Predição de Tamanho de Código utilizando Benchmarks Sintetizados: Um Estudo de Caso

André Felipe Zanella[1], **Anderson Faustino da Silva**[1]

[1]DIN – Universidade Estadual de Maringá (UEM) – Maringá – PR – Brasil

aft.zanella@gmail.com, anderson@din.uem.br

Abstract. *One of the main issues preventing Deep Neural Networks from becoming mainstream on compiler optimization is the difficulty of creating high quality training datasets. Real world programs are usually large and complex, and due the lack of benchmarks, predictive compilation models usually fails to generalize due the vast space of programs. Recent efforts from the community to mitigate this problem have resulted on open-source benchmark synthesis tools capable of generating thousands or millions of synthetic benchmarks. However, these code hardly match in terms of similarity with hand coded benchmarks. This paper aims to evaluate the performance of AnghaBench benchmarks, a prominent suit of synthetic C files for predictive compilation, in a machine learning task. We compare the performance of these benchmarks against applications from the LLVM test suite, using a LSTM model to predict the code size of a program provided by a optimization sequence without the need to compile it. The experimental results indicates that, in some domains, the synthetic benchmarks performs equally like the real applications, but does not outperforms it.*

Resumo. *Um dos principais problemas que impedem Redes Neurais Profundas se tornarem predominantes para otimização de compiladores é a dificuldade de criação de conjuntos de dados de alta qualidade. Benchmarks reais geralmente são programas grandes e complexos, e por não serem abundantes, modelos de compilação preditiva geralmente apresentam dificuldades de generalização devido ao vasto espaço de programas. Esforços recentes da comunidade para mitigar este problema resultaram em ferramentas para síntese automática de benchmarks, capazes de gerar milhares ou milhões de programas sintéticos. No entanto, esses códigos dificilmente coincidem em termos de similaridade com benchmarks codificados manualmente. Este artigo tem como objetivo avaliar o desempenho dos benchmarks AnghaBench, uma proeminente suíte de programas C sintetizados para compilação preditiva, em um problema de aprendizagem de máquina. Comparamos o desempenho desses benchmarks com aplicações provenientes da suíte de testes LLVM, utilizando um modelo LSTM para prever o tamanho de código proporcionado por uma sequência de otimizações, sem a necessidade de compilar o programa. Os resultados experimentais indicam que, em alguns domínios, benchmarks sintéticos são equiparáveis, mas não superam aplicações reais.*

1. Introdução

A última década foi agraciada com grandes avanços envolvendo aplicações de Redes Neurais Artificiais (do inglês *Artificial Neural Network* ou `ANNs`) para mitigação de diversos

problemas, principalmente graças ao suporte de aceleradores de *hardware* especializados. O método de aprendizagem tem sido a abordagem padrão em diversas tarefas, incluindo, mas não limitado à, Análise de Imagens [Wei et al. 2019], Processamento de Linguagem Natural [Otter et al. 2021] e Sistemas de Recomendação [Zhang et al. 2019]. `ANNs` e outros métodos de aprendizagem de máquina tem sido adotados pela comunidade de compiladores, em geral, para encontrar boas sequências de otimização [1] que superem planos pré-definidos por compiladores como `GCC` [2] e `LLVM` [Lattner and Adve 2004].

Apesar da adoção pela comunidade, a pesquisa em otimização de compiladores auxiliada por `ANNs` ainda é modesta quando comparada as áreas de pesquisa já citadas. Um grande limitante para uma adoção mais ampla é a dificuldade de construção de conjuntos de dados de alta qualidade, *i.e.*, abrangentes conjuntos de programas para treinamento que englobem significativamente as características de diversos domínios de aplicações. Tal fato ocorre devido a escassez de *benchmarks* disponíveis e a grande demanda por dados exigida por experimentos envolvendo `ANNs` [Cummins et al. 2017].

Síntese de *benchmarks* é uma abordagem promissora para mitigar o problema da escassez de dados. Visando compilação preditiva, o método consiste em gerar de forma algorítmica, grandes quantidades de *benchmarks* (milhares ou milhões) que encapsulem diversas propriedades semânticas e se assemelhem a *benchmarks* reais [Cummins et al. 2017]. No entanto, gerar código sintético que seja similar a programas reais é uma tarefa difícil. Diversos problemas devem ser mitigados dependendo da abordagem de geração, como por exemplo, inferência de tipos [da Silva et al. 2021] e treinamento de `ANNs` [Cummins et al. 2018]. Consequentemente, estes programas podem não corresponder fielmente a uma carga de trabalho convencional.

Um promissor conjunto de *benchmarks* sintéticos com mais de 1 milhão de amostras foi recentemente disponibilizado por Silva *et al* [da Silva et al. 2021]. O conjunto denominado `AnghaBench`, foi gerado a partir de um *framework* composto por um mecanismo de mineração de código em repositórios de código aberto na plataforma `GitHub` [3], um extrator de funções de aplicações, e um motor de inferência de tipos. Os *benchmarks* gerados são arquivos C compiláveis, *i.e.*, passíveis de geração de código objeto. Portanto, são programas direcionados para o estudo de propriedades estáticas e otimização de tamanho de código.

Os autores avaliam seus *benchmarks* sintéticos em comparação com outros geradores automáticos de programas, como `DeepSmith` [Cummins et al. 2018] e `LDRGen` [Barany 2018b] em termos de similaridade com programas reais e, geração de sequências de otimização para redução de código. Em contraste, este trabalho busca avaliar o desempenho de programas `AnghaBench` em um sistema de compilação preditiva baseado em Redes Neurais Recorrentes (`LSTM`) [Hochreiter and Schmidhuber 1997]. A heurística de compilação consiste em determinar a quantidade de instruções de uma representação intermediária (IR) `LLVM` não otimizada, quando uma sequência de passos de otimização é aplicada. Avaliamos a acurácia do modelo em diferentes domínios de programas e comparamos os resultados obtidos com *benchmarks* reais. Nossos experimen-

[1] Um plano, ou sequência de otimização, é um conjunto de passos de transformações aplicados pelo compilador durante o processo de geração de código final.

[2] https://gcc.gnu.org/

[3] https://github.com/

tos indicam que o modelo treinado com programas reais detêm os melhores resultados. Providenciamos uma discussão com relação as possíveis causas, em termos da modelagem do problema, heurística de compilação e conjuntos de dados.

2. Trabalhos Relacionados

Atualmente, a comunidade de Linguagens de Programação e Compiladores dispõem de algumas ferramentas geradoras de *benchmarks*. Alguns exemplos são os geradores de *kernels* OpenCL, tais como `CLgen` [Cummins et al. 2017] e `DeepSmith` [Cummins et al. 2018], e de programas C, `LDRGen` [Barany 2018b], `CSmith` [Yang et al. 2011] e `AnghaBench` [da Silva et al. 2021].

Código sintético já foi experimentado como base para compilação preditiva em trabalhos anteriores [Barany 2018a] [Hashimoto and Ishiura 2016] [Huang et al. 2020], no entanto, estes não levantaram a questão comparativa da performance com relação a outros conjuntos. Um trabalho similar em natureza a nossa proposta é realizado por Goens *et al.* [Goens et al. 2019]. Os autores propõem um estudo de caso em que avaliam artefatos gerados pela ferramenta `CLgen` em comparação aos *benchmakrs* utilizados como base de dados pela ferramenta. O problema abordado em questão é o mapeamento de dispositivos (`GPU/CPU`) utilizando aprendizagem de máquina. Similarmente à `AnghaBench`, `CLGen` utiliza uma base de programas coletados de repositórios de código aberto, porém, diverge na metodologia generativa, fazendo uso de redes `LSTM`. Os resultados indicam que as aplicações geradas não apresentam performance positiva em relação a aplicações reais.

Os autores da suíte `AnghaBench` avaliam os artefatos produzidos no tocante a: (1) similaridade com *benchmarks* reais; (2) predição do impacto de sequências de otimização; e (3) base de treinamento para Aprendizagem de Máquina. Em todas as questões `AnghaBench` é avaliado com relação à outras coleções de programas sintéticos, apresentando vantagens sobre estes. Em (3), os autores avaliam a suíte em relação a suíte de testes `LLVM` em uma heurística de criação de sequências para redução de código. Nossa proposta difere, pois, realizamos testes em *benchmarks* reais de diferentes domínios em um sistema baseado em `LSTM`.

Outra questão pertinente a este trabalho é a geração de código reduzido. Como aponta Ashouri *et al.* [Ashouri et al. 2018], o problema de predição de tamanho de código passou a ser uma preocupação secundária com o avanço de sistemas de armazenamento. No entanto, pesquisas recentes trazem a tona esta questão novamente. Trofin *et al.* [Trofin et al. 2021] adotam Aprendizagem por Reforço para decidir quando se fazer *inline* em uma chamada, alcançando códigos até 7% menores em comparação à `clang -Oz`. Heo *et al.* [Heo et al. 2018] também fazem uso do mesmo paradigma em combinação com técnicas de depuração para reduzir códigos intermediários `LLVM`. Silva *et al.* [Silva et al. 2021] realizaram um estudo sistemático sobre o espaço de transformações para redução de código e propõem uma metodologia para criação de sequências especializadas para tal tarefa. A metodologia proposta pelos autores foi utilizada no presente trabalho para criação de sequências de transformações. Ortogonalmente, novas transformações para redução de código foram pospostas por Rocha *et al.* [Rocha et al. 2019, Rocha et al. 2020].

3. Descrição do Problema e Objetivos

Encontrar bons planos de compilação, isto é, sequências de transformações que providenciem melhorias no código final é uma tarefa difícil devido ao vasto espaço de exploração. A busca exaustiva por um plano de compilação é muitas vezes inviável, pois, é necessário a compilação e, dependendo do objetivo, a execução da aplicação para avaliação de sua performance. Devido a isso, é desejável, em muitos casos, realizar compilação preditiva, a fim de se conhecer de antemão o desempenho que uma sequência pode proporcionar sem a necessidade da compilação e execução da aplicação propriamente dita [Ashouri et al. 2018].

Neste trabalho, modelamos o problema de predição de tamanho de código como um problema de regressão. A métrica utilizada para medição é o número de instruções do código intermediário `LLVM`. Dado um programa IR não otimizado, denotado P_{noopt}, e uma sequência de transformações S, desejamos encontrar uma função de predição R, tal que $R(P_{noopt}, S)$ forneça com alta probabilidade a estimativa do número de instruções `LLVM` do programa P_S, isto é, o programa P_{noopt} compilado com a sequência S. Para escolha do modelo de aprendizagem profunda, partimos da premissa de que linguagens de programação devem ser abordadas com os mesmos princípios e técnicas utilizados para Processamento de Linguagem Natural. Para atender tal requisito, utilizamos Redes Neurais Recorrentes LSTM em camadas e vetores densos de alta dimensão, também chamados de *embeddings*, para codificação numérica das aplicações, onde cada instrução `LLVM` é um vetor denso de 200 elementos.

Com o problema de compilação e modelo de Rede Neural definidos, a tarefa passa a ser avaliar a performance dos *benchmarks* sob estas condições. Avaliamos duas classes de *benchmarks* `AnghaBench`, a primeira contém arquivos com uma única função e a segunda contém arquivos com múltiplas definições de funções. Como base comparativa, treinamos nosso modelo com *benchmakrs* reais coletados da suíte de testes da infraestrutura `LLVM`. As predições foram realizadas em aplicações novas, dentre estas, foram selecionados *benchmarks* desenvolvidos para avaliação de sistemas embarcados. A justificativa para tal escolha está no fato de dispositivos embarcados serem os que mais se beneficiam de código pequeno. Também construímos um conjunto de dados para testes formado exclusivamente por arquivos contendo uma única função. O principal objetivo deste conjunto é avaliar a classe de dados `AnghaBench` com uma única função. Por fim, realizamos a avaliação em uma partição de programas previamente selecionados da suíte `LLVM`.

4. Materiais e Métodos

Como discutido nas seções anteriores, o principal objetivo deste trabalho é avaliar conjuntos de *benchmarks* sintéticos no problema de compilação preditiva. Para tal, definimos diferentes conjuntos de dados, uma representação de programas, construímos sequências de otimização e uma heurística de compilação.

Conjuntos de Dados e Estatísticas. Os dados que compõem os conjuntos de treino e teste foram coletados da suíte `AnghaBench`, da suíte de teste da infraestrutura `LLVM` e dos *benchmarks* `Mibench` [4] e `Coremark-Pro` [5]. Separamos os conjuntos de dados em classes, as quais passaremos a nos referir com as seguintes nomenclaturas:

[4] http://vhosts.eecs.umich.edu/mibench//

[5] https://www.eembc.org/coremark-pro/

- **Angha15**: Dentre os arquivos de funções `AnghaBench` encontramos as 15.000 maiores funções e selecionamos aleatoriamente 2.000 funções para compor o conjunto de dados.
- **AnghaW**: `AnghaBench` também contém programas completos, com aproximadamente 15.000 amostras. Seguimos a mesma metodologia e selecionamos aleatoriamente 2.000 amostras para este conjunto.
- **LLVM Suíte**: Classe composta por 218 aplicações de diversas conjuntos de *benchmarks*. Dentre estes, particionamos de forma aleatória 178 aplicações para treino e 40 aplicações para teste.
- **Embarcados**: Esta classe compreende as suítes `Coremark-Pro` contendo 9 aplicações e `Mibench` com 19 aplicações.
- **Funções**: Extraímos funções individuais de cada suíte da classe **Embarcados** e selecionamos os 300 maiores arquivos (nr. de instruções `LLVM`) de cada um quando compilados com `opt -O0`. Denotaremos os conjuntos da classe como `Coremark-f` e `Mibench-f`.

Dentre as classes definidas, utilizamos para testes os programas **Embarcados**, **Funções** e uma partição da classe **LLVM Suíte**. Adicionalmente, utilizamos para treinamento as combinações `Angha15+LLVM` e `AnghaW+LLVM`. As Tabelas 1 e 2 sumarizam as estatísticas com relação a quantidade de instruções dos conjuntos.

Tabela 1. Estatísticas dos conjuntos de dados utilizados para treinamento e teste.

	Angha15			**AnghaW**			**LLVM Suíte (Treino)**		
Instruções	Min	Média	Max	Min	Média	Max	Min	Média	Max
`opt -O0`	256	397,12	1735	1	267,10	4015	42	2590,26	173943
`opt -Os`	2	224,25	1021	1	143,51	2882	23	1516,61	82968
`opt -Oz`	2	216,97	980	1	137,11	2621	22	1306,25	81426
`Best_10`	2	220,42	1017	1	137,65	2542	21	1290,92	86584
	Funções-Coremark			**Funções-Mibench**			**LLVM Suíte (Teste)**		
`opt -O0`	48	216,41	6147	428	1390,26	21525	43	2064,32	12935
`opt -Os`	17	141,21	3407	152	697,17	7843	15	1370,03	8430
`opt -Oz`	17	117,88	3407	133	689,63	7879	18	1053,32	6803
`Best_10`	16	112,60	3415	144	709,92	10232	15	1032,34	7133

Tabela 2. Estatísticas dos conjuntos de dados da classe Embarcados.

	Coremark-Pro			**Mibench**		
Instruções	Min	Média	Max	Min	Média	Max
`opt -O0`	1488	7688	23334	93	23008	276055
`opt -Os`	842	4912	14279	39	12805	114317
`opt -Oz`	709	4102	12222	39	11822	113096
`Best_10`	696	3966	12789	39	11993	141879

Módulo Extrator de Funções. A fim de se construir arquivos compiláveis contendo funções individuais de uma aplicação, utilizamos a ferramenta `llvm-extract` da infraestrutura `LLVM`. Esta ferramenta tem como argumentos um programa em codificação IR e o nome da função que se deseja coletar. Tendo em vista que algumas aplicações possuem dezenas de definições de funções, como o caso da aplicação `Mibench-cjpg`, implementamos um passe `LLVM` para coleta de nomes e automatização do processo. A Figura 1 descreve a configuração utilizada para construção dos dados da classe **Funções**.

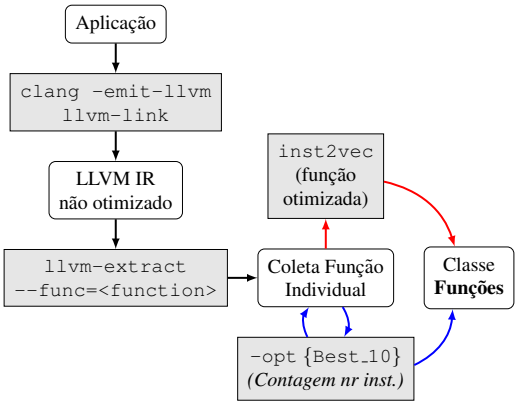


Figura 1. Configuração utilizada para geração de arquivos com funções. A ferramenta `llvm-extract` retorna arquivos compiláveis contendo a definição da função especificada.

Representação de Código. Devido ao sucesso do uso de *word-embeddings*, vetores densos de alta dimensão, para Processamento de Linguagem Natural, utilizamos a ferramenta `inst2vec` [6], que produz *embeddings* para instruções `LLVM` de um vocabulário formado por 8564 vetores de 200 dimensões. Para nossos experimentos, a codificação de cada aplicação corresponde a 300 vetores `inst2vec`. Caso a aplicação contenha um número de instruções inferior a este valor, completamos com vetores de inteiros que serão posteriormente ignorados durante o treinamento pela Rede Neural Artificial por uma camada *masking*. Utilizamos esta abordagem por duas razões: (1)- para que os conjuntos de dados possam ser totalmente carregados na memória e acelerar o treinamento, tendo em vista que foram gerados modelos de predição para diferentes conjuntos de dados, a utilização de geradores para carregar os dados durante o treinamento não possibilitaria a realização desta pesquisa em tempo hábil; (2)- como mostra a Tabela 1, a média de instruções (`opt -O0`) dos programas Angha15 e AnghaW, principal foco de análise desta pesquisa, não esta distante do valor escolhido, tornando a perda de informações pouco discrepante.

Sequências de Otimização. Foram construídas 10 sequências de otimização seguindo a metodologia proposta por Silva *et al* [Silva et al. 2021], utilizando algoritmos de aprendizagem de máquina e as 15.000 maiores funções `Angha`. Denotamos este conjunto de sequências por `Best_10`. Dado um programa não otimizado P_{noopt}, existe ao menos uma

[6] https://github.com/spcl/ncc

sequência $S_i \in$ `Best_10` que possui alta probabilidade de ser melhor ou igual à `opt -O0` ou `opt -Os` quando aplicada à P_{noopt}.

As sequências de `Best_10` são compostas por 25 passos de otimização, com sequências com tamanho variando no intervalo de inteiros $[9, 19]$. Todos os conjuntos de *benchmarks* descritos foram compilados com as sequências `Best_10`. A Tabela 3 indica a quantidade de amostras de cada conjunto utilizado para treinamento e teste.

Tabela 3. Número de amostras dos conjuntos compilados com `Best_10`.

		LLVM_Treino	**Angha15**	**AnghaW**	
#Treino		1.750	20.000	20.000	
	LLVM_Teste	**Coremark-f**	**Mibench-f**	**Coremark**	**Mibench**
#Teste	400	3.000	3.000	90	190

Arquitetura e Hiperparâmetros Nossa proposta consiste em utilizar arquiteturas `LSTM` e *embeddings* para codificação de programas e sequências. A Rede Neural Recorrente consiste em um modelo com duas entradas, onde uma entrada recebe a aplicação não otimizada codificada em vetores `inst2vec`, enquanto a outra recebe uma sequência de transformações codificada em inteiros no intervalo $[1, 19]$. Apesar de possível o processamento das sequências como inteiros, esta metodologia não é adequada, pois, não indica as relações de dependência entre as transformações da sequência. Dessa forma, uma camada `embedding` recebe as sequências de inteiros e as transformam em vetores densos de 32 dimensões. Este processo ocorre durante o treinamento.

Após o processamento da aplicação e da sequência de transformações por camadas `LSTM`, os vetores são então concatenados, formando um único vetor. Após a concatenação, aplicamos uma Normalização de Lotes, com o objetivo de estabilizar a aprendizagem e acelerar o treinamento. O vetor resultante serve como entrada para uma Rede Densa de duas camadas, cujo a saída é a estimativa do número de instruções da aplicação otimizada com a sequência processada. A Figura 2 ilustra nosso modelo de Rede Neural Artificial.

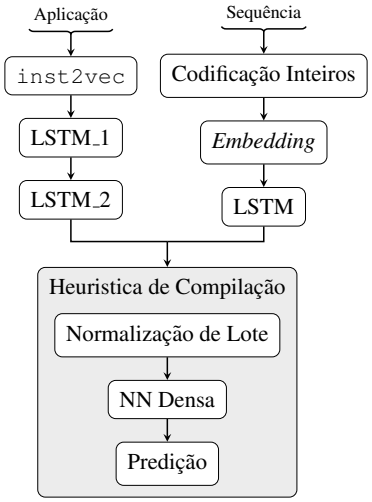


Figura 2. Modelo LSTM para predição de tamanho de código.

Todas as camadas do modelo utilizam a função de ativação `tanh`, com exceção da última camada, cujo a função é linear. O modelo foi testado com a função `ReLU`, porém, esta função de ativação ocasionou explosões de gradiente. A função de erro MAPE (*Mean Absolute Percentage Error*) 1 foi utilizada para calcular a acurácia, pois, sua saída é uma porcentagem e não depende do intervalo de valores y, portanto pode ser utilizada para comparação entre conjuntos de dados distintos.

$$MAPE(y, \hat{y}) = 100 * \frac{\sum_{i=1}^{n} |y - \hat{y}|}{n} \quad (1)$$

Além da métrica MAPE, avaliamos os conjuntos perante as métricas de correlação *Spearman* e *Pearson*, métricas que indicam relações lineares e monotônicas entre os dados. Os modelos foram ajustados para execução por 500 épocas, com critério de parada *EarlyStopping* definido para 20 épocas. O treinamento se deu em um *hardware* Intel(R) Xeon(R) CPU @ 2.20GHz, 4 núcleos, 26GB de RAM e `GPU` Tesla P100.

Buscamos investigar as seguintes questões pertinentes ao nosso sistema de compilação preditiva e aos conjuntos de dados avaliados:

- Qual a acurácia de nosso modelo nos diversos domínios de programas avaliados?
- Em qual cenário *AngaBench* é mais preciso?
- Quais os aspectos passíveis de melhorias do nosso preditor?
- *Benchmarks* sintéticos podem incrementar a acurácia do modelo quando utilizados em conjunto com aplicações reais?

5. Resultados e Discussão

Iniciamos mostrando o melhor erro médio entre 3 execuções do nosso preditor em cada conjunto de testes. A Figura 3 expõe a verdade com relação a classe de programas Embarcados. Nosso preditor não foi capaz de realizar estimativas com baixos erros. Com relação aos programas selecionados para teste da suíte `LLVM`, é notável que o conjunto de funções `Angha15` apresenta o maior erro, no mais, os outros conjuntos apresentam performance similar. `AnghaW` e `AnghaW+LLVM` apresentaram o menor erro médio para funções `Coremark-Pro`, $20,35\%$ e $20,75\%$ respectivamente, seguido por `LLVM` com $25,20\%$. Os erros tendem a ser maiores para funções `Mibench`, uma vez que, estas funções formam arquivos maiores, como indicado na Tabela 1. O menor erro médio para este conjunto foi obtido com a suíte de treinamento LLVM com um valor MAPE de aproximadamente $42,06\%$.

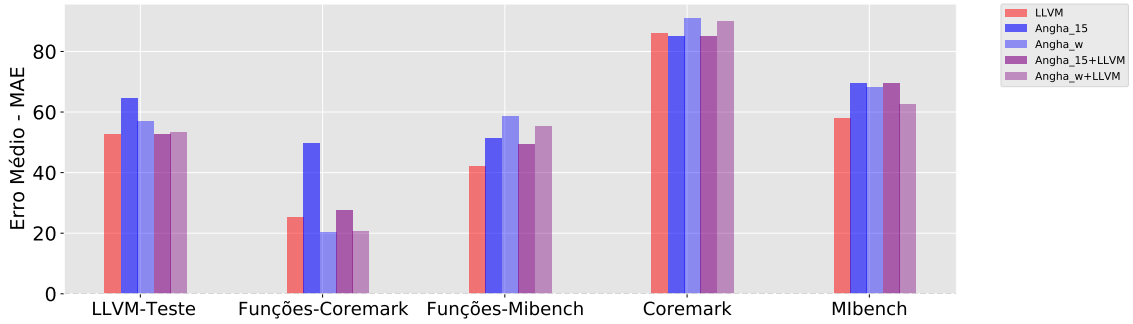


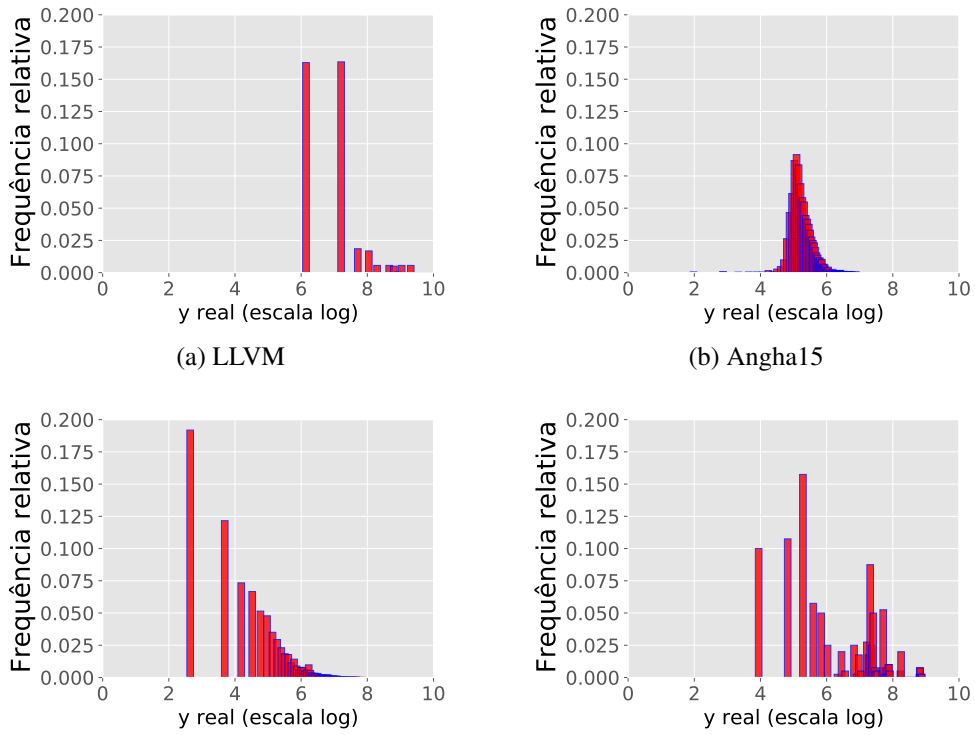
Figura 3. Melhor erro médio de 3 execuções para cada conjunto de testes.

Diferentemente, a Tabela 4 sumariza os resultados dos modelos com a melhor média em cada classe de programas. Os resultados indicam maior precisão de `AnghaW` com relação a `Angha15`, possuindo acurácia próxima à `LLVM`. Por outro lado, as combinações `Angha15+LLVM` e `AnghaW+LLVM` não proporcionaram ganhos com relação à `LLVM`, mas em sua maioria, incrementaram o erro.

Tabela 4. MAPE do modelo com melhor média da predição de tamanho de código

Conjunto de Dados	Funções	Embarcados	LLVM_Teste
`LLVM_Treino`	39, 15 %	72, 17 %	52, 60 %
`Angha15`	63, 61 %	78, 46 %	64, 57 %
`AnghaW`	39, 76 %	79, 76 %	56, 78 %
`Angha15+LLVM`	41, 42 %	78, 46 %	52, 49 %
`AnghaW+LLVM`	39, 06 %	76, 40 %	53, 39 %

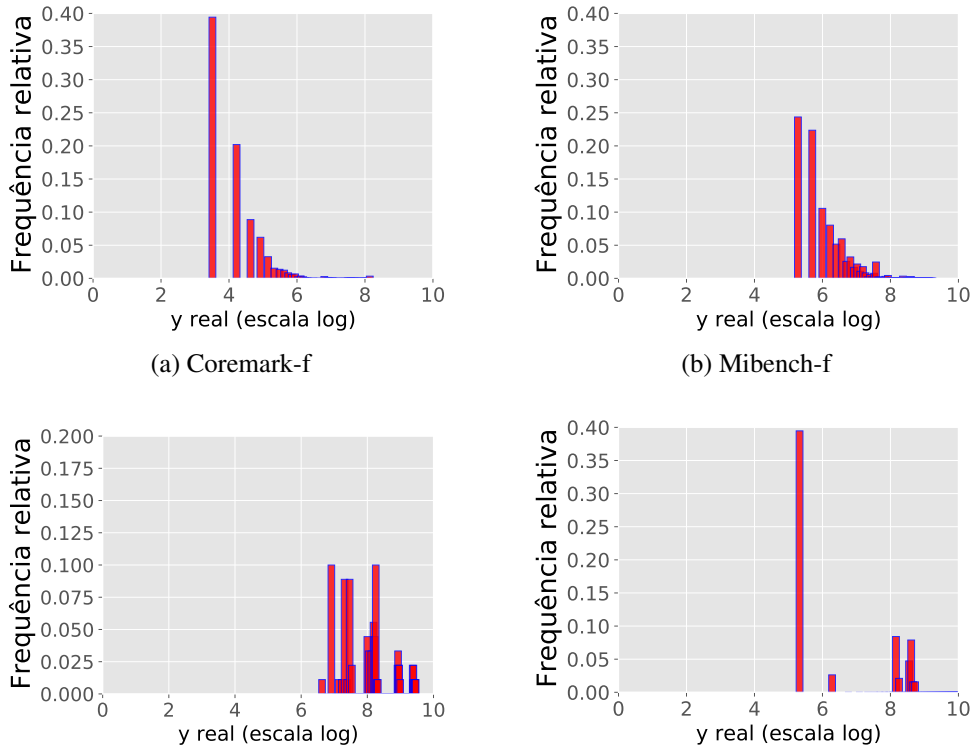
Os resultados, em parte, podem ser justificados pela distribuição dos alvos. As Figuras 4 e 5 expõe a frequência relativa dos valores reais, *i.e.*, a quantidade de instruções das amostras compiladas com sequências `Best_10`. Funções e *benchmarks* `Angha` tendem a ser arquivos menores e, ao observarmos (a), (b) e (c) é notável que `LLVM+Angha15` e `LLVM+AnghaW` deslocariam a frequência para valores menores, o que pode explicar a ausência de melhorias.



(a) LLVM (b) Angha15

(c) AnghaW (d) LLVM_Teste

Figura 4. Frequências relativas dos alvos (compilação com `Best_10`)



(a) Coremark-f (b) Mibench-f

(c) Coremark (d) Mibench

Figura 5. Frequências relativas dos alvos (compilação com `Best_10`)

Tabela 5. Coeficientes de *Pearson (CP)* e *Spearman (CS) dos modelos com melhor performance em cada conjunto individual de testes. A similaridade é calculada entre o valor real e o valor previsto*

	LLVM_Teste		**Coremark-f**		**Mibench-f**		**Coremark**		**Mibench**	
	CP	CS	CP	CS	CP	CS	CP	CS	CP	CS
LLVM_Treino	$0,34$	$0,37$	$0,44$	$0,91$	$0,04$	$0,1$	$-0,00$	$0,04$	$0,12$	$0,34$
Angha15	$0,22$	$0,34$	$0,14$	$0,21$	$0,26$	$0,19$	$0,02$	$0,07$	$0,46$	$0,57$
AnghaW	$0,24$	$0,27$	$0,39$	$0,92$	$-0,05$	$-0,01$	$-0,20$	$-0,24$	$-0,19$	$0,03$
Angha15+LLVM	$0,35$	$0,58$	$0,35$	$0,88$	$0,12$	$0,12$	$0,02$	$0,07$	$0,46$	$0,57$
AnghaW+LLVM	$0,24$	$0,27$	$0,39$	$0,92$	$0,32$	$0,90$	$0,01$	$0,04$	$-0,36$	$-0,33$

Como já discutido, a codificação dos programas em uma representação de tamanho fixo pode prejudicar a qualidade dos dados, pois, pode levar a perda de informações. Ao observarmos as Tabelas 1 e 2, a média dos *benchmarks* `Angha15` e `AnghaW` são $397,12$ e $267,10$ instruções, respectivamente. Portanto, a escolha do valor de truncamento correspondente a 300 vetores parece razoável. No entanto, isso pode prejudicar a aprendizagem do modelo em aplicações maiores, como é o caso das amostras `LLVM` e Embarcados, que contemplam aplicações com milhares de instruções. O fato da acurácia do conjunto `LLVM` (a) não ser satisfatória com relação ao conjunto Embarcados (g, h) pode estar relacionado a codificação já citada. Portanto, uma representação que seja capaz de codificar todas as propriedades dos *benchmarks* e seja invariante com relação ao tamanho do código pode resultar em melhorias.

Por fim, complementamos nossa avaliação calculando os coeficientes de similaridade de *Pearson* e *Spearman*, indicados na Tabela 5. Valores de coeficiente de *Pearson* próximos de $1,00$ indicam uma correlação linear muito forte entre os dados, por outro lado, valores de coeficiente de *Spearman* próximos de 1 indicam uma correlação monotônica muito forte. Valores positivos de *Spearman*, como no caso de funções `Coremark`, indicam que, conforme os valores reais das amostras crescem, os valores preditos pelo sistema também tendem a crescer.

6. Conclusões e Trabalhos Futuros

Neste trabalho, apresentamos um sistema de predição de tamanho de código para sequências de otimização, utilizando redes `LSTM` como heurística. Estudamos o comportamento de programas sintéticos `AnghaBench` em comparação com *benchmarks* reais. Os resultados indicam que, em alguns casos, *benchmarks* sintéticos apresentam performance similar a programas reais, mas não resultam em melhorias de acurácia quando utilizados em conjunto com *benchmarks* `LLVM`.

A análise dos conjuntos de dados, juntamente com o modelo de compilação preditiva nos leva a crer que, as limitações de acurácia podem ser parcialmente atribuídas aos *benchmarks* escolhidos, que não contemplam satisfatoriamente todos os domínios selecionados para avaliação, e a forma como a codificação das aplicações foi realizada. Acreditamos que o tratamento de cada um desses problemas pode levar em melhorias na acurácia. Os próximos passos incluem mitigar estas questões, utilizando representações de grafos (ASTs, GFCs) que podem enriquecer o modelo de aprendizagem com informações sobre dependência de dados e controle, juntamente com maiores conjuntos de dados.

Referências

Ashouri, A. H., Killian, W., Cavazos, J., et al. (2018). A Survey on Compiler Autotuning Using Machine Learning. *ACM Comput. Surv.*, 51(5).

Barany, G. (2018a). Finding Missed Compiler Optimizations by Differential Testing. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, page 82–92, New York, NY, USA. Association for Computing Machinery.

Barany, G. (2018b). Liveness-driven random program generation. In Fioravanti, F. and Gallagher, J. P., editors, *Logic-Based Program Synthesis and Transformation*, pages 112–127, Cham. Springer International Publishing.

Cummins, C., Petoumenos, P., Murray, A., et al. (2018). Compiler Fuzzing through Deep Learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 95–105, New York, NY, USA. Association for Computing Machinery.

Cummins, C., Petoumenos, P., Wang, Z., et al. (2017). Synthesizing Benchmarks for Predictive Modeling. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, page 86–99, Austin, USA. IEEE Press.

da Silva, A. F., Kind, B. C., de Souza Magalhães, J. W., et al. (2021). ANGHABENCH: A Suite with One Million Compilable C Benchmarks for Code-Size Reduction. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 378–390, Seoul, South Korea. IEEE Computer Society.

Goens, A., Brauckmann, A., Ertel, S., et al. (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*, MAPL 2019, page 38–46, New York, NY, USA. Association for Computing Machinery.

Hashimoto, A. and Ishiura, N. (2016). Detecting Arithmetic Optimization Opportunities for C Compilers by Randomly Generated Equivalent Programs. *IPSJ Transactions on System LSI Design Methodology*, 9:21–29.

Heo, K., Lee, W., Pashakhanloo, P., et al. (2018). Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 380–394, New York, NY, USA. Association for Computing Machinery.

Hochreiter, S. and Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Comput.*, 9(8):1735–1780.

Huang, Q., Haj-Ali, A., Moses, W., et al. (2020). AutoPhase: Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning.

Lattner, C. and Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*, pages 75–88, San Jose, CA, USA.

Otter, D. W., Medina, J. R., and Kalita, J. K. (2021). A Survey of the Usages of Deep Learning for Natural Language Processing. *IEEE Transactions on Neural Networks and Learning Systems*, 32(2):604–624.

Rocha, R. C. O., Petoumenos, P., Wang, Z., Cole, M., and Leather, H. (2019). Function merging by sequence alignment. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 149–163.

Rocha, R. C. O., Petoumenos, P., Wang, Z., Cole, M., and Leather, H. (2020). Effective function merging in the ssa form. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 854–868, New York, NY, USA. Association for Computing Machinery.

Silva, A. F. d., de Lima, B. N. B., and Pereira, F. M. Q. a. (2021). Exploring the Space of Optimization Sequences for Code-Size Reduction: Insights and Tools. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, CC 2021, page 47–58, New York, NY, USA. Association for Computing Machinery.

Trofin, M., Qian, Y., Brevdo, E., et al. (2021). MLGO: a Machine Learning Guided Compiler Optimizations Framework.

Wei, X.-S., Wu, J., and Cui, Q. (2019). Deep Learning for Fine-Grained Image Analysis: A Survey.

Yang, X., Chen, Y., Eide, E., et al. (2011). Finding and Understanding Bugs in C Compilers. *SIGPLAN Not.*, 46(6):283–294.

Zhang, S., Yao, L., Sun, A., et al. (2019). Deep Learning Based Recommender System: A Survey and New Perspectives. *ACM Computing Surveys*, 52(1).