comparison, showing, under each plot, the
 575 L-System and the generation that produced those numbers. The scatter plots show compilation time
 576 against execution time, with point size proportional to binary size. We observe that gcc and clang
 577 tend to produce programs with similar execution times at the lowest and highest optimization
 578 levels; however, gcc’s compilation times are generally lower.

587 ¹BENCHGEN provides a debug mode in which each program logs every behavioral operation defined in Section 3.1.2.

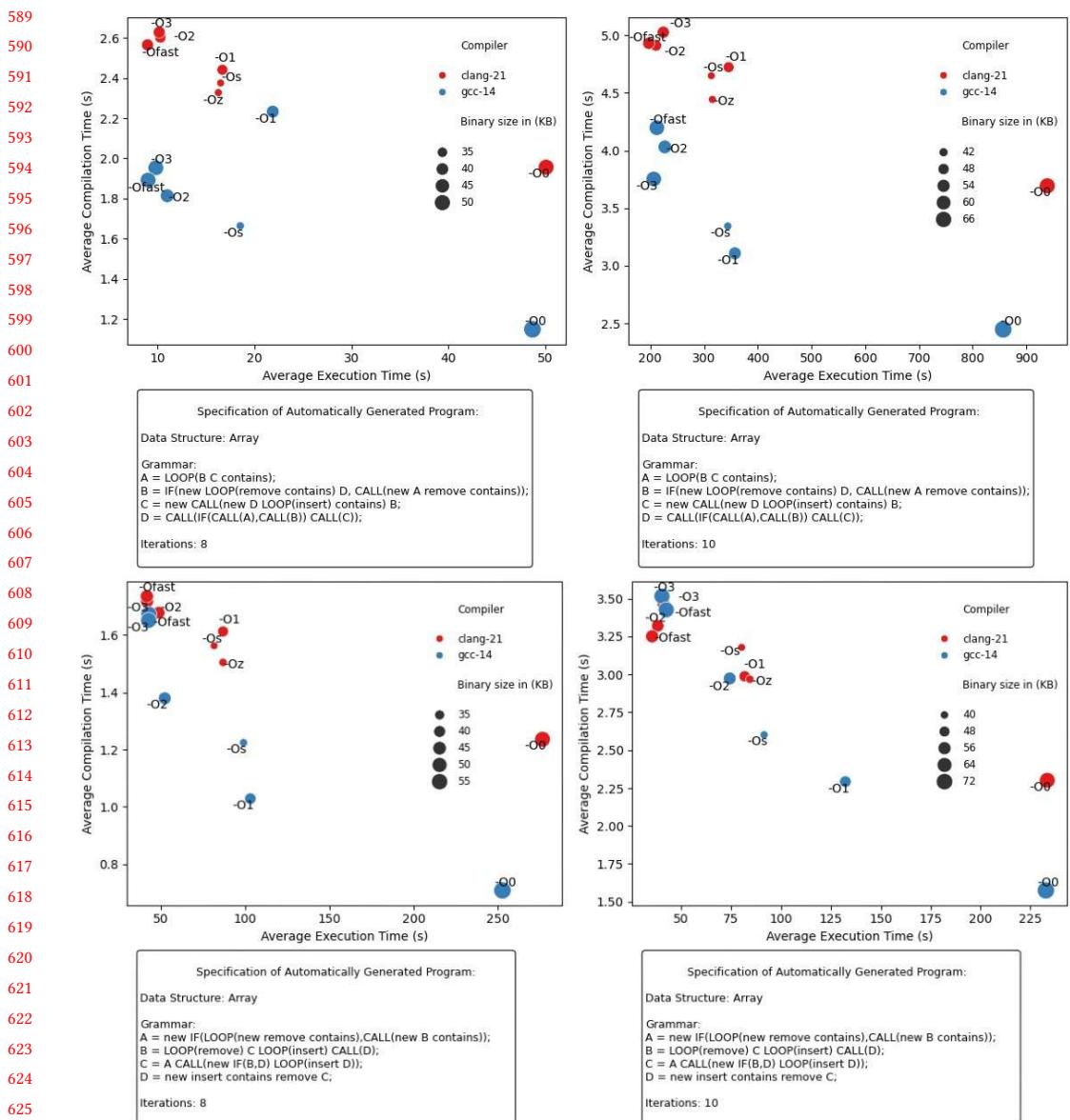


Fig. 12. Comparison of gcc 14.2 and clang 21.0 across optimization levels.

A clear trend emerges for gcc: it often exhibits a smooth trade-off between compilation time and execution time, forming a Pareto-like curve where longer compilation typically results in faster execution. This pattern is less evident for clang. Instead, clang shows a marked distinction between the lowest optimization levels (-O0, -O1) and the higher ones, while the most aggressive optimizations (-O2, -O3, and -Ofast) yield very similar results. This observation echoes findings by Curtsinger and Berger in their work on Coz [11], who reported no statistically significant difference

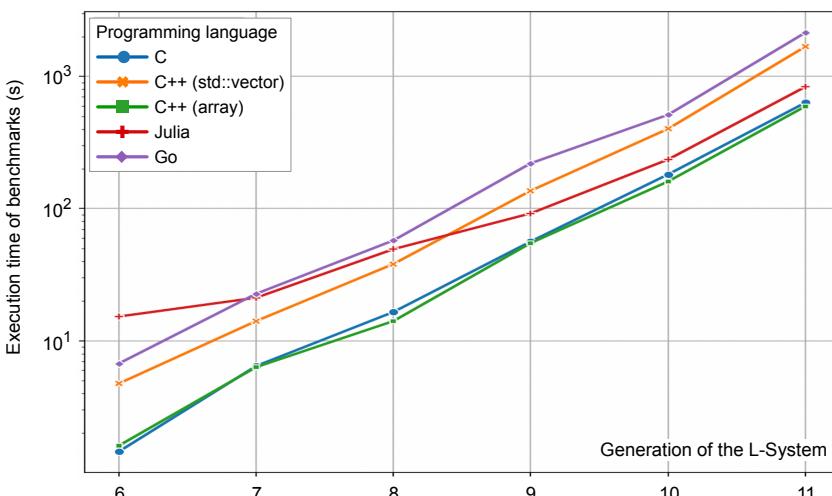
638 between -O2 and -O3 in clang². Regarding binary size, both compilers generate smaller executables
 639 at -Os, with clang holding a slight edge (2–3%) at the -Oz level. Nevertheless, both can reduce
 640 code size by nearly 50% when moving from -O3 to -Os.

641 642 CS2: Comparison between Different Programming Languages

643 As explained in Section 3.5, BENCHGEN can be customized to synthesize benchmarks in multiple
 644 programming languages. This section leverages its multi-language support to compare C, C++,
 645 Julia, and Go. Support for these languages is currently available in BENCHGEN’s official repository;
 646 however, the tool has also been extended to generate benchmarks in other languages, including Rust
 647 and Vale. These extensions, however, are not maintained by the BENCHGEN authors. To perform
 648 this comparison, we used the first grammar shown in Figure 12 to produce eleven generations of a
 649 program (reporting data for generations 6 through 11). Programs in each language were executed
 650 with the following tools:

651 C: gcc 13.3.0
 652 C++: g++ 13.3.0
 653 Go: go 1.25.0
 654 Julia: julia 1.11.6

655 Note that Julia runs in interpreted mode with just-in-time compilation, whereas the other languages
 656 are compiled ahead of time. This evaluation considers only execution time, excluding compilation
 657 time. Also, each language produces the same number of source files, except for C and C++, which
 658 generate one extra header file. For example, in the 11th generation, BENCHGEN produces 52 files
 659 for Julia and Go, and 53 for C and C++.
 660



677 Fig. 13. Execution time of benchmarks produced by BENCHGEN in different programming languages.
 678

680 *Discussion.* Figure 13 compares execution time across languages. The results largely align with
 681 expectations: they highlight the performance differences between low-level, statically compiled
 682 languages (C and C++) and higher-level languages with different compilation models (Go and Julia).
 683 All languages exhibit an exponential increase in execution time as program depth grows, which

684
 685 ²See <https://youtu.be/r-TLSBdHe1A?t=1438> at 23:58

is inherent to the fractal-based benchmark generator. Beyond this general trend, we note several specific findings:

- **C vs. C++ with arrays.** Performance is nearly identical: there is no statistically significant difference at the 95% confidence level when comparing programs of the same generation. This is because both are compiled with the same compiler infrastructure (gcc 13.3.0), which produces highly similar assembly code.
- **Julia.** Julia shows excellent performance, particularly for larger programs—a behavior already observed in previous work [6]. Its just-in-time (JIT) compiler, built on LLVM, can apply aggressive runtime optimizations such as loop unrolling and SIMD vectorization by specializing code based on runtime types and execution patterns. As program complexity increases in later generations, these optimizations give Julia a significant advantage, allowing it to outperform C++ with vectors.
- **C++ with vectors.** Programs using `std::vector` are slower than those using raw arrays. This overhead arises from additional metadata (size, capacity) and the cost of reallocation when capacity is exceeded, which requires copying elements to new memory. In benchmarks where data grows quickly, this reallocation cost is significant. Indeed, from the 8th to the 9th generation, Julia’s JIT optimizations allow it to surpass C++ with vectors in performance.
- **Go.** Go is consistently the slowest language. This is unsurprising, and follows trends already reported and discussed in open forums and technical books [37]. While Go is compiled, its runtime includes garbage collection and applies less aggressive optimizations, which introduces overhead not present in C, C++.

The trends observed in Figure 13 are consistent with those documented on the well-known “Benchmark Game” website³. A key distinction, however, lies in the implementation strategy. While the Benchmark Game compares hand-optimized solutions that may differ syntactically and semantically, BENCHGEN produces structurally similar code across languages. This methodology offers a more precise comparison of compiler performance by eliminating variability introduced by differing algorithmic implementations.

4.3 CS3: Asymptotic Behavior

The ability to gradually vary the size of programs generated by BENCHGEN makes it a suitable tool for conducting empirical asymptotic analyses of language processing systems. In this section, we illustrate this capability by empirically evaluating the asymptotic complexity of different phases of clang (front end, middle end, and back end), as well as the full compilation pipeline of gcc. For this purpose, we employ a standard L-System to produce eight generations of a program (from the 5th to the 12th) and feed these programs to the compilers.

Discussion. Figure 14 relates the running time of the different compilation phases to the size of the binary processed at each generation of the target program. This analysis yields several insights, discussed below:

- All analyzed tools exhibit linear behavior for large programs, with very strong correlations (above 0.9) across different compilers and optimization levels. This applies to both clang and gcc. However, for clang’s front end, linear growth only becomes evident in later generations (iterations 9–12), due to the heavy startup costs of the parser and IR generator.
- There is little statistical difference among the higher optimization levels of clang (-O2, -O3, and -Ofast). This observation corroborates the findings reported in Section 4.1. It appears

³Available at <https://benchmarksgame-team.pages.debian.net/benchmarksgame/> as of September 20th.

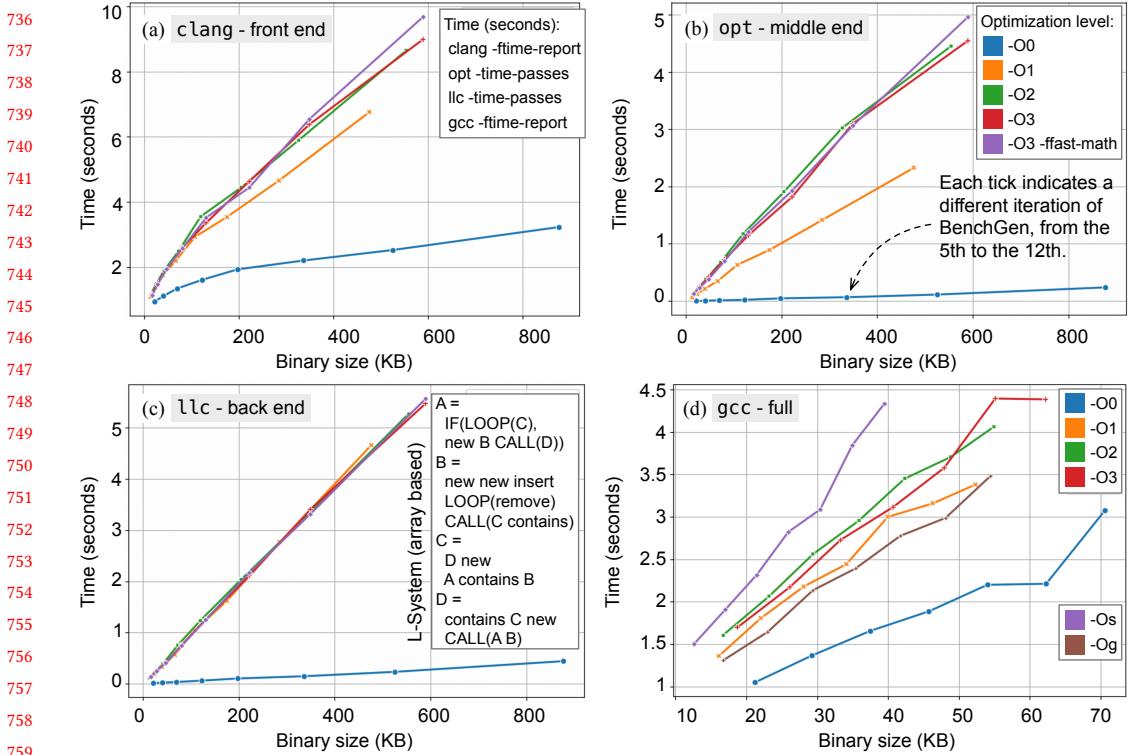


Fig. 14. Empirical evaluation of the asymptotic behavior of different compilation phases of clang and of the full compilation pipeline of gcc.

valid, at least in this experiment, across all three phases of clang's compilation pipeline, and is especially pronounced in llc, the machine code generator.

- Although gcc also shows strong linear behavior at all optimization levels, the constant factors vary considerably. The size-optimization level (-Os) exhibits the steepest growth, which is expected since gcc -Os produces the smallest binaries.

Overall, these results suggest that, at least when compiling BENCHGEN programs, neither clang nor gcc exhibit asymptotic performance bugs of the kind reported in previous work [18, 32].

4.4 CS4: The Evolution of gcc

GNU gcc, currently at version 14, is one of the oldest and most widely used C compilers. Over the years, it has undergone many changes, including support for additional languages and continuous performance improvements. This case study compares different versions of gcc across multiple optimization levels using three metrics:

- Execution time of the compiled program (measured with `hyperfine`).
- Compilation time required to build the program (also measured with `hyperfine`).
- Binary size of the `.text` segment of the executable (collected with the Linux `size` utility).

For this comparison, we used BENCHGEN with the first L-System shown in Figure 12. The program generated at the 10th iteration of this grammar was compiled with six versions of gcc.

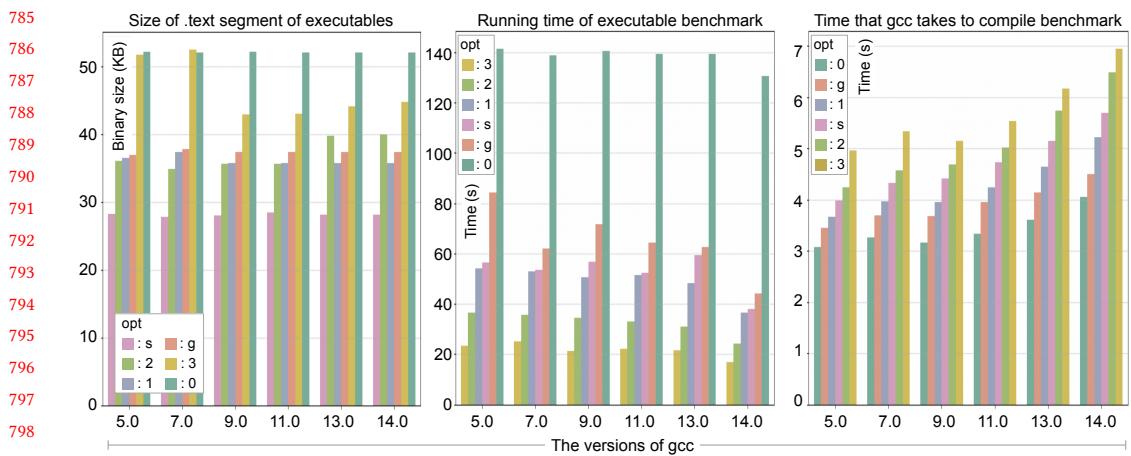


Fig. 15. Evolution of gcc from version 5 to 14 across six optimization levels. Results are shown for (left) binary size of the `.text` segment, (center) execution time of the benchmark, and (right) compilation time.

Discussion. Figure 15 summarizes results for six optimization levels: `-O0`, `-Og`, `-O1`, `-Os`, `-O2`, and `-O3`. Each group of bars corresponds to a compiler version: `gcc5`, `gcc7`, `gcc9`, `gcc11`, `gcc13`, and `gcc14`. Below we discuss some of the insights of this study:

Compilation time: We observe a steady increase in compilation time with each new version of gcc, particularly at higher optimization levels such as `-O3`. Comparing versions 5 and 14, compilation time rose by about 42% over nine years, with a 31% increase even at `-O0`. This is unsurprising: new optimization passes have been added over time, which increases compile time in exchange for better code quality.

Execution time: Performance has improved at the higher optimization levels. Comparing `gcc5` to `gcc14`, execution time decreased by roughly 27% at `-O3`, 32% at `-O1`, and as much as 47% at `-Og`. The gains at `-O0` were modest, about 7%. These results indicate that newer versions of gcc generate more efficient code, especially at intermediate optimization levels.

Binary size: Binary size has remained stable across versions. At `-Os`, code size decreased only 0.34% between `gcc5` and `gcc14`. At `-O3`, binary size initially increased from `gcc5` until `gcc7`, dropping noticeably in `gcc9`. Past this point, we again observe a small increase until `gcc14`. However, in this version code is still almost 10% smaller than in `gcc5`.

Over the evolution from version 5 to 14, gcc shows a clear trade-off: longer compilation times in exchange for more efficient executables. While binary size has changed little, execution performance has improved noticeably, particularly at `-Og` and `-O3`. These results highlight an important trend: the gcc community has consistently prioritized code quality and runtime efficiency over compilation speed, accepting longer build times in return for measurable performance gains.

4.5 CS5: Profile Guided Optimizations in clang

Profile-Guided Optimization (PGO) is a compilation technique that leverages runtime information collected from representative executions of a program to improve the quality of the generated code. By replacing purely static heuristics with empirical execution data, PGO enables the compiler to make more informed decisions regarding branch prediction, function inlining, loop unrolling, and code layout, thereby enhancing both performance and efficiency. In typical C, C++ and Rust compilers, PGO is integrated into the standard optimization pipeline rather than constituting a

separate optimization level. Specifically in clang, PGO serves as a refinement to existing optimization levels, most notably `-O2` and `-O3`. When profile data is provided via the `-fprofile-use` flag, clang applies the chosen baseline optimizations but augments them with profile-driven insights, generating code that more accurately reflects the observed runtime behavior of the program.

This section illustrates how BENCHGEN can be used to evaluate the effectiveness of profile-guided optimizations (PGO). As described in Section 3.2, the control flow of a program generated by BENCHGEN is governed by the variable PATH. To assess the performance gains provided by PGO, we conduct the following experiment:

- (1) Generate four program variants (Generations 6, 7, 8, and 9) using the following grammar with array structures:


```
A = new B B
B = IF(LOOP(insert A contains), LOOP(insert A contains))
```
- (2) Compile each program at the `-O2` optimization level with PGO enabled⁴.
- (3) During the “Training Phase,” collect profile information by running each program with $\text{PATH} = 2^0$ (i.e., $\text{PATH} = 1$).
- (4) During the “Testing Phase,” for $i = 1$ to 63:
 - (a) Execute each program with $\text{PATH} = 2^i - 1$, e.g., $\text{PATH} = 0b1$, $\text{PATH} = 0b11$, etc.

Let t denote the execution time of the program compiled with clang `-O2` without profile data, and let t_i denote the execution time with $\text{PATH} = 2^i$. Then, the ratio t/t_i quantifies the relative speedup due to profile-guided optimizations as the control-flow path diverges by i bits.

Discussion. Figure 16 shows the data collected during this experiment. The results in Figure 16 reveal two distinct behaviors of profile-guided optimizations. In the expected case (Generations 6–8), the performance trend matches our intuition: when the execution path during testing closely resembles the path exercised during training, we observe significant speedups, often reaching 2x. Similar results on real applications have been reported in previous work [8, 27, 34].

As the number of bit flips in PATH increases, however, the executed control flow diverges from the profiled run, and the benefit of PGO diminishes. Eventually, for highly divergent inputs, the profiled and non-profiled binaries converge to similar performance, as the compiler’s profile-driven decisions no longer apply.

Profile Overfitting: The unexpected case arises in Generation 9. Here, PGO yields a very large speedup when execution follows the training path (more than 2.2x). Yet, as the path diverges, the optimized binary becomes slower than its non-profiled counterpart. This behavior suggests an instance of profile overfitting: because PGO biases optimizations toward the observed hot path, code along unprofiled paths may suffer from unfavorable decisions.

A plausible explanation for this effect is the difference in the number of functions inlined with and without profiling data. Clang/LLVM makes inlining decisions at the call-site level. A function call along the hot path may be inlined aggressively, while the same call in a cold path may be left uninlined, leading to inconsistencies in call overhead, code size, and optimization opportunities. Combined with profile-driven code layout and branch prediction, this selective optimization can make non-profiled paths slower than in a neutral, non-PGO binary. Thus, Generation 9 highlights a limitation of PGO: while it can deliver substantial performance gains when training inputs are representative, it may degrade performance when profile data overfits to a narrow subset of execution paths.

⁴We report results for clang at the `-O2` level because the speedups achieved with PGO are comparable to those observed at `-O3`, and consistently higher than those obtained at `-O1`.

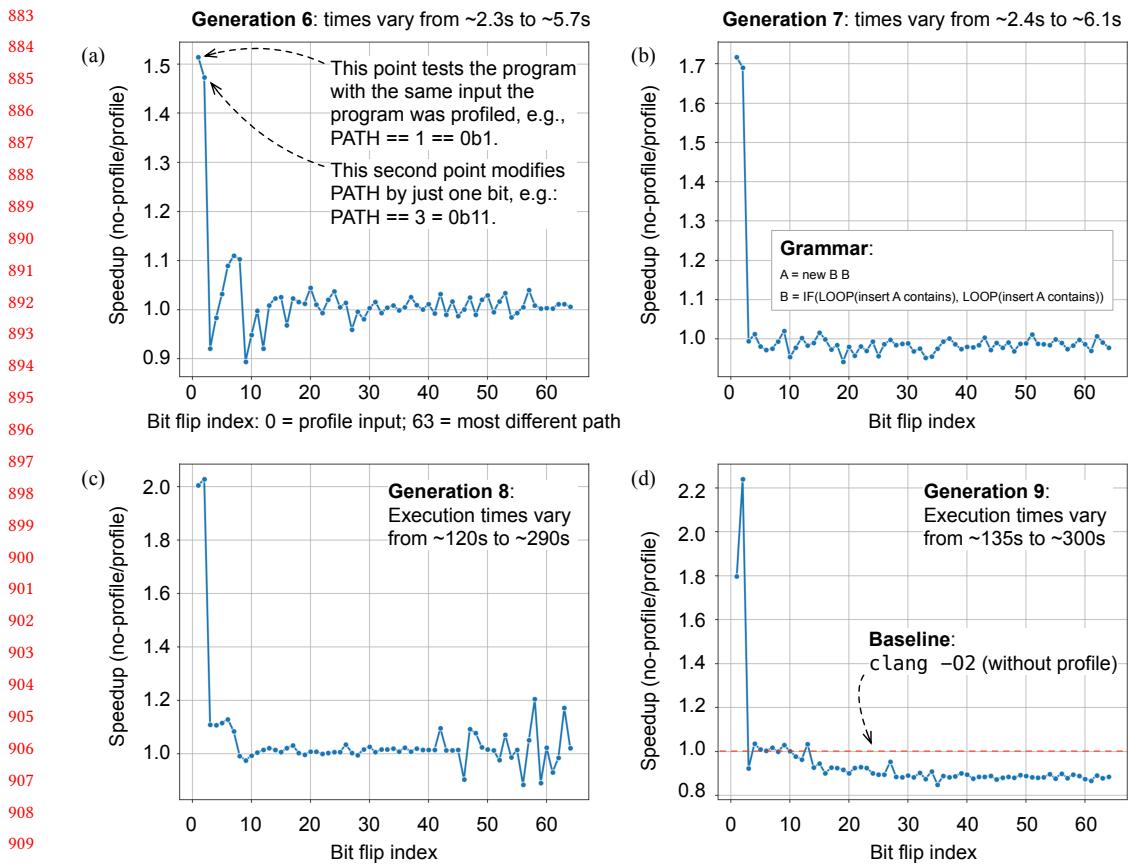


Fig. 16. The impact of profile guided optimizations on clang -O2.

4.6 CS6: Data Structures in GLIB

The GNOME Library (**GLIB**) is a C library developed as part of the GNOME project that provides a wide range of data structure implementations, utility functions, and portability abstractions. Among its generic containers are dynamic arrays (e.g., **GARRAY**, **GPTRARRAY**), byte arrays (**GBYTEARRAY**), singly linked lists (**GSLIST**), doubly linked lists (**GLIST**), double-ended queues (**GQUEUE**), hash tables (**GHASHTABLE**), balanced binary trees (**GTREE**), and others. This variety exists because no single container is optimal for all purposes: some favor fast insertion or deletion (especially at specific positions), others provide efficient random access, better memory locality, sorted traversal, or minimal overhead. By offering multiple data structures, **GLIB** allows programmers to select the container best suited to their performance, memory, and semantic requirements.

BENCHGEN enables the systematic evaluation of the dynamic behavior of these data structures. As discussed in Section 3.1.2, the current implementation of the C code generator supports all twelve containers provided in the default distribution of the library⁵. This section demonstrates how such support can be used to compare the performance characteristics of different containers. To this end, we employ three grammars designed to stress insertion (`insert`), deletion (`remove`),

⁵A detailed description of these containers is available at <https://docs.gtk.org/glib/data-structures.html>.

and search (`contains`) operations through a large number of executions. The adopted methodology is as follows:

- (1) Use the non-recursive L-System below to generate the last iteration of a program:

```
A0 = A1 A1 A1 A1 A1 A1 A1 A1;
A1 = A2 A2 A2 A2 A2 A2 A2 A2;
A2 = A3 A3 A3 A3 A3 A3 A3 A3;
A3 = insert insert;
B = LOOP(CALL(B) C);
C = B {<operations>} B;
```

- (2) The operations symbol acts as a placeholder, expanded into a random sequence of operations (`insert`, `remove`, and `contains`) in varying proportions.
(3) The seed string for the grammars is always “new A0 B”.
(4) Measure the running time of each program using `hyperfine`. Only one warm-up execution is performed due to the long total runtime of all experiments.
(5) Programs with a similar number of operations are grouped together, and we report the average running time within each group.

The first three productions of the grammar above populate the container with a minimum of random integers. The last step of the above methodology—grouping programs by the number of operations—is necessary because it is not possible to precisely control how many iterations are executed within loops generated by `BENCHGEN`. Although the maximum loop trip count is bounded as an input parameter, the actual number of iterations is chosen randomly within that bound.

Discussion. Figure 17 reports the running time of programs generated using the methodology described above. As the number of operations grows, hash tables and balanced trees clearly outperform lists and queues. Between these latter two containers, the queue consistently performs better than the list, except for very small numbers of insertions. This behavior is expected: while both data structures support $O(1)$ insertions, in the doubly linked GLIST two pointers must be updated, whereas in GQUEUE only one pointer is modified.

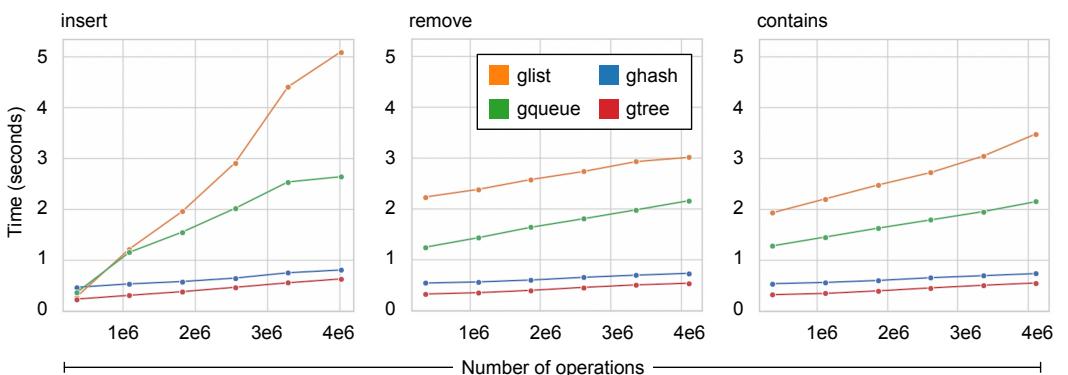


Fig. 17. Comparison of four data structures from GLIB: GHASHTABLE, GTREE, GQUEUE, and GLIST.

At the scale of operations shown in Figure 17, GTREE appears consistently faster than GHASHTABLE, regardless of the operation. However, this situation changes when the number of operations increases further, as illustrated in Figure 18. Around 6 million insertions, the hash table overtakes the balanced tree, and similar crossover points can be observed for deletion and search operations.

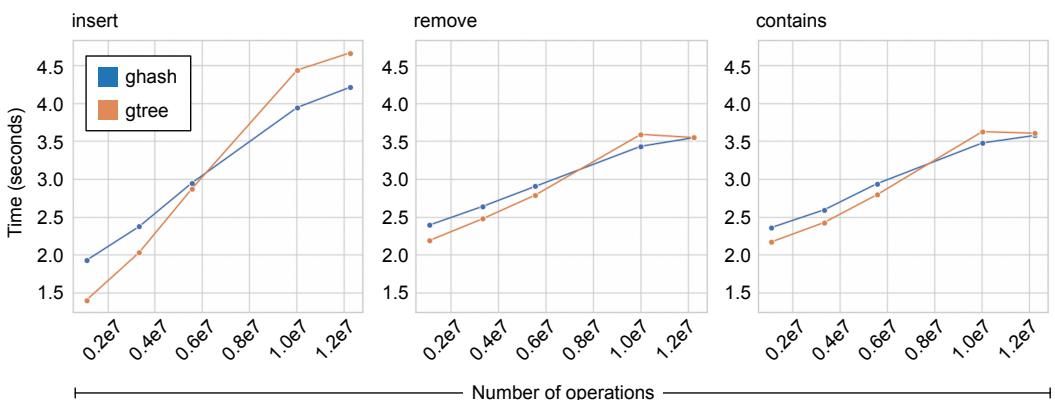


Fig. 18. Comparison of balanced trees (GTREE) and hash tables (GHashTable) from GLIB.

This behavior seen in Figure 18 can be explained by the different asymptotic properties and implementation trade-offs of the two data structures. Balanced trees guarantee $O(\log n)$ time per operation, while hash tables typically achieve $O(1)$ average time, at the cost of higher constant factors due to hashing and possible collisions. For smaller workloads, the overhead of maintaining a hash table, such as computing hash values and managing bucket arrays, outweighs the logarithmic costs of tree operations, making GTREE faster in practice. As the number of operations grows, however, the logarithmic scaling of trees eventually becomes the dominant factor, allowing GHashTable to surpass GTREE. Thus, the crossover point observed in Figure 18 reflects the causal interaction between constant-factor overheads and asymptotic growth rates in these data structures.

4.7 CS7: Comparison with CSMITH

Several programming language fuzzers enjoy widespread popularity. The main goal of such tools, including CSMITH [41] and YarpGen [25] (for C) or CHI GEN [38] (for Verilog), is to uncover bugs in language processing systems such as compilers or interpreters. The purpose of BENCHGEN, however, is different: it was designed to evaluate the performance of these systems. Its ability to expose bugs is limited, since the programs it produces are less diverse than those generated by a fuzzer like CSMITH. On the other hand, BENCHGEN can stress-test compiler performance in ways that CSMITH and related tools cannot. This section illustrates this point through the following challenge: generate a collection of programs whose execution time is comparable to the average runtime of SPEC CPU2017 programs, and whose behavior under optimization resembles that of real-world workloads.

Discussion. To compare CSMITH and BENCHGEN under this challenge, we have used these two program generators to produce a collection of four benchmarks and compare them with the programs from SPEC CPU2017. The adopted methodology is described below:

- (1) We ran the 28 SPEC CPU2017 programs that clang/LLVM 20.1 can compile.
- (2) Using CSMITH, we generated 10,000 programs and selected the four whose execution time was closest to the average runtime of the 28 SPEC CPU2017 programs compiled without optimizations.
- (3) Using the four grammars described in Section 4.1, we generated four BENCHGEN programs whose runtime approximated the same average.

	-O0	-O1	-O2	-O3
BenchGen				
Cycles (Mean)	1.44E+11	5.21E+10	5.20E+10	5.14E+10
Ratio	1	2.76	2.77	2.80
SPEC C2017				
Cycles (Mean)	5.63E+11	2.43E+11	2.37E+11	2.30E+11
Ratio	1	2.32	2.37	2.44
CSmith				
Cycles (Mean)	2.85E+06	1.57E+06	1.49E+06	1.44E+06
Ratio	1	1.85	1.91	1.98

Table 1. Mean cycles and ratios for different benchmarks across optimization levels.

- (4) We then optimized each collection of benchmarks (28, 4, and 4 programs, respectively) under different optimization flags.

Table 1 presents the results of this experiment. We stopped BENCHGEN as soon as we obtained programs whose running time was above the average SPEC time, and chose the generation immediately before. However, it is not possible to steer CSMITH in such a way. In other words, users cannot control CSMITH to produce programs that execute within a given time frame. Most CSMITH programs tend to run for only a very short time. Moreover, the impact of optimizations on these programs differs substantially from what we observe in real-world benchmarks such as those in the SPEC collection, whereas the effects observed on BENCHGEN programs more closely resemble the behavior of genuine workloads. This difference between the effects of compiler optimizations on CSMITH and real-world programs had been noticed in previous work [12].

5 Related Work

BENCHGEN is a tool designed to synthesize benchmarks. A number of works in the compiler literature have also proposed techniques for benchmark generation. In what follows, we review this body of work, emphasizing aspects that make BENCHGEN particularly well-suited for performance analysis.

Compiler Fuzzers. The key distinction between compiler fuzzers and BENCHGEN lies in their primary objectives. Most existing tools are designed to uncover bugs in language processing systems, whereas BENCHGEN is specifically aimed at analyzing their performance. Consequently, BENCHGEN would be less effective than a fuzzer such as CSMITH for identifying errors in C compilers. On the other hand, it enables tasks that are not easily supported by traditional fuzzers, such as studying the asymptotic behavior of compiler optimizations, comparing compilers in terms of compilation time, code size, and execution speed, or analyzing the evolution of a single compiler across multiple versions with respect to these same metrics.

Benchmark Synthesizers. The development of compilers requires *benchmarks*. For this reason, some of the most celebrated papers in programming languages describe benchmark suites, such as SPEC CPU2006 [20], MiBENCH [17], RODINIA [7], etc. These benchmarks are manually curated and usually comprise a small number of programs. A few years ago, Cummins et al. [9] demonstrated that this small size fails to cover the space of program features that a compiler is likely to explore during its lifetime.

The generation of benchmarks for tuning predictive compilers has been an active research field over the last ten years. Early efforts aimed at the development of predictive optimizers used synthetic benchmarks designed to find compiler bugs. Examples of such synthesizers include CSMITH [41], LDRGEN [2], and ORANGE3 [29, 30]. Although conceived as test case generators, these tools have also been used to improve the quality of optimized code emitted by mainstream C compilers [3, 19]. Even COMPILERGYM [10], a tool for applying machine-learning-based code optimization techniques, provides randomly produced CSMITH programs. However, more recent developments indicate that synthetic codes tend to poorly reflect the behavior of programs written by humans; consequently, they produce deficient training sets [12, 15]. Section 4.7 discusses precisely these issues with CSMITH: the difficulty of using it to generate programs that run for a reasonable amount of time and the ease with which compilers optimize such programs.

In order to produce more realistic benchmarks, a vast literature on extracting programs from repositories has recently emerged, seeking to build benchmark suites for training compilers. Some of these works aim to generate benchmarks to feed machine learning models [14, 16, 33], for example. This literature has some shortcomings when compared to BENCHGEN:

- Very large benchmark suites, such as ANGHABENCH [12] or PoJ [28], compile but do not run.
- Benchmark suites designed to run sometimes exhibit undefined behavior, such as EXEBENCH [1] and COLAGEN [5].
- Techniques that generate benchmarks using LLMs, such as the system of Italiano and Cummins [21], fail to produce large programs and take a long time to generate even a small number of programs, as recently shown by Guimarães *et al* [35].
- The JOTAI collection [22], which was built from programs that run without undefined behavior, contains only very small programs. In our attempt to run the programs available in COMPILERGYM [10], the longest-running program took only 0.017 seconds when compiled with clang -O0.

6 Conclusion

This paper has presented a methodology for generating benchmarks based on the observation that programs often exhibit self-similar structures, and thus can be described as derivations of L-Systems. We validated this idea through the design and implementation of BENCHGEN, a tool capable of producing benchmarks in multiple programming languages. In contrast to typical fuzzers, BENCHGEN is multilingual and allows users to generate programs of arbitrary size whose execution exercises complex control-flow patterns.

Future Work. Beyond the concrete implementation of BENCHGEN, the key contribution of this paper is the proposal that programs can be systematically generated as instances of L-Systems. The particular L-System flavor explored here employs a combination of seven constructs (three control-flow structures and four data-structure behaviors). Nevertheless, the approach can be extended to a much broader set of syntactic features. For example, the current version of BENCHGEN generates programs restricted to a single data structure at a time, whereas nothing in the formalism prevents richer combinations of data structures. Likewise, additional control-flow constructs such as switch, do-while, or even goto could be naturally incorporated. We leave the investigation of such extensions as promising directions for future work.

Acknowledgment

This project was supported by Google and by FAPEMIG (Grant APQ-00440-23). We are grateful to Xinliang (David) Li and Victor Lee for their efforts in making the Google sponsorship possible. We

1128 also thank Lucas Victor Silva for producing the data in Table 1, and Bruno Pena Baêta and Matheus
 1129 Alcântara for their work on an earlier version of BENCHGEN.

1130

1131

1132 References

- [1] Jordi Armengol-Estabé, Jackson Woodruff, Alexander Brauckmann, José Wesley de Souza Magalhães, and Michael F. P. O’Boyle. Exebench: an ml-scale dataset of executable c functions. In *MAPS*, page 50–59, New York, NY, USA, 2022. Association for Computing Machinery.
- [2] Gergö Barany. Liveness-driven random program generation. In *LOPSTR*, pages 112–127, Heidelberg, Germany, 2017. Springer.
- [3] Gergö Barany. Finding missed compiler optimizations by differential testing. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 82–92, New York, NY, USA, 2018. ACM.
- [4] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The dogged pursuit of bug-free c programs: the frama-c software analysis platform. *Commun. ACM*, 64(8):56–68, July 2021.
- [5] Maksim Berezov, Corinne Ancourt, Justyna Zawalska, and Maryna Savchenko. COLA-Gen: Active Learning Techniques for Automatic Code Generation of Benchmarks. In Francesca Palumbo, João Bispo, and Stefano Cherubin, editors, *PARMA-DITAM*, volume 100 of *Open Access Series in Informatics (OASIcs)*, pages 3:1–3:14, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [6] Jeff Bezanson, Jiahao Chen, Benjamin Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. Julia: dynamism and performance reconciled by design. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.
- [7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54, Washington, DC, USA, 2009. IEEE.
- [8] Dehao Chen, David Xinliang Li, and Tipp Moseley. Autofdo: automatic feedback-directed optimization for warehouse-scale applications. In *CGO*, page 12–23, New York, NY, USA, 2016. Association for Computing Machinery.
- [9] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Synthesizing benchmarks for predictive modeling. In *CGO*, pages 86–99, Piscataway, NJ, USA, 2017. IEEE.
- [10] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, et al. Compilergym: Robust, performant compiler optimization environments for ai research. In *CGO*, pages 92–105, New York, USA, 2022. IEEE.
- [11] Charlie Curtsinger and Emery D. Berger. Coz: finding code that counts with causal profiling. In *SOSP*, page 184–197, New York, NY, USA, 2015. Association for Computing Machinery.
- [12] Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quintão Pereira. AnghaBench: A suite with one million compilable C benchmarks for code-size reduction. In *CGO*, pages 378–390, Los Alamitos, CA, USA, 2021. IEEE.
- [13] Chucky Ellison and Grigore Rosu. An executable formal semantics of c with applications. *SIGPLAN Not.*, 47(1):533–544, January 2012.
- [14] Jaroslav Fowkes and Charles Sutton. Parameter-free probabilistic api mining across github. In *FSE*, page 254–265, New York, NY, USA, 2016. Association for Computing Machinery.
- [15] Andrés Goens, Alexander Brauckmann, Sebastian Ertel, Chris Cummins, Hugh Leather, and Jeronimo Castrillon. A case study on machine learning for synthesizing benchmarks. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, pages 38–46, New York, NY, USA, 2019. ACM.
- [16] Georgios Gousios and Diomidis Spinellis. Mining software engineering data from github. In *ICSE-C*, page 501–502, Washington, DC, US, 2017. IEEE Press.
- [17] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC*, pages 3–14, Washington, DC, USA, 2001. IEEE.
- [18] Xue Han and Tingting Yu. An empirical study on performance bugs for highly configurable software systems. In *ESEM*, New York, NY, USA, 2016. Association for Computing Machinery.
- [19] Atsushi Hashimoto and Nagisa Ishiura. Detecting arithmetic optimization opportunities for c compilers by randomly generated equivalent programs. *IPSJ Transactions on System LSI Design Methodology*, 9:21–29, 2016.
- [20] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.
- [21] Davide Italiano and Chris Cummins. Finding missed code size optimizations in compilers using large language models. In *International Conference on Compiler Construction*, page 81–91, New York, NY, USA, 2025. Association for Computing Machinery.

1176

- 1177 [22] Cecilia Conde Kind, Michael Canesche, and Fernando Magno Quintao Pereira. Jotai: a methodology for the generation
 1178 of executable c benchmarks. Technical report, Technical Report 02-2022. Universidade Federal de Minas Gerais, 2022.
- 1179 [23] Kota Kitaura and Nagisa Ishiura. Random testing of compilers' performance based on mixed static and dynamic code
 1180 comparison. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection,
 1181 and Evaluation, A-TEST 2018*, page 38–44, New York, NY, USA, 2018. Association for Computing Machinery.
- 1182 [24] Aristid Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs.
Journal of theoretical biology, 18(3):280–299, 1968.
- 1183 [25] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for c and c++ compilers with yarpgen. *Proc.
 1184 ACM Program. Lang.*, 4(OOPSLA), November 2020.
- 1185 [26] Valentin J.M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and
 1186 Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*,
 47(11):2312–2331, 2021.
- 1187 [27] Angélica Aparecida Moreira, Guilherme Ottoni, and Fernando Magno Quintão Pereira. Vespa: static profiling for
 1188 binary optimization. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021.
- 1189 [28] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming
 1190 language processing. In *AAAI*, page 1287–1293, Palo Alto, CA, US, 2016. AAAI Press.
- 1191 [29] Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. Reinforcing random testing of arithmetic optimization of c
 1192 compilers by scaling up size and number of expressions. *IPSJ Trans. System LSI Design Methodology*, 7:91–100, 2014.
- 1193 [30] Kazuhiro Nakamura and Nagisa Ishiura. Introducing loop statements in random testing of c compilers based on
 expected value calculation, 2015.
- 1194 [31] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In
 1195 *PLDI*, page 89–100, New York, NY, USA, 2007. Association for Computing Machinery.
- 1196 [32] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In
 1197 *PLDI*, page 369–378, New York, NY, USA, 2015. Association for Computing Machinery.
- 1198 [33] David N Palacio, Alejandro Velasco, Daniel Rodriguez-Cardenas, Kevin Moran, and Denys Poshyvanyk. Evaluating
 and explaining large language models for code using syntactic structures, 2023.
- 1199 [34] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: a practical binary optimizer for data centers
 1200 and beyond. In *CGO*, page 2–14. IEEE Press, 2019.
- 1201 [35] Gabriel Ricardo, Natanael Santos Junior, Flavio Figueiredo, and Fernando Pereira. On the practicality of llm-based
 1202 compiler fuzzing. In *SBLP*, pages 59–66, Porto Alegre, RS, Brasil, 2025. SBC.
- 1203 [36] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: a fast address
 1204 sanity checker. In *USENIX ATC*, page 28, USA, 2012. USENIX Association.
- 1205 [37] Bob Strecansky. *Hands-On High Performance with Go: Boost and optimize the performance of your Golang applications
 at scale with resilience*. Packt Publishing Ltd, 2020.
- 1206 [38] João Victor Amorim Vieira, Luiza de Melo Gomes, Rafael Sumitani, Raissa Maciel, Augusto Mafra, Mirlaine Crepalde,
 1207 and Fernando Magno Quintão Pereira. Bottom-up generation of verilog designs for testing eda tools, 2025.
- 1208 [39] Zheng Wang and Michael F. P. O'Boyle. Machine learning in compiler optimization. *Proceedings of the IEEE*,
 106(11):1879–1901, 2018.
- 1209 [40] Chunqiu Steven Xia, Matteo Paltinghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing
 1210 with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering,
 1211 ICSE '24*, New York, NY, USA, 2024. Association for Computing Machinery.
- 1212 [41] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings
 1213 of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11*, page 283–294,
 1214 New York, NY, USA, 2011. Association for Computing Machinery.
- 1215
- 1216
- 1217
- 1218
- 1219
- 1220
- 1221
- 1222
- 1223
- 1224
- 1225