Synthesis of Benchmarks for the C Programming Language by Mining Software Repositories

Breno C F Guimarães
DCC
UFMG
Belo Horizonte, Minas Gerais, Brazil
brenosfg@dcc.ufmg.br

José Wesley de S Magalhães
DCC
UFMG
Belo Horizonte, Minas Gerais, Brazil
jwdesmagalhaes@gmail.com

Anderson Faustino da Silva
DIN
UEM
Maringá, Paraná, Brazil
anderson@din.uem.br

Fernando M Q Pereira
DCC
UFMG
Belo Horizonte, Minas Gerais, Brazil
fernando@dcc.ufmg.br

Resumo

Compilers are usually distributed with a test framework. This framework supports the task of tuning optimizations and static analyses. As an example, clang has a test suite that, in March 2019, counted 259 benchmarks. Although in principle a large collection, this number is small once we consider the needs of the automatic tuning techniques that became fashionable recently. To mitigate the problems caused by such lack of benchmarks, this paper introduces a technique that allows the automatic construction of compilable programs out of open-source repositories. Our approach has made it possible to build, in less than 24 hours, a collection with over 500 thousand functions that clang can compile. In this paper, we show that such abundance of data gives us precise information about the behavior of compiler optimizations, and lets us create accurate prediction models. This collection of benchmarks is today freely available to the open-source community.

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

Keywords Benchmarks, Repositórios, Síntese

SBLP 2019, September 23–27, 2019, Salvador, Brazil

ACM ISBN 978-1-4503-7638-9/19/09…$15.00
https://doi.org/10.1145/3355378.3355380

ACM Reference Format:
Breno C F Guimarães, José Wesley de S Magalhães, Anderson Faustino da Silva, and Fernando M Q Pereira. 2019. Synthesis of Benchmarks for the C Programming Language by Mining Software Repositories. In *XXIII Brazilian Symposium on Programming Languages (SBLP 2019), September 23–27, 2019, Salvador, Brazil.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3355378.3355380

1 Introdução

A crescente popularização de técnicas de auto-adaptação em compiladores tem causado uma revolução nesta área de pesquisa. As mais vatriadas técnicas podem ser usadas hoje para configurar sistemas de compilação e execução de programas de forma automática [2]. Essas técnicas possuem um modo comum de operação: o comportamento do compilador é testado em um conjunto de programas, e, a partir desse treinamento, tal comportamento é repetido sobre programas similares. Em outras palavras, programas que apresentam estrutura sintática próxima são tratados da mesma maneira durante sua análise e otimização. Um problema dessa abordagem é a necessidade de uma grande quantidade de benchmarks: programas que possam ser usados para apurar o comportamento do compilador ou do analisador estático.

A carência de benchmarks é um problema sério. Conforme descrito por Cummins *et al.* [10], muitos trabalhos importantes na área de linguagem de programação utilizam uma quantidade insuficiente de programas para treinamento. Em um estudo de 25 trabalhos publicados em conferências e periódicos da área de compiladores, Cummins *et al.* constataram que, em média, cada artigo usada 17 benchmarks para validar hipóteses acerca do comportamento de programas. Conjuntos pequenos de treinamento levam a criação de técnicas de auto-adaptação sub-óptimas. A fim de lidar com a falta de benchmarks, existem mecanismos para gerar programas de forma automática. Possivelmente, a técnica mais conhecida desse tipo é aquela empregada pelo gerador de programas CSmith [23]. CSmith produz programas na

linguagem C. Embora essa ferramenta seja muito usada para gerar conjuntos de testes, seu propósito original não é predizer o comportamento de compiladores, mas descobrir bugs neles. Na seção 3 deste artigo demonstramos que programas gerados por CSmith não podem ser usados para prever adequadamente o efeito de otimizações sobre programas reais.

A fim de suprir a falta de benchmarks, este artigo descreve uma metodologia para gerar programas compiláveis na linguagem C, a partir de códigos reais. Tais códigos são minerados a partir de repositórios públicos. Um dos empecilhos desta abordagem –frequentemente usada como justificativa para sua não adoção– é o fato de que tais programas não podem ser compilados automaticamente [9–11]. Para contornar tal desafio, nós utilizamos um inferidor de tipos feito para a linguagem C [17] para reconstruir as dependências que faltam nos programas minerados automaticamente. Essa ferramenta, Psyche-C, nos permite compilar grandes quantidades de programa de forma automática. Dependências são reconstruídas, e, ao final do processo, um compilador como clang ou gcc é usado para validar o código gerado. A execução dos benchmarks que criamos pode levar a comportamento indefinido –o que também acontece com outros programas produzidos automáticamente [3, 4, 11]. Porém, conforme demonstraremos na seção 3 deste trabalho, ainda assim eles podem ser empregados em diversas técnicas de auto-adaptação. Em particular, nossos benchmarks podem ser usados para apurar otimizações de código que buscam reduzir o tamanho de programas.

Como forma de validar nossa metodologia de criação de benchmarks, nós criamos uma ferramenta, Angha, que produz programas C compiláveis a partir de código disponível em repositórios públicos. Nós utilizamos Angha para construir uma coleção de programas contendo 530.000 amostras compiláveis. Tal conjunto chama-se AnghaBench. Na seção 3, nós demonstramos que AnghaBench é consistentemente melhor que CSmith para aproximar o comportamento de programas reais. Para demonstrar a utilização desse conjunto de benchmarks, nós mostraremos como ele pode ser usado para prever o comportamento de Clang. Com tal propósito, mostramos que AnghaBench prevê com grande acurácia o efeito de diferentes níveis de otimização (-O1 e -O3) sobre o tamanho dos programas de teste disponíveis na infra-estrutura de compilação LLVM [14].

2 Construção Automática de Benchmarks

A fim de melhor explicar o funcionamento de Angha, nosso gerador de benchmarks, nós o dividimos em quatro partes. Cada uma dessas partes consiste de uma ferramenta independente. Essas ferramentas são o Rastreador Web (seção 2.1), o Extrator de Funções (seção 2.2), o Reconstrutor de Tipos (seção 2.3) e o Validador de Arquivos (seção 2.4). A figura 1 mostra como essas diferentes partes se relacionam. A seguir, detalhamos cada uma dessas partes.

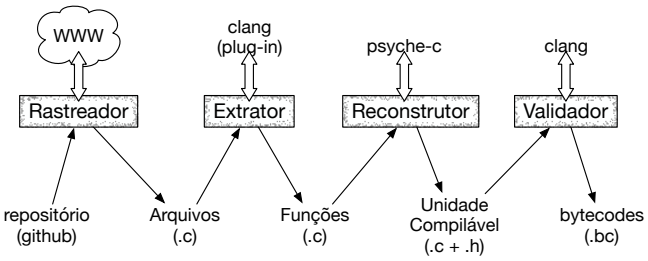

Figura 1. O Arcabouço Angha para construção automática de benchmarks. Acima de cada parte do sistema é vista a infra-estrutura em torno da qual aquela parte é construída.

2.1 O Rastreador Web

A primeira etapa para a construção dos benchmarks é a obtenção de dados brutos a serem processados. Para tal propósito nós construímos um rastreador Web que realiza mineração de repositórios no GitHub, procurando por códigos fonte em C. A busca feita por esse rastreador retorna projetos ordenados de forma decrescente pela relevância, mensurada pela quantidade de vezes que um projeto foi avaliado como útil por usuários da plataforma GitHub. Para um maior controle dos dados minerados, criamos uma estrutura para cada projeto obtido, contendo a URL original do repositório, a data na qual a mineração foi feita e o número de arquivos C encontrados.

É comum e esperado que sejam utilizados os mais diversos recursos para a construção de sistemas complexos. Esse fator faz com que os principais repositórios contenham vários arquivos diferentes dos arquivos de código pertinentes para a realização desse trabalho. Para contornar esse problema, em todo repositório minerado foi feita uma limpeza dos dados, eliminando assim qualquer tipo de arquivo irrelevante para a construção de benchmarks. Essa limpeza foi aplicada de maneira automática, através de um programa em Shell Script, que foi executado no diretório raiz de cada projeto. O script recursivamente acessa todos os subdiretórios do projeto apagando os arquivos irrelevantes para a compilação de programas C, assim como os diretórios que ficaram vazios após a limpeza. Apenas foram mantidos arquivos de código fonte C e arquivos de cabeçalho. Esses arquivos constituem a entrada para a próxima etapa da construção dos AnghaBench: a extração de funções.

2.2 O Extrator de Funções

Uma vez que coletamos um corpo de arquivos C, a próxima etapa consiste em extrair funções dos mesmos. Para tanto, implementamos o Extrator de Funções. O extrator recebe como entrada um arquivo C, sem que o mesmo tenha sido preprocessado. O extrator então identifica todas as funções contidas no arquivo, que possuam uma definição (isto é, um corpo), e extrai cada uma delas para um arquivo C próprio. Durante a cópia das funções, alguns cuidados são tomados.

```
1  int bs_list_find(const BS_LIST *list, const uint8_t *data)
2  {
3     int r = find(list, data);
4
5     //return only -1 and positive values
6     if (r < 0) {
7        return -1;
8     }
9
10    return list->ids[r];
11 }
```

Figura 2. Exemplo de código de uma função, extraída do repositório toxcore.

Por exemplo, funções que possuem classe de armazenamento estática são anotadas com um atributo "(used)", para evitar que um compilador as otimize (não gere código para as mesmas).

Implementamos o extrator como um *plugin* para Clang, o front-end de compilação C da infraestrutura LLVM. Desta forma, o extrator executa durante a compilação de um arquivo C. Assim que o compilador termina o processo de geração da Árvore de Sintaxe Abstrata (AST) do programa, ela é processada pelo extrator. O extrator percorre a árvore, identificando quaisquer nós que se referem a declarações de funções. Para cada declaração encontrada, o extrator determina se há uma definição disponível para tal função. Caso a encontre, sua implementação é copiada e extraída. É importante notar que um arquivo não precisa ser compilável para que seja processado. Clang gera uma AST para programas mesmo que ocorram erros durante a compilação, como a ausência de definições para nomes. Contanto que Clang gere um nó na AST para uma função do arquivo, o *plugin* é capaz de extraí-la.

Exemplo 2.1. *A Figura 2 mostra um exemplo de código minerado do repositório toxcore, que implementa um protocolo de comunicação segura. Note que a função em questão não é compilável. A estrutura dos tipos "BS_LIST" e "uint8_t" recebidos como parâmetro na linha 1 não é conhecida, assim como a definição da função "find" usada na linha 3.*

Ao final do processamento de um arquivo, o extrator terá gerado vários arquivos, cada um com a implementação de uma função, similar ao código mostrado no Exemplo 2.1. Porém, é improvável que essas funções sejam compiláveis por si só. Sem informações como definições de nomes, tipos, macros e outras funções usadas no código das mesmas, não seria possível gerar código para qualquer implementação não-trivial. Logo, para possibilitar a compilação de códigos na presença de tais restrições, usamos um reconstrutor de tipos –assunto da próxima seção deste trabalho.

```
1  typedef int uint8_t;
2  struct TYPE_4__ { int* ids; };
3  typedef struct TYPE_4__ BS_LIST ;
4  int find (BS_LIST const*, int const*);
5  int bs_list_find(const BS_LIST *list, const uint8_t *data)
6  {
7     int r = find(list, data);
8     //return only -1 and positive values
9     if (r < 0) {
10       return -1;
11    }
12
13    return list->ids[r];
14 }
```

Figura 3. Exemplo de código de função reconstruído, com definições de tipos ausentes inclusas.

2.3 O Reconstrutor de Tipos

Dado o código de uma função extraída, nosso objetivo é garantir que a mesma seja compilável. O código de uma função por si só pode não ser compilável por vários motivos, como a ausência de tipos usados, referências a nomes não definidos no escopo da própria função, etc. Logo, cria-se a necessidade de uma ferramenta que preencha as lacunas no código incompleto. Para tal, usamos Psyche-C [17]. Psyche-C é capaz de inferir tipos e definições para os nomes usados em uma função que não são explicitamente definidos na mesma, gerando um arquivo cabeçalho com tais definições. Anexando a implementação desse cabeçalho ao código original, criamos então um arquivo que codifica um programa compilável. A título de exemplo, Psyche-C já foi utilizada com sucesso na construção de entradas para benchmarks [19].

Exemplo 2.2. *A Figura 3 mostra o código gerado pelo reconstrutor de tipos para a função do Exemplo 2.1. As linhas 1 a 3 fornecem definições compatíveis para os tipos "uint8_t" e "BS_LIST", que eram usados sem ser definidos na função original. A linha 4 fornece uma declaração válida para a função "find". Essa nova versão da função é compilável.*

Psyche-C usa um algoritmo de unificação para encontrar definições válidas para cada tipo ausente, guiado pelos usos encontrados no código. Uma vez que a única informação presente consiste nos usos de cada tipo, não há garantia que o código gerado por Psyche-C seja *semanticamente* equivalente ao código completo. Por exemplo, no código reconstruído do Exemplo 2.2, o reconstrutor gerou um único campo "ids" para a estrutura "BS_LIST", pois esse é o único campo usado dentro da implementação da função. É provável que a definição real do tipo "BS_LIST" possua outros membros. Porém, apenas a definição gerada já é suficiente para tornar a função compilável.

2.4 O Validador de Arquivos

Uma vez que o código incompleto foi reconstruído, passamos então à fase final do processo de geração: a validação do código gerado. Mesmo que Psyche-C tenha gerado definições para os tipos ausentes, é necessário garantir que o programa final é de fato compilável. Para tanto, usamos o próprio Clang como validador. Simplesmente passamos o arquivo final como entrada para o compilador, para que o mesmo gere uma representação intermediária em LLVM para o mesmo. Caso a compilação ocorra com sucesso, consideramos o programa válido. Salientamos que os programas gerados são apenas *compiláveis*. Isto é, não geramos executáveis para os programas. Embora os programas possuam definições para os tipos ausentes, elementos como implementações de funções chamadas de dentro do código não são gerados, e portanto a *linkedição* dos mesmos não é possível.

Para a grande maioria dos programas, se Psyche-C é capaz de reconstruir os tipos incompletos, o código gerado tende a representar programas válidos. Para alguns casos, porém, a compilação ainda pode falhar. Em alguns desses casos, tal situação ocorre devido a pequenos bugs na própria implementação de Psyche-C. Tais casos foram reportados aos desenvolvedores da ferramenta. Em outros casos, a compilação falha devido ao uso de funcionalidades *builtin* exclusivas de outros compiladores, como gcc. Vale ressaltar que poderíamos usar qualquer compilador C como validador. A escolha por Clang foi simplesmente pela facilidade da realização de análises estáticas na infraestrutura LLVM.

3 Aplicações

Detalhes de Implementação Angha foi produzido a partir da combinação de diferentes ferramentas. O rastreador Web descrito na Seção 2.1 foi implementado como uma aplicação em Java versão 8, e utilizando as API's para mineração de repositórios Jcabi-github[1] e JGit[2]. O extrator de funções descrito na Seção 2.2 foi desenvolvido como um *plugin* para Clang 8.0. Para o reconstrutor descrito na Seção 2.3, usamos a versão mais recente de Cnippet[3], que é equivalente à versão publicamente disponível de Psyche-C[4]. Para a validação dos programas reconstruídos e geração dos benchmarks, foram usados Clang 8.0 e LLVM 8.0.

3.1 Visão Geral dos Dados

Para demonstrar sua efetividade, usamos Angha para gerar funções compiláveis a partir de 79 dos mais populares repositórios de código C na plataforma GitHub. Entre estes repositórios encontram-se, por exemplo, o sistema de controle de versões *git*[5], o núcleo do sistema operacional *linux*[6] e a biblioteca de processamento multimídia *FFmpeg* [7].

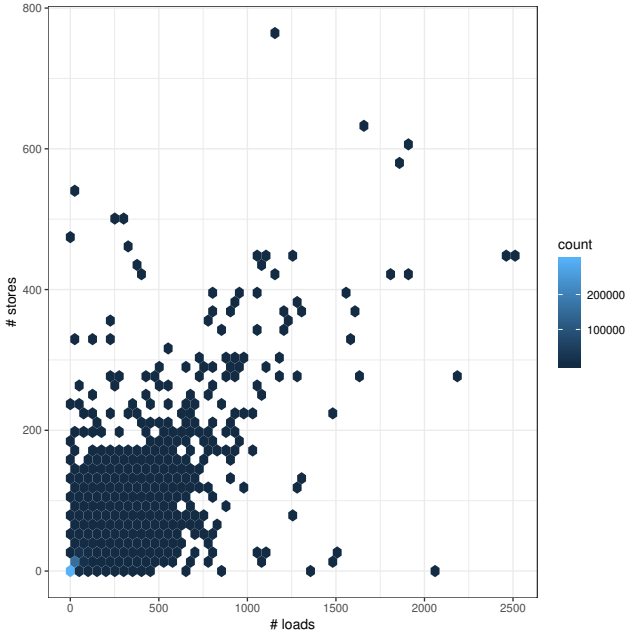

Figura 4. Distribuição de programas por número de instruções *load* e *store* em sua representação intermediária LLVM.

No total, coletamos 54.431 arquivos de código fonte C. Dos mesmos, extraímos 698.449 funções. O tamanho de tais funções varia entre apenas uma linha de código até a maior função com 45.263 linhas, provinda do projeto do desmontador Radare2. Executamos então o reconstrutor de tipos, com tempo limite de 30 segundos para cada função. O mesmo foi capaz de gerar código compilável para 529.498 funções, uma taxa de sucesso de aproximadamente 75,8%. O reconstrutor de tipos falha, ou devido a *timeouts*, ou devido à existência de macros na função alvo que não são sintaticamente válidas em C. Logo, obtivemos um conjunto de dados para modelagem de aproximadamente 530 mil programas.

A fim de estudar a estrutura do conjunto de dados formado por tais programas, medimos a quantidade de instruções LLVM contidas na representação intermediária de cada um deles. A Figura 4 mostra a distribuição dos programas em função do número de instruções de leitura e escrita em memória (*loads* e *stores*) geradas para cada um. É fácil observar pela figura que a grande maioria dos programas possui poucas instruções destes tipos. De fato, os números médios de *loads* e *stores* na população são de 11 e 5 instruções, respectivamente. Os valores medianos para ambos são de 22,1 e 8,1 instruções.

[1] https://github.com/jcabi/jcabi-github
[2] https://www.eclipse.org/jgit/
[3] http://cnippet.cc/
[4] https://github.com/ltcmelo/psychec
[5] https://github.com/git
[6] https://github.com/torvalds/linux
[7] https://github.com/FFmpeg/FFmpeg

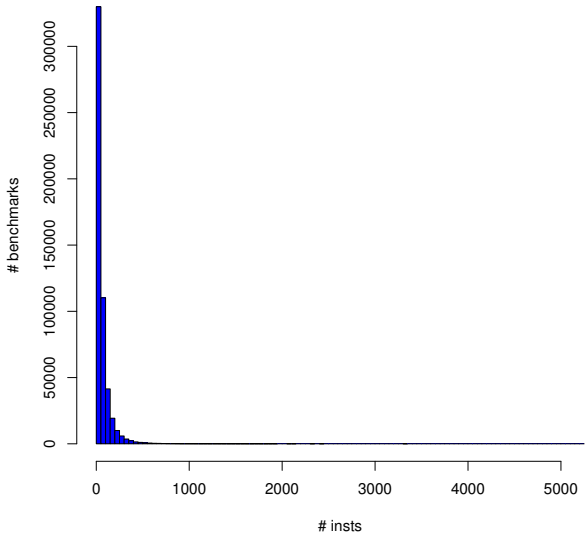

Figura 5. Histograma da população de 530 mil programas, por tamanho em número de instruções LLVM.

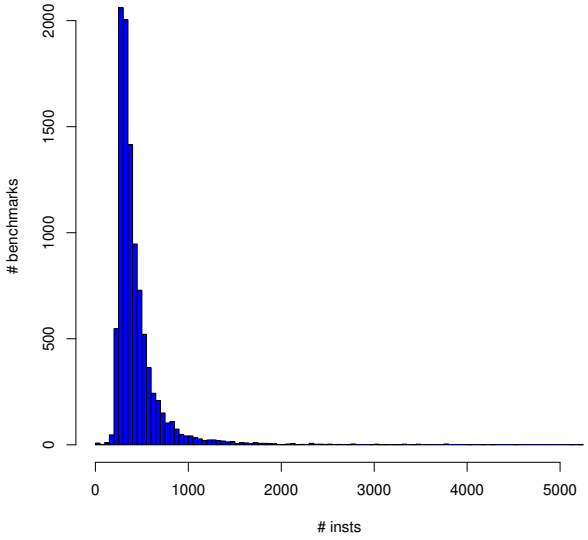

Figura 6. Histograma da população de 10 mil programas, por tamanho em número de instruções LLVM.

Para representar o tamanho de cada programa, medimos seu número total de instruções LLVM (de qualquer tipo). O histograma da Figura 5 mostra a frequência de programas para cada tamanho, em número de instruções. Nesta figura também é possível observar que a população concentra-se fortemente em programas de tamanho pequeno. Aproximadamente 60% dos programas possuem 50 ou menos instruções LLVM. O número médio é de 63,27 instruções, e o valor mediano é de 36 instruções.

Como observamos que a população de programas é fortemente enviesada, decidimos criar um novo conjunto de dados. Selecionamos, entre os 530 mil programas iniciais, os 10.000 maiores, em número de instruções. O histograma da Figura 6 mostra a frequência de programas para cada tamanho nesse novo conjunto. Embora a nova população ainda seja relativamente concentrada em programas menores, a distribuição é mais equilibrada. A média do número de instruções nesses programas é de 441,8, enquanto o valor mediano é de 360 instruções. Devido à sua disposição mais heterogênea, consideramos apenas esse novo conjunto para a realização dos experimentos descritos nas próximas seções.

3.2 Análise da Efetividade de Otimizações

Uma otimização de código busca melhorar algum parâmetro de eficiência de um programa: seu tempo de execução, seu consumo de energia, ou a quantidade de espaço que ele ocupa, por exemplo. Uma otimização é dita efetiva quando ela consegue, consistentemente, melhorar algum desses aspectos. Benchmarks permitem medir a efetividade de otimizações de código.

Objetivo do Experimento O objetivo deste experimento é verificar como diferentes níveis de otimizações de código permitem reduzir o número de instruções presentes no código objeto de um programa. Neste experimento iremos comparar os três conjuntos de teste: AnghaBench, LLVMBench e CSmithBench, verificando qual a redução média de instruções que LLVM -O1 e -O3 provoca em cada um deles. Consideraremos como *verdade hipotética* o efeito de LLVM sobre LLVMBench, e buscaremos mostrar que AnghaBench aproxima melhor o comportamento da verdade hipotética do que os programas produzidos por CSmith.

Discussão de resultados A figura 7 mostra o efeito das otimizações em cada um dos conjuntos de benchmarks que testamos. Em cada figura pode ser vista a diagonal principal (linha negra) e a linha de regressão (vermelha). Pontos abaixo da diagonal principal representam benchmarks que tiveram o número de instruções reduzido pela otimização. Conforme é possível ver na figura, tal foi o caso na maioria dos benchmarks. Em algumas situações, otimizações como integração de funções ou desenlaço levaram ao crescimento de código. Tal fato é mais comum no nível 3 de otimização, quando a integração passa a poder acontecer.

Principais Conclusões A principal conclusão deste experimento é que AnghaBench aproxima muito melhor o comportamento de programas reais do que CSmithBench. Tal

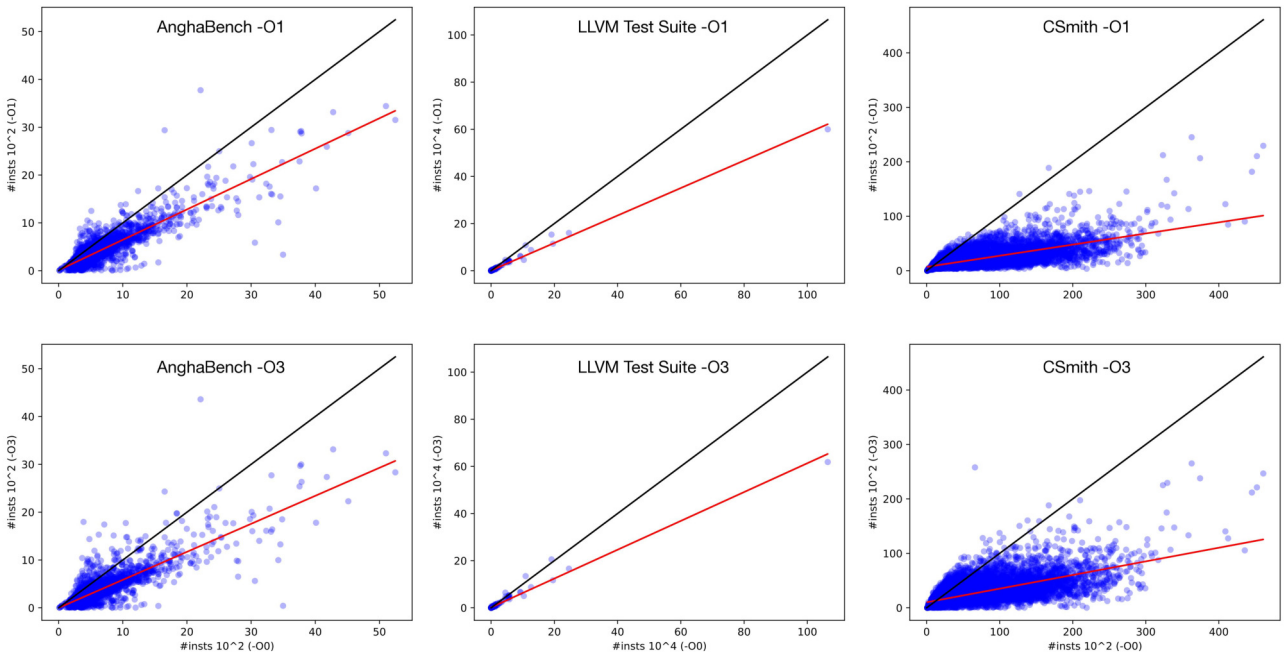

Figura 7. Número de instruções de programas compilados com LLVM -O0 e -O3. Pontos abaixo da linha representam programas que sofreram redução de instruções tendo sido compilados com LLVM -O3.

fato pode ser constatado a partir de observações visuais: as linhas de regressão produzidas pelos conjuntos AnghaBench e LLVMBench são muito similares na figura 7. É possível mensurar esta proximidade. No nível O1 de otimizaç~oes, o *erro médio quadrático* de um preditor treinado usando AnghaBench, quando aplicado sobre LLVMBench, é 1.6x maior do que se LLVMBench fosse usado para treinar predições nele mesmo. Por outro lado, caso CSmithBench fosse usado como o conjunto de treinamento, então o erro quadrático seria 33.3 vezes maior! No nível O3 a diferença é menor, porém não trivial: AnghaBench leva a predições 10.1x mais precisas do que CSmithBench, quando usados para prever o comportamento de otimizações sobre LLVMBench. Tal diferença decorre do fato das entradas dos programas gerados por CSmith serem constantes. Uma vez que este padrão é fácil de otimizar, LLVM consegue reduzir muitas instruções dos programas aleatórios –algo que não se dá sobre os benchmarks que mineramos de repositórios.

3.3 Análise de Complexidade de Otimizações

Benchmarks permitem a inferência empírica da complexidade assimptótica de otimizações e análises estáticas. Essa análise experimental é útil para detectar bugs de desempenho em compiladores e ferramentas similares que processam programas. Nesta seção, mostraremos como AnghaBench suporta esse tipo de estudo empírico.

Objetivo do Experimento O objetivo desse experimento é analisar o comportamento assimptótico de otimizações em código fonte. Nós executamos duas transformações de código em nosso conjunto de benchmarks: *global value numbering* (GVN) e conversão para o formato de atribuição estática única (SSA). Ambas as transformações são distribuídos juntos com a infra-estrutura de compilação LLVM, e são implementadas a partir de algoritmos clássicos da literatura de compiladores. A implementação de GVN segue o algoritmo proposto por Briggs *et al.* [7]. A criação do formato SSA envolve três algoritmos: a construção da árvore de dominância, a criação de funções ϕ, e a propagação esparsa de constantes. A construção da árvore de dominância é implementada a partir do algoritmo de Lengauer & Tarjan [15]. A criação de funções ϕ segue a técnica proposta por Sreedhar & Gao [20]. Finalmente, a propagação de constantes é feita via o algoritmo de Wegman & Zadeck [22]. Todas essas implementações são supostamente lineares no tamanho dos programas. Nesta seção testaremos tal hipótese.

Discussão de resultados A figura 8 mostra os resultados de tempo de execução das duas otimizações nos 10,000 maiores benchmarks de AnghaBench. O tamanho dos benchmarks é medido pelo número de instruções na representação intermediária de LLVM, sem a aplicação de qualquer otimização (LLVM -O0). Tempo é medido em milisegundos. Cada ponto no gráfico representa os tempos de execução da conversão para o formato SSA (eixo z) e de GVN (eixo y). Uma

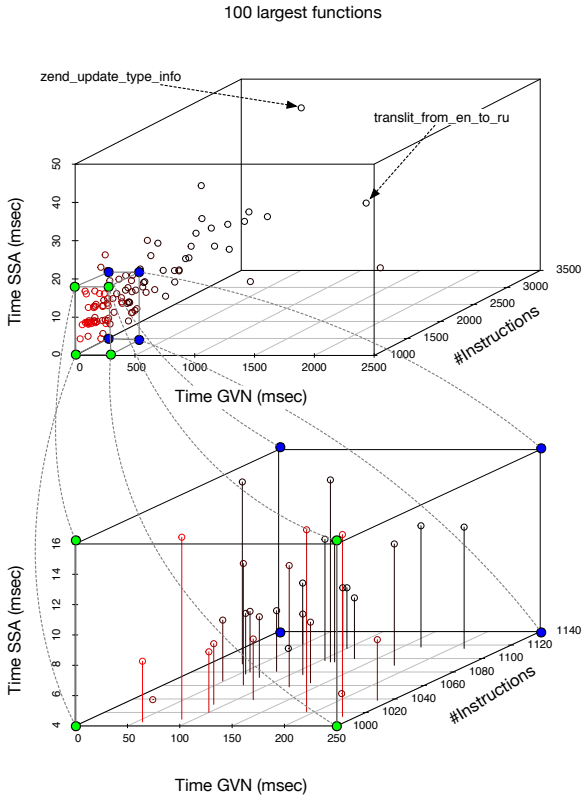

Figura 8. Avaliação de complexidade assimptótica de duas transformações de código: *global value numbering* (GVN) e conversão para o formato de atribuição estática única (SSA).

parte da figura 8 mostra um zoom das menores funções que analisamos.

Principais Conclusões Esse experimento nos permitiu averiguar que o tempo gasto em ambas as otimizações aumenta de forma linear com o tamanho dos benchmarks. Como consequência, mesmo em nossos maiores benchmarks as otimizações utilizadas são eficientes, executando em milisegundos. O coeficiente de determinação (R^2) entre GVN e o tamanho de programas é 0.43, enquanto que essa mesma relação entre SSA e o tamanho dos programas é 0.27. Em nenhum dos casos, temos valores muito próximos de 1.0, ainda que os coeficientes sejam positivos. Assim, existe uma correlação entre o tamanho dos programas e o tempo para otimizá-los –fato já esperado. Porém, não temos forte evidência para concluir que essa relação seja linear, pelo menos dados os benchmarks que usamos para inferi-la.

4 Trabalhos Relacionados

A falta de benchmarks tem sido percebida como um problema sério para o treinamento de compiladores. Esse problema leva projetistas de compiladores a desenvolverem micro-benchmarks que exercitam características particulares da arquitetura alvo da compilação. Exemplos desse *modus-operandi* incluem a implementação de CHOMP, um arcabouço para mapear programas para diferentes configurações de hardware [21], e o escalonador caixa-preta de Barik *et al* [6]. Outros exemplos podem ser vistos em trabalhos bem recentes, como as análises realizadas por Jia *et al.* [13] e Arafa *et al.* [1]. Nesse caso, micro-benchmarks são usados para descobrir características específicas de GPUs, como a latência de instruções. Em todos esses trabalhos, geradores de micro-benchmarks foram usados para produzir milhares de programas, os quais foram, posteriormente, usados para apurar os efeitos de diferentes otimizações de código. Tais micro-benchmarks são muito parecidos entre si –eles diferem somente em algumas poucas constantes usadas, por exemplo, para definir o número de interações de laços. Embora úteis no contexto em que foram usados, não é claro como essas técnicas poderiam ser estendidas para produzir benchmarks mais gerais.

Nessa direção –a produção de benchmarks que aproximem o comportamento de programas em geral– também muito foi feito. Os primeiros trabalhos na área tinham como foco principal encontrar bugs em compiladores. Tais bugs podem manifestar-se durante a execução do código binário produzido [5, 23], ou durante o próprio processo de geração de código [3, 5]. Nesse último caso, o defeito leva à terminação anormal da compilação, isto é, sem a geração de código. O primeiro trabalho nessa linha, do qual temos notícia, deve-se a Hanford [12] e data de 1970. Hanford produzia programas em PL/I. Embora fosse relativamente trivial reconhecer o caráter aleatório daqueles programas, eles já possuíam algumas características de códigos reais. Por exemplo, Hanford preocupou-se em assegurar que variáveis eram declaradas antes de serem usadas. Décadas mais tarde foram lançados geradores de programas em Fortran [8] e C [16]. Em 2011, pesquisadores da Universidade de Utah lançaram CSmith [23], até a presente data, o gerador de programas aleatórios mais popular entre usuários da linguagem C.

Nenhum dos trabalhos acima citados tinha como foco treinar técnicas de auto-adaptação. Ainda assim, CSmith tem sido utilizado com tal propósito, como foi salientado por Cummins *et al.* [10]. Entretanto, com a crescente adoção de técnicas de aprendizagem automática como uma forma de melhorar o processo de geração de código, geradores de benchmarks mais reais passaram a ser um importante objetivo de pesquisa dentro da comunidade de linguagens de programação. Nessa linha de trabalho, temos CLgen, uma ferramenta produzida por Cummins *et al.* para gerar programas com diretivas OpenCL [10]. Baseada em técnicas similares, Han *et al.* propuseram uma técnica para gerar benchmarks voltados para o uso de memória em sistemas formados por CPUs e GPUs [11]. Tanto o trabalho de Cummins *et al.* quanto o trabalho de Han *et al.* são específicos para arquiteturas formadas por CPUs e GPUs. Uma vez que os programas que ambos os grupos representam *kernels* de

alto desempenho, dificilmente eles poderiam ser usados em contextos mais gerais, por exemplo, para prever o efeito das diferentes otimizações utilizadas por um compilador.

Salientamos que as técnicas de Cummins e Han geram programas a partir de um modelo probabilístico. Tal modelo foi construído a partir de programas extraídos de repositórios, mas os programas criados nunca chegaram a existir como partes de aplicações. Nossa técnica transforma programas reais em benchmarks: nós mineramos tais programs diretamente a partir de repositórios públicos. Abordagem similar foi utilizada por Richards *et al.* [18] para produzir benchmarks em JavaScript. Porém, o desafio enfrentado por Richards *et al.* é bem mais simples que o nosso: uma função JavaScript pode ser interpretada sem suas dependências: erros em tempo de execução são capturados pelo interpretador, e o programa nunca fica em um estado indefinido. Em nosso caso, precisamos utilizar um reconstrutor de tipos para permitir que nossos benchmarks possam ser –estaticamente– compilados.

5 Conclusão

Este artigo descreveu uma metodologia para a construção de conjuntos de benchmarks em C, e a implementação dessa metodologia, a ferramenta Angha. A partir de Angha, fomos capazes de criar AnghaBench, um conjunto com 530 mil funções compiláveis. Esse conjunto pode ser facilmente aumentado com novas amostras, pois o processo de construção de benchmarks é automático. Conforme descrito neste trabalho, benchmarks são minerados a partir de repositórios públicos, e pré-processados com um inferidor de tipos. Esse pré-processamento nos permite compilar cada amostra presente em AnghaBench. Nosso conjunto de benchmarks nos permitiu observar diversas características de programas reais, tais como o número de instruções que eles contêm. Além disso, pudemos demonstrar que AnghaBench é mais efetivo em prever o efeito de otimizações de código do que benchmarks gerados por CSmith, por exemplo.

Referências

[1] Yehia Arafa, Abdel-Hameed A. Badawy, Gopinath Chennupati, Nandakishore Santhi, and Stephan Eidenbenz. 2019. Instructions' Latencies Characterization for NVIDIA GPGPUs. *CoRR* abs/1905.08778 (2019), 65.

[2] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A Survey on Compiler Autotuning Using Machine Learning. *Comput. Surv.* 51, 5 (2018), 96:1–96:42.

[3] Antoine Balestrat. 2016. CCG: A Random C code generator. https://github.com/Mrktn/ccg

[4] Gergö Barany. 2017. Liveness-Driven Random Program Generation. In *LOPSTR*. CoRR, arxiv, 1–15.

[5] Gergö Barany. 2018. Finding Missed Compiler Optimizations by Differential Testing. In *CC*. ACM, New York, NY, USA, 82–92.

[6] Rajkishore Barik, Naila Farooqui, Brian T. Lewis, Chunling Hu, and Tatiana Shpeisman. 2016. A Black-box Approach to Energy-aware Scheduling on Integrated CPU-GPU Systems. In *CGO*. ACM, New York, NY, USA, 70–81.

[7] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. 1997. Value Numbering. *Softw. Pract. Exper.* 27, 6 (1997), 701–724.

[8] C. J. Burgess and M. Saidi. 1996. The automatic generation of test cases for optimizing Fortran compilers. *Information & Software Technology* 38, 2 (1996), 111–119.

[9] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler Fuzzing Through Deep Learning. In *ISSTA*. ACM, New York, NY, USA, 95–105.

[10] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Synthesizing Benchmarks for Predictive Modeling. In *CGO*. IEEE Press, Piscataway, NJ, USA, 86–99.

[11] Tianyi David Han and Tarek S. Abdelrahman. 2017. Use of Synthetic Benchmarks for Machine-Learning-Based Performance Auto-Tuning. In *IPDPS Workshops*. IEEE Press, Piscataway, NJ, USA, 1350–1361.

[12] K. V. Hanford. 1970. Automatic Generation of Test Cases. *IBM Syst. J.* 9, 4 (1970), 242–257.

[13] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. 2019. Dissecting the NVidia Turing T4 GPU via Microbenchmarking. *CoRR* abs/1903.07486 (2019), 9.

[14] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. IEEE Computer Society, Washington, DC, USA, 75–.

[15] Thomas Lengauer and Robert Endre Tarjan. 1979. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Trans. Program. Lang. Syst.* 1, 1 (1979), 121–141.

[16] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[17] Leandro T. C. Melo, Rodrigo G. Ribeiro, Marcus R. de Araújo, and Fernando Magno Quintão Pereira. 2017. Inference of Static Semantics for Incomplete C Programs. *Proc. ACM Program. Lang.* 2, POPL (2017), 29:1–29:28.

[18] Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. 2011. Automated Construction of JavaScript Benchmarks. In *OOPSLA*. ACM, New York, NY, USA, 677–694.

[19] Marcus Rodrigues, Breno Guimarães, and Fernando Magno Quintão Pereira. 2019. Generation of In-bounds Inputs for Arrays in Memory-unsafe Languages. In *CGO*. IEEE Press, Piscataway, NJ, USA, 136–148.

[20] Vugranam C. Sreedhar and Guang R. Gao. 1995. A Linear Time Algorithm for Placing phi-nodes. In *POPL*. ACM, New York, NY, USA, 62–73.

[21] Jyothi Krishna Viswakaran Sreelatha, Shankar Balachandran, and Rupesh Nasre. 2018. CHOAMP: Cost Based Hardware Optimization for Asymmetric Multicore Processors. *Trans. Multi-Scale Computing Systems* 4, 2 (2018), 163–176.

[22] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (1991), 181–210.

[23] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *PLDI*. ACM, New York, NY, USA, 283–294.