Geração de Benchmarks via Sistemas L

Vinicius Francisco da Silva
UFMG
Belo Horizonte, Brazil
silva.vinicius@dcc.ufmg.br

Lucas Victor da Silva Costa
UFMG
Belo Horizonte, Brazil
lucas.victor@dcc.ufmg.br

Matheus Alcântara Souza
Pontifical University Catholic of Minas Gerais
Belo Horizonte, Brazil
matheusalcantara@pucminas.br

Bruno Pena Baêta
Pontifical University Catholic of Minas Gerais
Belo Horizonte, Brazil
brunopenabaeta@outlook.com

Fernando Magno Quintão Pereira
UFMG
Belo Horizonte, Brazil
fernando@dcc.ufmg.br

Resumo

Sistemas L são um formalismo matemático proposto pelo biólogo Aristid Lindenmayer com o objetivo de simular estruturas orgânicas, como árvores, flocos de neve, flores e outros fenômenos ramificados. Eles são implementados como uma linguagem formal que definem como padrões podem ser iterativamente reescritos. Este artigo descreve como tal formalismo pode ser usado para criar programas artificiais escritos na linguagem de programação C. Esses programas, sendo grandes e complexos, podem ser usados para testar o desempenho de compiladores, sistemas operacionais e arquiteturas de computadores. Este artigo demonstra a utilidade de tais *benchmarks* de duas maneiras. Primeiro, ele explica como tais programas permitem averiguar a complexidade de diferentes fases do processo de compilação. Segundo, ele mostra como esses programas permitem comparar compiladores de C, tais como `gcc`, `clang` e `tcc` em termos de tempo de compilação, tamanho e velocidade do código gerado.

CCS Concepts: • **Software and its engineering → Runtime environments**; **Compilers**.

Keywords: Fractal, geração de código, síntese de programas

ACM Reference Format:
Vinicius Francisco da Silva, Lucas Victor da Silva Costa, Matheus Alcântara Souza, Bruno Pena Baêta, and Fernando Magno Quintão Pereira. 2025. Geração de Benchmarks via Sistemas L. In . ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introdução

Compiladores são ferramentas complexas, cujo teste demanda programas escritos na linguagem compilada. Em geral, a quantidade de *benchmarks* disponíveis para um compilador é pequena [26]. Assim, é comum que sejam desenvolvidas ferramentas que fazem a geração automática de programas para testar compiladores [28]. A essa geração automática de entradas de teste damos o nome de *fuzzing* [19]. *Fuzzing* é uma metodologia amplamente utilizada para detecção de *bugs*, como travamentos e vazamentos de memória. Para compiladores, existem *fuzzers* como o Csmith [28] e o YARPGen [18] que geram aleatoriamente programas em C para realizar testes de estresse e análises estáticas. Na literatura, também é possível encontrar *fuzzers* que utilizam Grandes Modelos de Linguagem *Large Language Models (LLM)* para gerar programas, como o Fuzz4All [27], que além de ser utilizado para teste de compiladores, é utilizado também para testar solucionadores de *constraints*, interpretadores e *softwares* ou bibliotecas com APIs acessíveis.

Embora existam muitas ferramentas que geram entradas aleatórias para compiladores [28], acreditamos que os sistemas atuais ainda apresentam limitações. Uma dessas limitações é o tamanho dos programas que são gerados. Por exemplo, CSmith, possivelmente o *fuzzer* de compiladores mais conhecido, não permite que o tamanho dos programas gerados possa ser controlado. Esse tamanho segue uma distribuição normal. Quando esses programas são compilados com clang `-O0`, os programas têm, em média 20.190 instruções LLVM, com desvio padrão de 3.650 instruções, e mediana de 19.161 instruções [8]. Outros *fuzzers* como YarpGen [18], LDRGen [1] e Orange3 [16] apresentam comportamento similar. Como consequência, dificilmente seria possível usá-los para testar o desempenho do compilador (velocidade de *parsing*, análise semântica e/ou geração de código).

Ideia: Programas via Sistemas L. A fim de resolver essa limitação, este artigo propõe uma metodologia para a geração de programas cujo tamanho pode ser facilmente controlado pelo usuário. Essa metodologia permite a criação de códigos

artificiais cujo tamanho somente é limitado pelos recursos disponíveis, a saber, tempo de geração de código e espaço de armazenamento. A ideia básica desta metodologia centra-se na observação da *auto-similaridade de programas*. De fato, programas possuem estrutura recursiva e auto-similar. Por exemplo, os braços de um programa que contem uma estrutura de controle (*if-then-else*) são programas também, possivelmente contendo outras estruturas de controle. A fim de tornar essa metodologia concreta, propomos uma forma de gerar programas baseada na noção de Sistemas-L [17]. A teoria dos sistemas L foi proposta por Aristid Lindenmayer a fim de representar, simular e formalizar o crescimento de estruturas orgânicas. Este artigo expande a noção de sistemas L para geração de código. Nesse sentido, o artigo explica como a estrutura de uma família de programas autosimilares pode ser descrita via uma determinada gramática L. A partir de uma gramática deste tipo podem ser geradas *strings* via um processo iterativo. Cada uma dessas *strings* descreve a estrutura de um programa construído em torno de alguma estrutura de dados, como um arranjo ou lista.

Resumo de Resultados. As ideias discutidas neste artigo foram implementadas em uma ferramenta disponível publicamente via licença GPL 3.0, o gerador de *benchmarks* LGen. A seção 4 descreve a utilidade de LGen de duas formas. Na seção 4.1, mostramos como tal ferramenta pode ser usada para analisar o comportamento assintótico de etapas de compilação. Na seção 4.2 apresentamos uma comparação entre gcc e clang, realizada a partir de programas produzidos por LGen.

2 Sistemas L

Um sistema L (ou *L-system*, de *Lindenmayer system*) é um modelo formal baseado em regras de reescrita usado para descrever o crescimento de plantas e outras estruturas fractais. Ele consiste em um alfabeto de símbolos, um conjunto de regras de produção que transformam esses símbolos, e uma *string* inicial (axioma) que serve como ponto de partida. A cada passo, as regras são aplicadas recursivamente, até que a profundidade passada pelo usuário seja alcançada. A cada recursão vai se gerando sequências cada vez mais complexas. O exemplo 2.1 ilustra o funcionamento desse tipo de formalismo.

Example 2.1. A figura 1 mostra um exemplo de sistemas L. O formalismo apresentado na figura segue um conjunto de regras para gerar padrões geométricos por meio de reescrita de *strings*. O axioma inicial é A, e as produções definem como A e B evoluem a cada iteração: $A \rightarrow B-A-B$ e $B \rightarrow A+B+A$. Aqui, os símbolos $-$ e $+$ representam rotações de 60 e 300 graus, respectivamente. O processo começa com A, que é substituído conforme as regras, gerando sequências cada vez mais complexas. Quando interpretadas graficamente, essas sequências criam curvas fractais, como acontece com o conhecido Triângulo de Sierpinski.

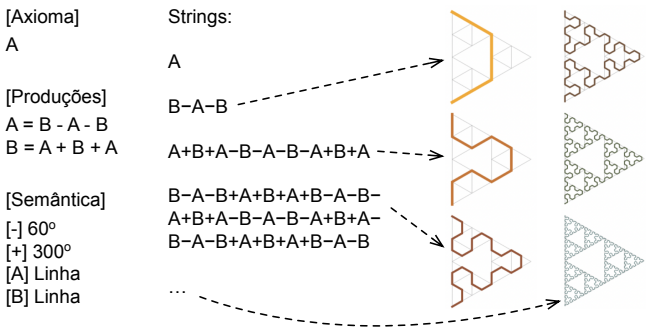

Figura 1. Sistema L que descreve o Triângulo de Sierpinski.

2.1 Programas são Estruturas Auto-Similares

Sistemas L possuem uma propriedade conhecida como *auto-similaridade*. A auto-similaridade caracteriza certas estruturas nas quais partes menores se assemelham ao todo em diferentes escalas. Em sistemas L, essa característica emerge porque as regras de reescrita aplicadas recursivamente geram padrões que repetem a mesma forma em níveis progressivos de detalhe. Isso é evidente em fractais como o Triângulo de Sierpinski visto no exemplo 2.1, onde cada segmento é substituído por uma versão menor de si mesmo, mantendo a estrutura global. Essa repetição hierárquica é essencial para modelar fenômenos naturais, como o crescimento de plantas, a formação de costas marítimas e a distribuição de galhos em árvores.

A auto-similaridade está presente na natureza do código computacional porque muitos algoritmos e estruturas de dados seguem princípios recursivos e hierárquicos. Programas são frequentemente construídos a partir da composição de funções menores, que podem ser chamadas dentro de si mesmas ou organizadas de maneira aninhada, como em estruturas condicionais (`if-then-else`), loops e árvores sintáticas. Além disso, a modularidade e a reutilização de código reforçam essa propriedade, pois funções genéricas podem ser aplicadas repetidamente em diferentes níveis de abstração.

Example 2.2. A figura 2 ilustra a auto-similaridade na estrutura de um bloco `if-then-else` em código de programação. Inicialmente, há uma função `g(x)` definida com um único `if-then-else`. No entanto, essa estrutura pode ser aninhada recursivamente dentro de si mesma, resultando em `g(x) = if g(x) then g(x) else g(x)`. Esse padrão reflete a auto-similaridade porque cada nível da definição contém réplicas menores da própria estrutura condicional. Esse princípio aparece frequentemente em algoritmos recursivos, árvores de decisão e programas que manipulam estruturas aninhadas, como compiladores e interpretadores de linguagens.

Resumo de Ideias. Este artigo explora a noção de auto-similaridade para gerar programas em C que executam sem comportamento indefinido, que podem ser tão grandes e

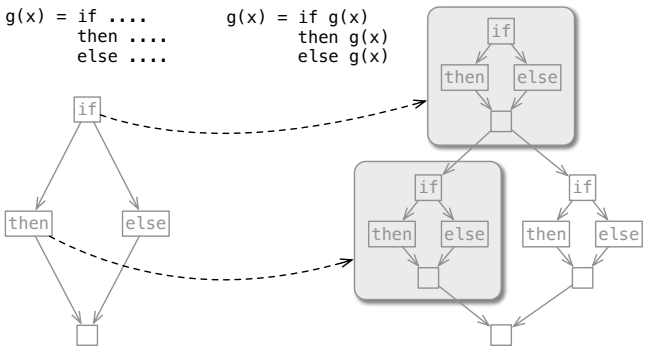

Figura 2. A natureza auto-similar de código computacional.

complexos quanto desejável. O modelo de geração e crescimento de tais programas é descrito via uma gramática L. Porém, ao contrário da figura vista no exemplo 2.1, a semântica de cada construção que emerge de tal gramática é código C, não linhas e ângulos.

3 Geração de Código via Sistemas L

A ferramenta LGen que incorpora as ideias descritas neste trabalho gera programas escritos em C, que operam sobre alguma estrutura de dados. A seção 3.1 descreve os blocos fundamentais que são combinados para formar a estrutura de programas. A forma como podemos guiar a execução de tais programas, por sua vez, é discutida na seção 3.2.

3.1 Blocos Fundamentais

Os sistemas L descritos neste artigo são construídos a partir de duas famílias de construções, as quais descrevemos a seguir:

Estrutura: construções que determinam a estrutura de fluxo de controle de um programa: `IF`, `LOOP` e `CALL`.

Comportamento: construções que determinam o comportamento dinâmico do programa: `new`, `insert`, `remove` e `contains`.

3.1.1 Blocos de Estrutura. O fluxo de controle de um programa produzido por LGen resulta da composição de quatro tipos de blocos de código, que seguem a gramática abaixo:

$$
\begin{array}{rcll}
b & ::= & \text{IF}(b_{\text{cond}}, b_{\text{then}}) & ;; \textit{if\_then} \\
 & | & \text{IF}(b_{\text{cond}}, b_{\text{then}}, b_{\text{else}}) & ;; \textit{if\_then\_else} \\
 & | & \text{LOOP}(b_{\text{cond}}, b_{\text{body}}) & ;; \textit{while} \\
 & | & \text{CALL}(b) & ;; \textit{function\_call}
\end{array}
$$

Cada uma das construções vistas na gramática acima gera uma estrutura de programa diferente, a saber: blocos `if-then` ou `if-then-else`, *loops* do tipo `while` ou invocação de funções. O exemplo 3.1 ilustra esta semântica.

Example 3.1. A figura 3 mostra um exemplo de gramática usada para criar programas. O lado direito da figura contém exemplos de duas iterações desta gramática L. A figura mostra também um código C (simplificado) que resulta da interpretação da *string* formada na segunda iteração da gramática L.

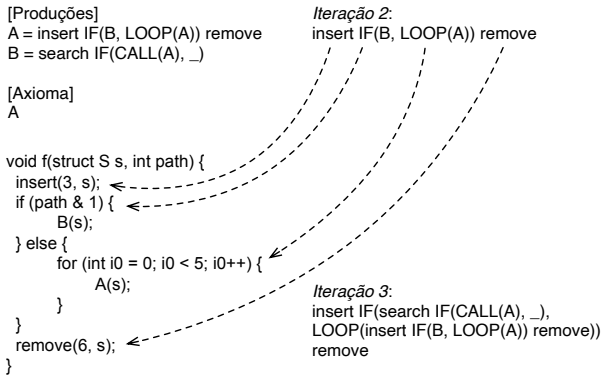

Figura 3. Exemplo de gramática L usada para definir programas.

3.1.2 Blocos de Comportamento. O comportamento dos programas produzidos por LGen emerge a partir de interações feitas sobre estruturas de dados. Todas as estruturas de dados usadas em um programa devem possuir o mesmo tipo. Atualmente, LGen suporta a criação de programas que manipulam arranjos de inteiros ou listas encadeadas ordenadas de inteiros. As operações que podem ser aplicadas sobre as estruturas de dados são definidas por quatro construções presentes em uma gramática L:

new: Cria e inicializa uma estrutura de dados.

insert: Insere um elemento em alguma estrutura de dados que esteja em escopo.

remove: Remove um elemento em alguma estrutura de dados em escopo.

contains: Busca por um elemento em alguma estrutura de dados em escopo

LGen permite que usuários determinem qual código será gerado em cada bloco. Logo, a exata semântica de cada um desses blocos depende de como o usuário de LGen os define. Exemplo 3.2 motram três definições usadas para gerar programas que manipulam arranjos.

Example 3.2. A figura 4 mostra código que é gerado para programas que manipulam arranjos. Um tal programa pode criar e modificar centenas de instâncias diferentes de estruturas deste tipo. Os códigos da figura 4 se referem a uma particular variável: `array156`. As operações que são realizadas sobre um arranjo são definidas pelo usuário de LGen, que precisa sobrescrever cada um dos métodos genéricos `new()`, `insert()` e `contains()`.

```
new
    array_t* array156;
    array156 = (array_t*)malloc(sizeof(array_t));
    array156->size = 42;
    array156->refC = 1;
    array156->id = 156;
    array156->data = (unsigned int*)malloc(array156->size*sizeof(unsigned int));
    memset(array156->data, 0, array156->size*sizeof(unsigned int));
    DEBUG_NEW(array157->id);
insert
    unsigned int loop46 = 0;
    unsigned int loopLimit46 = (rand()%loopsFactor)/2 + 1;
    for(; loop46 < loopLimit46; loop46++) {
      for (int i = 0; i < array156->size; i++) {
        array156->data[i]--;
      }
    }
constains
    unsigned int loop46 = 0;
    unsigned int loopLimit46 = (rand()%loopsFactor)/2 + 1;
    for (int i = 0; i < array156->size; i++) {
      if (array156->data[i] == 61) {
        break;
      }
    }
```

Figura 4. Exemplos de definições de blocos `new`, `insert` e `contains`.

3.2 Fluxo de execução

Os programas produzidos via LGen executam. O fluxo de execução é controlada por um parâmetro: uma variável `path`, de tipo inteiro sem sinal, que determina o resultado de desvios condicionais. O i-ésimo bit the `path` determina o resultado de todos os testes condicionais realizados em desvios com fator de aninhamento i. O Exemplo 3.3 provê uma explicação mais clara.

Example 3.3. A figura 5 mostra o grafo de fluxo de execução de um programa dividido em regiões aninhadas. Cada região possui um fator de aninhamento. Se o fator de aninhamento de uma região R for um valor d, então R encontra-se aninhada dentro de d outras regiões. Conforme mostra a figura 5, o fator de aninhamento de uma região R determina o resultado do desvio condicional (se houver) que encabeça R. Assim, o teste condicional no bloco `%9` é definido pelo resultado de `path & 2`, já que este bloco encabeça uma região aninhada dentro de duas outras regiões.

3.2.1 Chamadas de Funções.

LGen suporta a criação de programas com chamadas de funções, via a cláusula `CALL`. Assim, a presença da sequência `CALL`(e) em uma *string* faz com que toda a *substring* e seja definida em uma função separada. O exemplo 3.4 explica em maiores detalhes como se dá a geração de código interprocedural.

Example 3.4. A figura 6 mostra o fluxo de controle que resulta de um bloco `CALL`. Toda a *string* dentro de uma construção deste tipo dará origem a uma nova função que integra o programa sintetizado. Essa nova função é invocada no ponto em que a construção `CALL` ocorre na *string* L.

Note que todas as construções `CALL`(b) contendo exatamente as mesmas *strings* b dão origem a chamadas da mesma

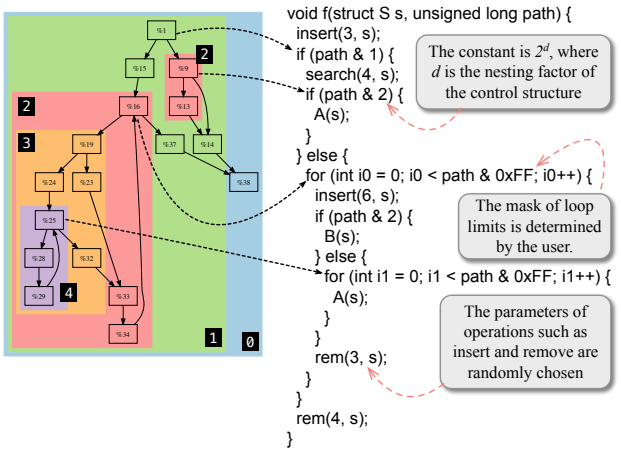

Figura 5. O fluxo de controle de um programa sintético é determinado pelo parâmetro `path`.

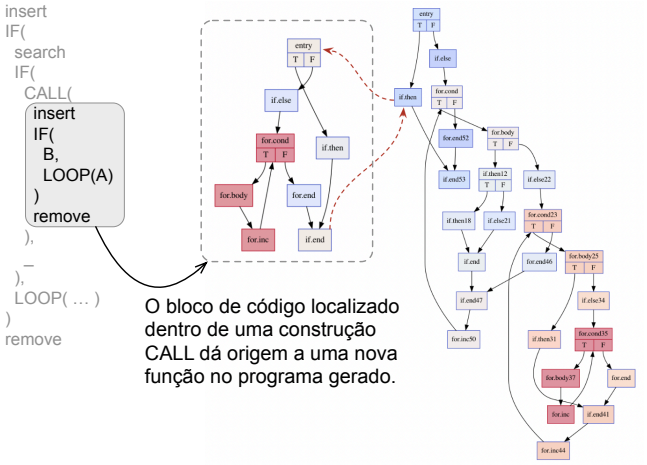

Figura 6. Fluxo de controle de programa que contém um bloco do tipo `CALL`.

função. Para evitar redundâncias, LGen mantém uma tabela que associa *strings* a funções, de modo que *strings* iguais representam a mesma função. As várias funções que compõe o programa gerado a partir de uma dada especificação L podem ser todas agrupadas em um mesmo arquivo, ou podem ser criadas em arquivos separados. Um parâmetro da ferramenta LGen determina se funções são separadas em arquivos ou não.

Passagem de Parâmetros. Funções geradas por LGen recebem dois parâmetros:

- **Data:** um arranjo de estruturas de dados, que permite que estruturas criadas em uma função possam ser usadas dentro de outra.
- **Path:** a variável que controla o fluxo de execução, conforme explicado no Exemplo 3.3.

O arranjo **Data** é preenchido com auxílio de uma análise de definições alcançáveis, que determina, dentro da função chamadora, quais variáveis estão disponíveis para serem usadas como parâmetro no ponto de chamada. O exemplo 3.5 mostra como se dá a passagem de parâmetros.

Example 3.5. A figura 7 mostra como a passagem de parâmetros acontece em programas criados por LGen. No ponto em que uma função é chamada, uma análise de definições alcançáveis determina que três variáveis, `array1`, `array156` e `array157` estão disponíveis. Ponteiros para essas variáveis são inseridos dentro da estrutura `param.data`, que é então passada como parâmetro para a função `func0`. Dentro desta função, essas variáveis são copiadas para as novas variáveis criadas via `new`. Um tal parâmetro é somente copiado uma vez. Caso não haja mais parâmetros disponíveis para cópia, próximas ocorrências da estrutura `new` vão criar novas variáveis, conforme visto no exemplo 3.3.

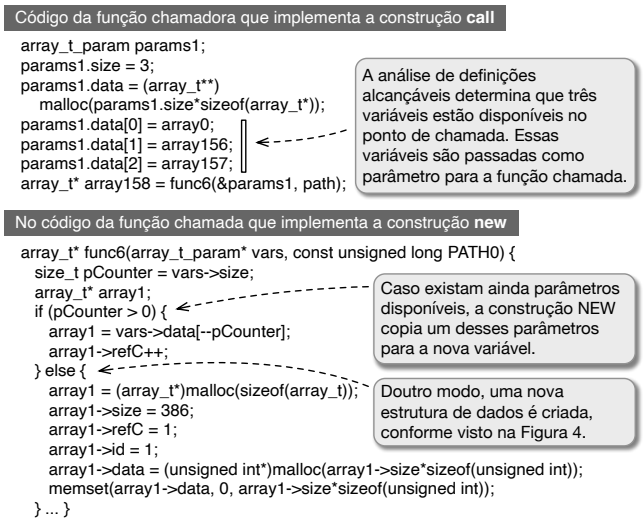

Figura 7. Passagem de parâmetros em programas criados por LGen.

3.2.2 Gerenciamento de Memória. Programas criados por LGen não apresentam vazamento de memória, embora usem com frequência a alocação de espaço em *heap*. Para evitar vazamento de memória, LGen usa um coletor de lixo baseado em contagem de referências. Um coletor de lixo baseado em contagem de referências é um mecanismo de gerenciamento automático de memória que rastreia o número de referências (ponteiros) para cada objeto alocado dinamicamente. Sempre que uma referência é criada, o contador é incrementado, e quando uma referência é removida, o contador é decrementado. Quando o contador atinge zero, significa que o objeto não é mais acessível pelo programa e sua memória pode ser liberada. Para implementar o coletor, cada estrutura definida por LGen possui um campo extra, `refC`, que conta referências para aquela estrutura.

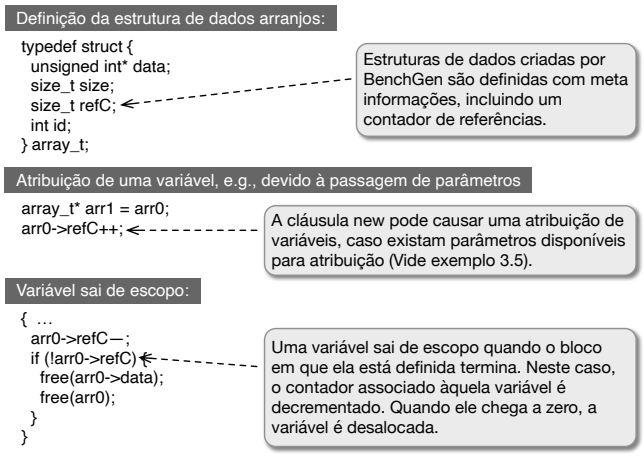

Figura 8. Implementação do contador de referências.

4 Avaliação Experimental

O objetivo desta seção é demonstrar que LGen pode ser utilizado para avaliar diferentes aspectos da implementação de compiladores. Com este fim, mostraremos como programas produzidos automaticamente por LGen podem ser usados para responder duas questões de pesquisa, a saber:

RQ1: Qual é o comportamento assintótico de diferentes fases do processo de compilação implementado por `clang`?

RQ2: Como se comparam `clang` e `gcc` em termos de tempo de compilação, tamanho de binários e qualidade de código produzido?

4.1 RQ1: Comportamento Assintótico

O comportamento assintótico de uma implementação de algoritmo descreve como seu tempo de execução ou uso de memória que cresce à medida que o tamanho da entrada aumenta. No caso de um compilador, seu comportamento assintótico refere-se à forma como o tempo de compilação e o uso de memória crescem à medida que o tamanho do código-fonte aumenta. Isso pode depender de vários fatores, como a complexidade dos algoritmos de análise léxica/sintática, otimizações e processo de geração de código. Analisar o comportamento assintótico de um compilador ajuda a prever sua escalabilidade em programas grandes e pode revelar *bugs* de desempenho. Nesta seção, usamos LGen para efetuar tal estudo sobre `clang`.

Metodologia. A fim de analisar o comportamento assintótico de `clang`, iremos analisar o tempo que cada uma de suas fases leva para processar oito programas diferentes. Cada programa representa uma iteração diferente da mesma gramática, começando pela quinta, e indo até a décima segunda. A gramática em questão pode ser vista na figura 9 (c). Para este estudo, escolhemos uma gramática que leva a um crescimento aproximadamente linear com relação às iterações

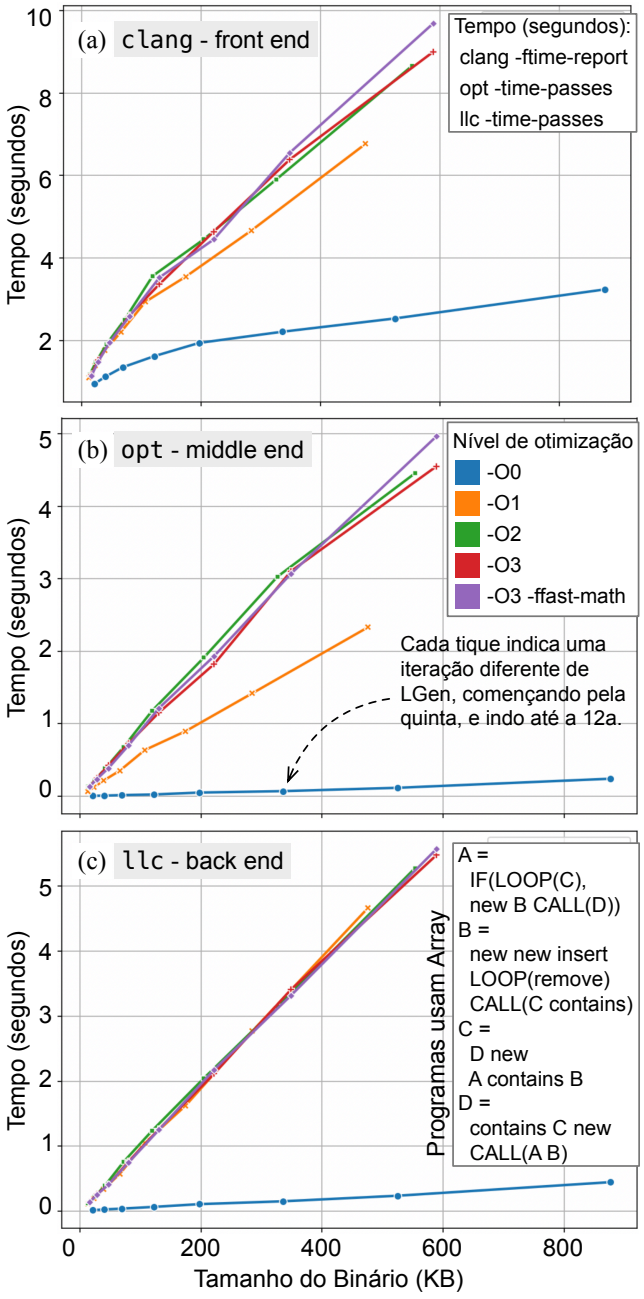

Figura 9. Complexidade assintótica de fases de compilação.

de LGen. As diferentes fases de compilação de `clang` são separadas da seguinte maneira:

Front: `clang -S -emit-llvm -ftime-report -Ox`
Middle: `opt -time-passes -Ox`
Back: `llc -time-passes -Ox -march=x86`

O parâmetro `Ox` denota o nível de compilação. Neste estudo, consideramos cinco níveis, a saber: `-O0`, `-O1`, `-O2`, `-O3`, e `-O3 -ffast-math`.

Discussão: A figura 9 sumariza os resultados deste estudo. Uma das conclusões mais imediatas é que o comportamento assintótico das diferentes fases de compilação tende a ser linear com relação ao tamanho dos programas compilados. Notamos também que não parece existir diferença relevante entre comportamento assintótico dos níveis mais altos de `clang`. Com efeito, otimizações em `O2`, `O3` e `O3 -ffastmath` parecem levar aproximadamente o mesmo tempo para serem executadas por `opt`. O fato de que não existe diferença estatisticamente significante entre essas fases de compilação, em `clang`, já foi reportado em outras ocasiões[1]. Contudo, é fácil observar, na figura 9 (b), que `clang -O1` é substancialmente mais rápido do que os outros níveis de otimização – algo esperado, já visto que neste nível `clang` deixa de executar várias otimizações custosas, como *inlining* e vetorização.

4.2 RQ2: Comparação de Compiladores

Programas gerados por LGen podem ser compilados por diferentes compiladores como `tcc`, `gcc` e `clang`. Comparações então, podem ser úteis para encontrar vantagens e desvantagens sobre o uso de um determinado compilador. Tais comparações podem ser realizadas em diferentes dimensões como: tempo de compilação, tamanho de binários e qualidade do código produzido. Esse último quesito, qualidade de código, é definido como o tempo de execução dos programas compilados. Nesta sessão demonstramos como LGen pode ser usado com tal propósito.

Metodologia: Esta seção compara dois compiladores de C, a saber: `clang 21.0` e `gcc 14.0`. Tempo de compilação e tempo de execução são obtidos via *hyperfine* [23]. Hyperfine é uma ferramenta de *benchmarking* que oferece medições estatisticamente robustas, executando múltiplas iterações automaticamente, descartando outliers, calculando médias, desvio padrão e mostrando comparações entre execuções. Nesta seção, configuramos Hyperfine para ignorar as duas primeiras execuções de cada *benchmark*, a fim de que o tempo de transferência das dependências do compilador para a memória principal na primeira execução não influencie o resultado do experimento. Hyperfine também foi configurado para executar no máximo cinquenta vezes, caso o tempo medido não estabilize. O tamanho do binário é extraído via o comando `size`. Nesta seção, reportamos somente o tamanho do segmento de texto; ou seja, as instruções de máquina.

Para a comparação, foram produzidos oito programas, a partir de duas gramáticas. Quatro desses programas utilizam arranjos (*Array*) como a estrutura de dados e os outros quatro utilizam listas encadeadas ordenadas (*SortedList*). Assim, cada um dos resultados que discutiremos combina oito amostras. Tempo de execução do programa e o tempo de compilação é a média de cinquenta execuções do programa e

[1]Uma análise detalhada dessa diferença pode ser acompanhada em uma palestra de *Emery Berger*, que encontra-se disponível somente em vídeo (https://youtu.be/r-TLSBdHe1A?). A discussão começa por volta de 23:40.

cinquenta compilações do programa respectivamente; logo, resultados de tempo envolvem 8×50 amostras.

Discussão: A figura 10 mostra os resultados obtidos via a metodologia anteriormente descrita. Observa-se que `clang` apresenta tempos de compilação maiores que `gcc-14` em quase todos os casos. Conforme já discutido na seção 4.1, programas otimizados por `-O3` e `-O2` e `-Ofast` no `clang` ficam entre os que apresentam maiores tempos de compilação, justificado pela quantidade de passes de otimização realizados.

Em termos de tempo de execução, não encontramos diferença substancial entre os dois compiladores, quando consideramos os níveis mais altos de otimização. Em relação ao tamanho do binário, nota-se, conforme esperado, o efeito positivo dos níveis `-Os` (no caso dos dois compiladores) e `-Oz` (no caso de `clang`) Por outro lado, os níveis de otimização `-O3` e `-Ofast` causam aumento de tamanho. Nota-se que `clang` consegue gerar binários um pouco menores que `gcc` quando ambos os compiladores são usados em níveis mais agressivos como o `-O3` e `-Ofast`. Observamos também que há situações em que `clang`, no nível `-O2` produz código que executa mais rápido do que o código produzido no nível de otimização `-O3`, corroborando observações feitas na seção 4.1.

5 Trabalhos Relacionados

O desenvolvimento de compiladores requer *benchmarks*. Por isso, alguns dos artigos mais celebrados em linguagens de programação descrevem conjuntos de *benchmarks*, como Spec CPU2006 [15], MiBench [13], Rodinia [4], etc. Esses *benchmarks* são curados manualmente e normalmente compreendem um pequeno número de programas. Recentemente, Cummins et al. [6] demonstraram que este tamanho reduzido falha em cobrir o espaço de características de programas que um compilador provavelmente explorará durante sua existência. Assim, pesquisadores e entusiastas têm trabalhado para gerar um grande número de *benchmarks* diversos e expressivos. Esta seção aborda alguns desses esforços.

Síntese Aleatória. A geração de *benchmarks* para ajustar compiladores preditivos tem sido um campo ativo de pesquisa nos últimos dez anos. Esforços iniciais direcionados ao desenvolvimento de otimizadores preditivos utilizavam *benchmarks* sintéticos concebidos para encontrar *bugs* em compiladores. Exemplos desses sintetizadores incluem CSmith [28], LDRGen [1] e Orange3 [20, 21]. Embora concebidos como geradores de casos de teste, essas ferramentas também foram usadas para melhorar a qualidade do código otimizado emitido por compiladores C *mainstream* [2, 14]. Até mesmo o CompilerGym [7], uma ferramenta para aplicação de técnicas de otimização de código baseadas em aprendizagem automática, fornece programas CSmith produzidos aleatoriamente. Entretanto, desenvolvimentos mais recentes indicam que códigos sintéticos tendem a refletir pobremente o comportamento de programas escritos por humanos; consequentemente, produzindo conjuntos de treinamento deficientes [8, 11].

Síntese Guiada. Vários grupos de pesquisa têm usado abordagens guiadas para sintetizar *benchmarks* [3, 5, 6, 9]. Essas técnicas podem se basear em um template de códigos aceitáveis, como Deniz et al. [9] fazem, ou podem usar um modelo de aprendizado de máquina para direcionar a geração de programas, como Cummins et al. [6] ou Berezov et al. [3] fazem. A síntese é restrita a um domínio específico, como *kernels* OpenCl [6, 9]; ou *loops* regulares [3]. LGen é uma forma de síntese guiada; porém, dentre os trabalhos citados, é a única que utiliza sistemas L como guia.

Mineração de Código. Existe uma vasta literatura sobre extração de programas de repositórios que busca construir conjuntos de *benchmarks* para treinar compiladores. Alguns desses trabalhos visam gerar *benchmarks* para alimentar modelos de aprendizado de máquina [10, 12, 22], por exemplo. Em contraste, *benchmarks* usados para treinar modelos de linguagem grande para geração de código são formados por trechos de programas que devem ser analisáveis, mas não necessariamente compiláveis [22, 25]. Observe que repositórios de código aberto não são a única fonte de *benchmarks* para popular tais modelos. Por exemplo, Richards *et al.* produziram *benchmarks* realistas de JavaScript a partir de seções de navegadores monitoradas [24]. Uma limitação da abordagem de Richards [24] é a inconveniência: um ser humano ainda precisa criar uma seção de navegação que dará origem a um *benchmark*. LGen não minera repositórios para gerar *benchmarks* – ao contrário, tais *benchmarks* são criados de forma artificial.

6 Conclusão

Este artigo apresentou a ferramenta LGen, que utiliza Sistemas L para gerar *benchmarks* sintéticos em C com tamanho controlável, preenchendo uma lacuna deixada por ferramentas de *fuzzing* tradicionais. Ao explorar a auto-similaridade inerente a programas computacionais, LGen demonstrou ser eficaz na comparação de desempenho entre diferentes compiladores, como clang e gcc. Os resultados experimentais destacaram a capacidade da ferramenta em produzir programas complexos e escaláveis, úteis para testar compiladores em diversos cenários. LGen oferece uma abordagem inovadora para a geração de código, combinando princípios matemáticos com necessidades práticas de engenharia de *software*. Futuros trabalhos podem expandir essa metodologia para outras linguagens de programação além de C, ou a mais estruturas de dados, além de arranjos e listas encadeadas, ampliando ainda mais a aplicabilidade dessa metodologia no desenvolvimento e teste de ferramentas de compilação.

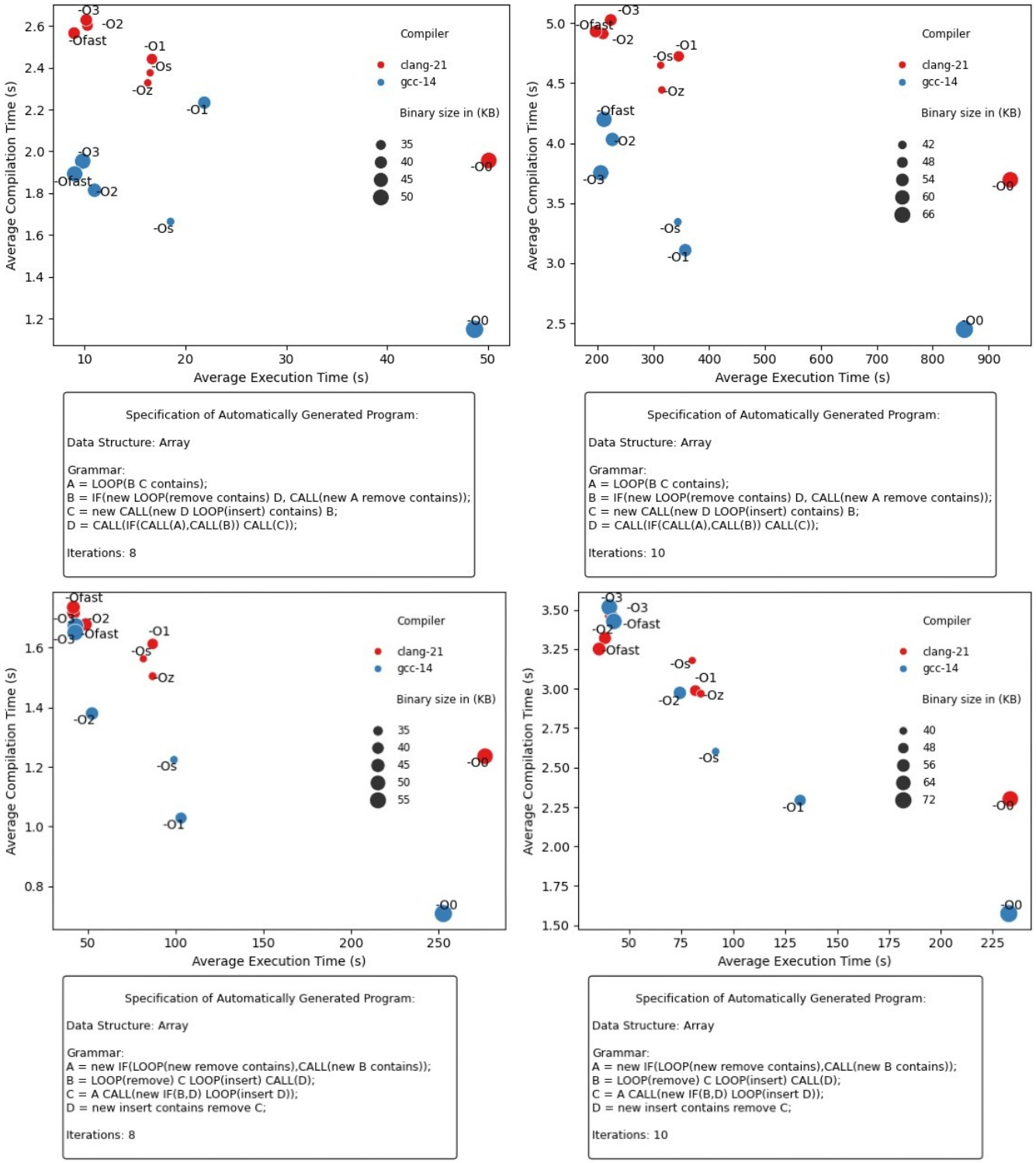

Figura 10. Comparação de diferentes compiladores.

Referências

[1] Gergö Barany. 2017. Liveness-Driven Random Program Generation. In *LOPSTR*. Springer, Heidelberg, Germany, 112–127. doi:10.1007/978-3-319-94460-9_7

[2] Gergö Barany. 2018. Finding Missed Compiler Optimizations by Differential Testing. In *Proceedings of the 27th International Conference on Compiler Construction* (Vienna, Austria) *(CC 2018)*. ACM, New York,

NY, USA, 82–92. doi:10.1145/3178372.3179521

[3] Maksim Berezov, Corinne Ancourt, Justyna Zawalska, and Maryna Savchenko. 2022. COLA-Gen: Active Learning Techniques for Automatic Code Generation of Benchmarks. In *PARMA-DITAM (Open Access Series in Informatics (OASIcs), Vol. 100)*, Francesca Palumbo, João Bispo, and Stefano Cherubin (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 3:1–3:14. doi:10.4230/OASIcs.PARMA-DITAM.2022.3

[4] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*. IEEE, Washington, DC, USA, 44–54. doi:10.1109/IISWC.2009.5306797

[5] Alton Chiu, Joseph Garvey, and Tarek S. Abdelrahman. 2015. Genesis: A Language for Generating Synthetic Training Programs for Machine Learning. In *CF* (Ischia, Italy). Association for Computing Machinery, New York, NY, USA, Article 8, 8 pages. doi:10.1145/2742854.2742883

[6] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Synthesizing Benchmarks for Predictive Modeling. In *CGO*. IEEE, Piscataway, NJ, USA, 86–99. doi:10.1109/CGO.2017.7863731

[7] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, et al. 2022. Compilergym: Robust, performant compiler optimization environments for ai research. In *CGO*. IEEE, New York, USA, 92–105.

[8] Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quintão Pereira. 2021. AnghaBench: A Suite with One Million Compilable C Benchmarks for Code-Size Reduction. In *CGO*. IEEE, Los Alamitos, CA, USA, 378–390. doi:10.1109/CGO51591.2021.9370322

[9] Etem Deniz and Alper Sen. 2015. MINIME-GPU: Multicore Benchmark Synthesizer for GPUs. *ACM Trans. Archit. Code Optim.* 12, 4, Article 34 (nov 2015), 25 pages. doi:10.1145/2818693

[10] Jaroslav Fowkes and Charles Sutton. 2016. Parameter-Free Probabilistic API Mining across GitHub. In *FSE* (Seattle, WA, USA). Association for Computing Machinery, New York, NY, USA, 254–265. doi:10.1145/2950290.2950319

[11] Andrés Goens, Alexander Brauckmann, Sebastian Ertel, Chris Cummins, Hugh Leather, and Jeronimo Castrillon. 2019. A Case Study on Machine Learning for Synthesizing Benchmarks. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. ACM, New York, NY, USA, 38–46. doi:10.1145/3315508.3329976

[12] Georgios Gousios and Diomidis Spinellis. 2017. Mining Software Engineering Data from GitHub. In *ICSE-C* (Buenos Aires, Argentina). IEEE Press, Washington, DC, US, 501–502. doi:10.1109/ICSE-C.2017.164

[13] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *WWC*. IEEE, Washington, DC, USA, 3–14. doi:10.1109/WWC.2001.15

[14] Atsushi Hashimoto and Nagisa Ishiura. 2016. Detecting arithmetic optimization opportunities for C compilers by randomly generated equivalent programs. *IPSJ Transactions on System LSI Design Methodology* 9 (2016), 21–29.

[15] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. doi:10.1145/1186736.1186737

[16] Kota Kitaura and Nagisa Ishiura. 2018. Random testing of compilers' performance based on mixed static and dynamic code comparison. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation* (Lake Buena Vista, FL, USA) *(A-TEST 2018)*. Association for Computing Machinery, New York, NY, USA, 38–44. doi:10.1145/3278186.3278192

[17] Aristid Lindenmayer. 1968. Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. *Journal of theoretical biology* 18, 3 (1968), 280–299.

[18] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (Nov. 2020), 25 pages. doi:10.1145/3428264

[19] Valentin J.M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2312–2331. doi:10.1109/TSE.2019.2946563

[20] Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. 2014. Reinforcing Random Testing of Arithmetic Optimization of C Compilers by Scaling up Size and Number of Expressions. *IPSJ Trans. System LSI Design Methodology* 7 (2014), 91–100.

[21] Kazuhiro Nakamura and Nagisa Ishiura. 2015. Introducing Loop Statements in Random Testing of C compilers Based on Expected Value Calculation. 226–227 pages.

[22] David N Palacio, Alejandro Velasco, Daniel Rodriguez-Cardenas, Kevin Moran, and Denys Poshyvanyk. 2023. Evaluating and Explaining Large Language Models for Code Using Syntactic Structures. arXiv:2308.03873 [cs.SE]

[23] David Peter. 2023. *hyperfine*. SharkDP. https://github.com/sharkdp/hyperfine

[24] Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. 2011. Automated Construction of JavaScript Benchmarks. *SIGPLAN Not.* 46, 10 (2011), 677–694.

[25] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *CHI EA* (New Orleans, LA, USA). Association for Computing Machinery, New York, NY, USA, Article 332, 7 pages. doi:10.1145/3491101.3519665

[26] Zheng Wang and Michael F. P. O'Boyle. 2018. Machine Learning in Compiler Optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901. doi:10.1109/JPROC.2018.2817118

[27] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 126, 13 pages. doi:10.1145/3597503.3639121

[28] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. doi:10.1145/1993498.1993532