

Alisson Ferreira Teles

**Refinamento e Integração do Modelo LLM
Compiler na Otimização de Código Alvo para
Microcontroladores STM32 na Robotics
Language**

Jataí-GO

2025

Alisson Ferreira Teles

Refinamento e Integração do Modelo LLM Compiler na Otimização de Código Alvo para Microcontroladores STM32 na Robotics Language

Monografia apresentada ao curso de Ciência da Computação do Instituto de Ciências Exatas e Tecnológicas da Universidade Federal de Jataí (UFJ), como requisito para obtenção do título de Bacharel em Ciência da Computação.

Universidade Federal de Jataí

Orientador: Prof. Dr. Thiago Borges de Oliveira

Jataí-GO

2025

Dedico este trabalho à minha Mãe, Carla Teles Lima, ao meu Pai, Carlos Borges de Souza, e à minha Avó, Rauilda Teles Lima, pelo o orgulho e confiança que sempre tiveram em mim durante a minha jornada de vida. Devo tudo a vocês, porque em cada passo meu esteve o esforço de cada um de vocês.

Agradecimentos

Agradeço à minha família, pelo amor, paciência e apoio constante em cada etapa, não só na minha jornada como acadêmico, mas em minha vida como um todo. Aos meus amigos e colegas de curso, pela companhia, pelas conversas, pelas dúvidas e respostas compartilhadas, e por toda a ajuda nos momentos em que precisei. Registro também minha gratidão a todos os professores que fizeram parte da minha formação e, em especial, ao meu orientador, Thiago Borges de Oliveira, pelas ideias, correções, conselhos e por cada contribuição depositada no desenvolvimento deste trabalho.

“Todos esses que aí estão atravancando meu caminho, eles passarão... Eu passarinho!..”
(Mario Quintana)

Resumo

Este trabalho investiga o uso de modelos de linguagem de grande porte (LLMs) para otimização de código de microcontroladores, com foco no microcontrolador STM32F103C8T6 e na linguagem de programação *Robotics Language* (ROBL). O objetivo principal é adaptar o modelo *LLM Compiler*, originalmente treinado para unidades centrais de processamento (CPUs), de propósito geral, ao contexto de sistemas embarcados com fortes restrições de recursos, visando à redução do tamanho do código objeto. Para isso, foi desenvolvido um fluxo de processamento que: (i) coleta programas na linguagem C provenientes dos conjuntos *Csmith* e *AnghaBench*; (ii) converte automaticamente esses programas para a linguagem ROBL por meio do conversor *c2rob*; (iii) compila os códigos convertidos com diferentes sequências de etapas de otimização do compilador LLVM, obtidas na literatura; (iv) seleciona, para cada programa, a sequência que gera o menor código objeto para a arquitetura baseada no microcontrolador STM32; e (v) organiza esses dados em pares de entrada e saída (*prompt-label*) para ajuste fino supervisionado do modelo *LLM Compiler*. Tal modelo foi refinado com cerca de 1,907 amostras e avaliado de forma exploratória em códigos que não foram utilizados no treinamento. Os testes indicam que o modelo é capaz de reproduzir, em alguns casos, padrões de otimização compatíveis com as sequências obtidas por busca automática, ainda que com capacidade de generalização limitada pelo tamanho do conjunto de dados usado no refinamento. A principal contribuição deste trabalho é a elaboração de um conjunto de rotinas e scripts que elaboram datasets de treinamento para modelos LLMs, que integram uma linguagem de programação voltada a microcontroladores (ROBL), o compilador *Robcmp* e um modelo de linguagem de grande porte especializado em compiladores, abrindo caminho para estudos futuros com conjuntos de dados maiores e avaliações quantitativas mais robustas.

Palavras-chave: Compiladores; Microcontroladores; Otimização de Código; Modelos de Linguagem; Aprendizado de Máquina; Modelos de Linguagem de Grande Porte.

Abstract

This work investigates the use of large language models (LLMs) for code optimization on microcontrollers, focusing on the STM32F103C8T6 microcontroller and the domain-specific programming language Robotics Language (ROBL). The main goal is to adapt the LLM Compiler model—originally trained for general-purpose central processing units—to the context of embedded systems with strict resource constraints, targeting object code size reduction. To this end, we designed a complete processing pipeline that: (i) collects C programs from the Csmith and AnghaBench datasets; (ii) automatically converts these programs into ROBL using the c2rob converter; (iii) compiles the converted codes with different sequences of optimization passes of the LLVM compiler, including short sequences proposed in the literature; (iv) selects, for each program, the sequence that produces the smallest object file for the architecture based on the STM32 microcontroller; and (v) organizes these results into input–output pairs (prompt–label) for supervised fine-tuning of the LLM Compiler. The refined model was trained on approximately two thousand examples and evaluated in an exploratory manner on codes that were not used during training. The experiments show that, in some cases, the model can reproduce optimization patterns consistent with those obtained through automatic search, although its generalization ability is limited by the reduced size of the training set. The main contribution of this work is a reproducible pipeline that integrates a microcontroller-oriented programming language, the Robcmp compiler, and a compiler-focused large language model, paving the way for future studies with larger datasets and more comprehensive quantitative evaluations.

Keywords: Compilers; Microcontrollers; Code Optimization; Language Models; Machine Learning; Large Language Models.

Lista de ilustrações

Figura 1 – Arquitetura macro de um compilador. Adaptada de Cooper e Torczon (2012).	18
Figura 2 – Exemplo de árvore ambígua de derivação para a expressão $A + B - C$	20
Figura 3 – AVR16DD14, de 24 Mhz, com 14 portas de entrada e saída	23
Figura 4 – Placa com um microcontrolador STM32F103C8T6, de 72 Mhz	24
Figura 5 – Fluxograma de etapas metodológicas.	37
Figura 6 – Esquema de Treinamento e Inferência do LLM Compiler para STM32. Figura adaptada de Cummins et al. (2023)	40
Figura 7 – Comparação de tamanho entre os códigos otimizados pelas sequências do (FAUSTINO et al., 2021) e os códigos otimizados pela <i>flag -Oz</i>	41

Lista de tabelas

Tabela 1 – Comparativo entre trabalhos relacionados	34
---	----

Sumário

1	INTRODUÇÃO	12
1.1	Motivação	12
1.2	Objetivo do Trabalho	15
1.3	Contribuição do Trabalho	15
2	REFERENCIAL TEÓRICO	17
2.1	Introdução	17
2.2	Compiladores	17
2.2.1	<i>Front-End</i>	18
2.2.2	Analisador Léxico	18
2.2.3	Analisador Sintático	19
2.2.4	Analisador Semântico	20
2.2.5	<i>Middle-End</i>	21
2.2.6	<i>Back-end</i>	21
2.3	Microcontroladores	22
2.4	Linguagem Específica de Domínio e a <i>Robotics Language</i>	23
2.5	Inteligência Artificial	25
2.5.1	Aprendizado de Máquina	25
2.5.2	Treinamento supervisionado	26
2.5.3	<i>Large Language Models</i> (LLMs)	27
2.6	Otimização em compiladores	28
2.6.1	Otimização Clássica – sem Inteligência Artificial	28
2.6.2	Otimização Com Inteligência Artificial	28
2.6.3	Otimização com <i>Large Language Models</i>	29
3	TRABALHOS RELACIONADOS	32
3.1	Introdução	32
3.1.1	Trabalhos Analisados	32
3.1.2	New Optimization Sequences for Code-Size Reduction for the LLVMCompilation Infrastructure	32
3.1.3	CompilerDream: Learning a Compiler World Model for General Code Optimization	33
3.1.4	LLMCompiler: Foundation Language Models for Compiler Optimization	34
3.1.5	Resumo Comparativo	34
4	METODOLOGIA E EXPERIMENTOS	36

4.1	Classificação da Pesquisa	36
4.2	Aspectos Metodológicos Gerais	36
4.3	Conjunto de Dados	37
4.4	Conversor de código C para RobCmp: c2rob	38
4.5	Formação dos Dados de Treinamento	38
4.6	Caracterização do Dataset quanto a Sequência de Passes	40
4.7	Treinamento do Modelo	41
4.8	Especificações de Hardware e Software	42
4.9	Resultados: Refinamento do Modelo	42
4.10	Resultados: Ferramentas para Geração do Dataset de Treinamento	44
4.11	Discussão e Limitações	46
5	CONCLUSÃO E TRABALHOS FUTUROS	48
5.1	Trabalhos Futuros	48
	REFERÊNCIAS	50
	ANEXOS	53
	ANEXO A – CÓDIGO 00668.ROB, E RESULTADO DA INFERÊNCIA	54
A.1	Código em ROBL	54
A.2	Resultado da Inferência	59
	ANEXO B – CÓDIGO 08016.ROB, USADO PARA CRIAR O IR PARA INFERÊNCIA	60
B.1	Código em ROBL	60
B.2	Resultado da Inferência	61
	ANEXO C – CÓDIGO 45071.ROB, USADO PARA CRIAR O IR PARA INFERÊNCIA	62
C.1	Código em ROBL	62
C.2	Resultado da Inferência	63
	ANEXO D – CÓDIGO EXTR_.....GDBSTUB.C_GDB_CMD_MEMWRITE, USADO PARA CRIAR O IR PARA INFERÊNCIA	64
D.1	Código em ROBL	64
D.2	Resultado da Inferência	65

	ANEXO E – CÓDIGO EXTR_LIBCOREPROXY.C_COMPRESS_HINT_TO_ENUM.ROB,	
	USADO PARA CRIAR O IR PARA INFERÊNCIA . .	66
E.1	Código em ROBL	66
E.2	Resultado da Inferência	67

1 Introdução

1.1 Motivação

A busca por otimização de código é cada vez mais importante no contexto de sistemas embarcados, nos quais os recursos computacionais são extremamente limitados (HONG; PARK; KIM, 2025). Os microcontroladores, ou MCUs, que são o cérebro destes sistemas e os responsáveis por executar tarefas específicas de acordo com a programação definida, contam com menos memória e menor capacidade de processamento quando comparados aos processadores usados em computadores *desktops* ou *notebooks*. Um exemplo são os modelos de microcontroladores da família STM32 (STMicroelectronics, 2023), amplamente utilizados na indústria, que apresentam memórias SRAM variando entre 2 e 620 KB e podem operar com frequência de até 600 MHz. Cada família de microcontroladores apresenta um conjunto de registradores, periféricos e instruções da CPU, as quais devem ser consideradas no processo de otimização.

A otimização de código consiste em aplicar técnicas, ou transformações, que melhoram o desempenho ou reduzem o tamanho de um programa sem alterar sua funcionalidade original (MASLOV; GANESH, 2014). A eficácia de cada transformação pode variar conforme o programa e a arquitetura de hardware considerada (WANG; O'BOYLE, 2018). Por exemplo, um microcontrolador AVR ATmega328P, arquitetura RISC de 8 *bits*, possui um conjunto reduzido de 131 instruções, em sua maioria executando em um ciclo de *clock* e recursos de memória bem modestos: apenas 2 KB de RAM e 32 KB de memória *flash* (AVR16/Datasheet, 2024). Ao contrário, um processador moderno como o Intel Core i7-1165G7, arquitetura CISC de 64 *bits*, suporta centenas de instruções e conta com memórias *flashes* e execução fora de ordem para alto desempenho (INTEL®™, 2025). Essas diferenças de arquitetura impactam a geração de código intermediário (*Intermediate Representation*): uma operação que resulta em uma única instrução complexa no Core i7 pode requerer várias instruções mais simples no AVR, influenciando o tamanho do código gerado e o nível de otimização possível em cada caso.

Para simplificar o uso dos MCUs, os fabricantes fornecem bibliotecas de funções e rotinas, geralmente nomeadas como *Hardware Abstraction Layers* (HALs). Como cada HAL é específico do fabricante, a diversidade de periféricos e a falta de padronização fazem com que as interfaces de programação específicas não sejam compatíveis entre si. Como resultado, sempre que um novo modelo de microcontrolador é adotado em um sistema embarcado, os programadores precisam aprender sua interface de programação específica, dificultando ainda mais a otimização de código. Neste sentido, foram propostas ao longo do tempo, bibliotecas para linguagem de programação de propósito geral e *frameworks*. Esses

frameworks atuam como uma camada de *software* intermediária, oferecendo funções que vão além das camadas HAL disponibilizadas pelos fabricantes. Um exemplo bastante conhecido é o Arduino ([ARDUINO, 2025](#)), que abstrai o acesso aos periféricos mais utilizados em plataformas como AVR, STM32 e ExpressIf32. Outras ferramentas similares são o CMSIS¹, MBED², STM32CUBE³ e LibOpenCM3⁴, sendo essas últimas especificamente voltadas para arquiteturas baseadas em ARM.

Tendo a otimização de código um papel fundamental na realização do potencial máximo de *software* e *hardware*, desenvolvedores buscam uma solução universal para transformar programas de entrada em versões semanticamente equivalentes, porém mais eficientes, sem esforço manual. A fim de alcançar este objetivo, os compiladores utilizam um *front-end*, que traduz o código-fonte de uma linguagem de programação para uma representação intermediária (IR), um *middle-end optimizer*, que executa otimizações independentes da linguagem e da plataforma sobre a IR do código original, e um *back-end*, que converte a IR em código binário. O otimizador é comumente implementado como uma sequência de passes que aplicam transformações no código. O desempenho da otimização depende principalmente da seleção e ordem das passagens de otimização ([DENG et al., 2025](#)).

Os compiladores disponibilizam estratégias de otimização com ordens pré-definidas, cada uma estabelecendo um conjunto ordenado de passes de otimização, como -O1, -O2 e -O3 para melhorar a velocidade de execução, e -Os e -Oz⁵ para reduzir o tamanho do programa. Porém, com a diversidade de programas e plataformas, essas estratégias pré-definidas acabam não sendo ideais em todos os casos, e é possível melhorar o desempenho de um programa específico se encontrarmos uma sequência de passes melhor, comparada com os passes fornecidos pelos compiladores ([GEORGIOU et al., 2018](#); [CUMMINS et al., 2025](#); [DENG et al., 2025](#)).

Na última década, no trabalho de [Wang e O'Boyle \(2018\)](#), foi empregado inteligência artificial para otimização de código por meio de aprendizado de máquina, utilizando atributos como tamanho das funções e/ou quantidade de blocos básicos, definidos com base no conhecimento humano, para representar o programa de forma que um modelo de aprendizado de máquina conseguisse operar sobre ele. Outras abordagens usaram redes neurais gráficas ([LIANG et al., 2023](#)). Contudo, segundo [Cummins et al. \(2025\)](#), tais pesquisas têm um problema em comum: elas não conseguem representar o programa original por completo, o que interfere no aprendizado e resultado do modelo de IA. Modelos

¹ Site do CMSIS: <https://www.arm.com/technologies/cmsis>

² Site do MBED: <https://os.mbed.com/handbook/mbed-Microcontrollers>

³ Site do STM32CUBE: <https://www.st.com/en/ecosystems/stm32cube.html>

⁴ Site do LibOpenCM3: <https://libopencm3.org>

⁵ -Oz existe somente na família de compiladores do LLVM.

de IA projetados para representação textual, como ChatGPT⁶, *Code Llama*⁷ e Codex⁸, avançam no entendimento estatístico de códigos e podem sugerir conclusões prováveis para códigos incompletos. No entanto, os mesmos não foram treinados de forma específica para otimização de código. O ChatGPT, por exemplo, consegue fazer pequenos ajustes em programas, como marcar variáveis para serem armazenadas como registradores e até tenta otimizações mais substanciais como vetorização. No entanto, ele com frequência alucina e comete erros, resultando muitas vezes em códigos incorretos (CUMMINS et al., 2025).

Recentemente, a elaboração de *Large Language Models* (LLMs) para otimização de código tem mostrado resultados promissores, como evidenciado por Cummins et al. (2025), que apresentou o modelo LLM Compiler, treinado com aprendizado supervisionado, especificamente para selecionar passes de otimização no nível LLVM-IR, alcançando uma redução média de 3 % a 5 % na contagem de instruções em comparação à otimização padrão do compilador LLVM com a flag -Oz. Além disso, os autores disponibilizaram os modelos publicamente na plataforma *Hugging Face*⁹, sob uma licença comercial personalizada, com o objetivo de incentivar ampla reutilização, permitindo assim que pesquisadores acadêmicos e profissionais da indústria possam expandir e aprofundar pesquisas nessa área emergente.

O *Hugging Face* destaca-se como uma plataforma central no atual ecossistema de Inteligência Artificial, atuando como um repositório massivo de modelos de linguagem e outros artefatos de *machine learning*. O *Hugging Face Hub* hospeda hoje mais de 900 mil modelos de IA, além de centenas de milhares de conjuntos de dados e aplicativos de demonstração, todos de acesso aberto, facilitando a colaboração e a reutilização pela comunidade. Essa plataforma cumpre um papel de distribuição e compartilhamento de modelos: pesquisadores e desenvolvedores podem publicar seus modelos treinados, permitindo que outros os utilizem ou aprimorem sem precisar treiná-los do zero. Isso democratiza o acesso a modelos avançados de linguagem e incentiva uma abordagem comunitária na evolução da IA, em contraste a soluções proprietárias restritas a grandes empresas (HUGGING, 2025).

O *LLM Compiler* Cummins et al. (2025) é voltado à otimização de compiladores, porém focado em arquiteturas de CPUs de propósito geral (principalmente desktop/servers x86-64 e ARM de 64 bits) não abrangendo microcontroladores de 8 ou 16 bits. O modelo foi treinado com conjuntos de dados de LLVM-IR e código de montagem dessas plataformas de alto desempenho. Assim, surge a necessidade de customizar o modelo, de forma a torná-lo assertivo para microcontroladores (MCUs) como o STM32, considerando suas restrições arquiteturais (baixos clocks na ordem de dezenas de MHz e pouquíssimos kilobytes de memória disponíveis). Um *LLM Compiler* especializado para MCUs poderia ser treinado

⁶ Site de introdução do ChatGPT: <<https://openai.com/index/chatgpt/>>

⁷ Site da Meta Code Llama: <<https://www.llama.com/code-llama>>

⁸ Site do Codex: <<https://openai.com/index/openai-codex>>

⁹ Site do *Hugging face*: <<https://huggingface.co>>

para “entender” as peculiaridades do código em arquiteturas de 8/16 bits e otimizar visando tamanho e eficiência energética, explorando instruções e padrões específicos desses processadores de recursos limitados. Em outras palavras, o *LLM Compiler* proposto por Cummins et al. (2025) mostrou o potencial dos modelos de linguagem na otimização de código para CPUs poderosas e é possível estender essa abordagem aos microcontroladores, refinando o treinamento do modelo.

Neste sentido, os projetos “*Robotics Language: Uma Linguagem de Programação de Propósito Específico para Microcontroladores*” (PI05974-2024) e seu antecessor “Especificação e Construção de Protótipos Funcionais de Kits Robóticos de Baixo Custo para uso em Processos de Ensino-Aprendizagem” (PI02361-2018) (OLIVEIRA, 2024), ambos desenvolvidos na Universidade Federal de Jataí, têm como foco o desenvolvimento de uma linguagem de programação voltada especificamente para aplicações em robótica e microcontroladores, denominada *The Robotics Language* (ROBL), juntamente com seu compilador correspondente, o Robcmp, com o fim de criar um ecossistema de robótica educacional de baixo custo.

1.2 Objetivo do Trabalho

Portanto, este trabalho tem como objetivo refinar o modelo *LLM Compiler* no contexto de MCUs, especificamente o STM32F103C8T6, usando como base programas escritos na *Robotics Language*. A proposta busca adaptar técnicas de otimização já consolidadas para arquiteturas de propósito geral à realidade de sistemas embarcados com recursos limitados, ainda pouco explorada na literatura. Os objetivos específicos foram:

1. Converter repositórios de código escritos em C para a ROBL, ignorando os códigos em C que necessitam de funcionalidades inexistentes na ROBL;
2. Compilar o repositório convertido usando otimizações pré-determinadas e busca semi-exaustiva de passes, para encontrar o melhor código otimizado de cada programa;
3. Criar, para treinamento supervisionado do modelo, pares *prompt-label* com o código em ROBL e o melhor código otimizado encontrado; e
4. Executar experimentos para validar o desempenho do modelo refinado ao indicar sequências de passes para otimização de programas não usados no treinamento, escritos em ROBL.

1.3 Contribuição do Trabalho

Como contribuição, desenvolveu-se um conjunto de *scripts* de preparação de dados, conversão de código, *autotuning* e avaliação quantitativa, que geram datasets de treinamento

para modelos LLMs. Os datasets podem ser utilizados para refinar ou treinar modelos, com o objetivo de produzir sequências de passes de otimização a partir de programas fonte de entrada, visando gerar versões de código objeto menores.

2 Referencial Teórico

2.1 Introdução

Este capítulo apresenta os conceitos necessários para a compreensão deste texto, definindo conceitos sobre os compiladores e suas principais características, Inteligência Artificial e, por fim, estratégias de otimização em compiladores.

2.2 Compiladores

Compiladores são fundamentais para a computação moderna. Eles atuam como tradutores, transformando linguagens de programação orientadas ao ser humano em linguagens de máquina orientadas ao computador (FISCHER; CYTRON; LEBLANC, 2010). Com o surgimento de novas instruções para MCUs e CPUs, os compiladores devem se adaptar, com o fim de melhorar seu desempenho em processamento e tempo de execução.

Um compilador é uma ferramenta que traduz *software* escrito em uma linguagem para outra linguagem. Para traduzir texto de uma linguagem para outra, a ferramenta precisa entender tanto a forma, ou sintaxe, quanto o conteúdo, ou significado, da linguagem de entrada. Ela precisa compreender as regras que governam a sintaxe e o significado na linguagem de saída. Por fim, ela precisa de um esquema para mapear o conteúdo da linguagem fonte para a linguagem alvo (COOPER; TORCZON, 2012).

Um compilador é dividido internamente em fases, cada uma contribuindo para o processo de compilação (FOLEISS et al., 2009). A Figura 1 mostra a sequência macro de passos de um compilador. Segundo Fischer, Cytron e LeBlanc (2010), após receber um programa fonte como entrada, um compilador passa por três etapas principais:

- i. O *front-end*, onde serão realizadas as etapas de análise léxica, sintática e semântica, além de produzir a árvore sintática e a transformar em código intermediário (ou representação intermediária-IR) (sendo este item um *parser*);
- ii. O *middle-end* que processa as IRs de maneira que beneficia as fontes e destinos(basicamente gerando otimizações sobre as IRs, excluindo funções desnecessárias etc.); e
- iii. O *back-end* que gera a linguagem de destino ou objeto.

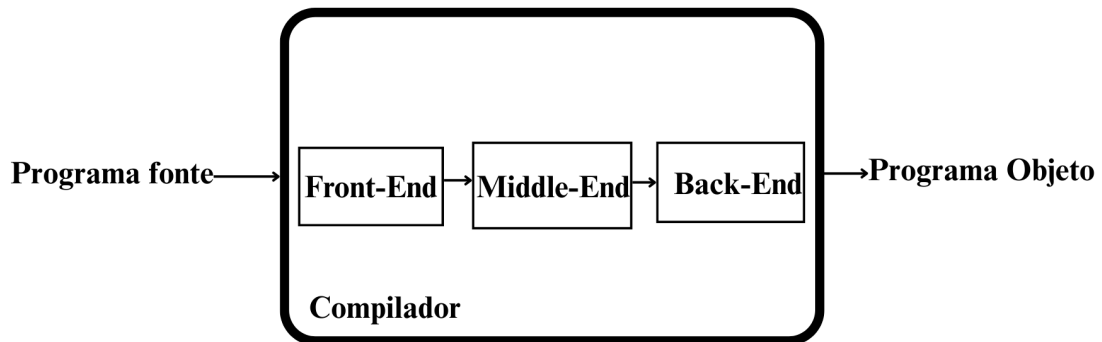


Figura 1 – Arquitetura macro de um compilador. Adaptada de [Cooper e Torczon \(2012\)](#).

2.2.1 *Front-End*

O *Front-end* determina se o código fonte é bem formado em termos de sintaxe e semântica. Se o código é válido, é criada uma representação intermediária sua no compilador; se não é válido, é reportado de volta ao usuário com mensagens de erro como diagnóstico para identificar problemas no código ([COOPER; TORCZON, 2012](#), p 11). Por sua vez, o *front-end* do compilador se divide em etapas, cada uma com sua respectiva funcionalidade na tradução do código-fonte. De acordo com [Cooper e Torczon \(2012\)](#), a primeira etapa é a análise léxica, seguida da análise sintática, análise semântica e, por último, o gerador de código intermediário. Etapas estas são descritas a seguir.

2.2.2 Analisador Léxico

Como descrito por [Fischer, Cytron e LeBlanc \(2010\)](#), o principal objetivo da análise léxica é transformar um fluxo de caracteres em um fluxo de *tokens*. Além disso, o analisador léxico ou *scanner* é o único passo de um compilador que manipula cada caractere do programa de entrada ([COOPER; TORCZON, 2012](#)).

Os analisadores léxicos dos compiladores são comumente implementados com o uso de expressões regulares, que funcionam como uma forma compacta de especificar autômatos finitos não determinísticos (AFNs). Esses padrões utilizam uma combinação de operadores e símbolos para descrever classes de caracteres e regras de repetição ([COOPER; TORCZON, 2012](#)). De acordo com [Blauth \(2010\)](#), os principais elementos das expressões regulares incluem: o ponto (.), que representa qualquer caractere; os colchetes ([]), que

definem uma classe de caracteres; o acento circunflexo (^), que nega uma classe; a barra vertical (|), que representa a união de padrões; o asterisco (*), que permite zero ou mais repetições; o sinal de soma (+), que exige pelo menos uma ocorrência; e o ponto de interrogação (?), que indica uma ou nenhuma ocorrência. Por exemplo, a expressão regular $[0-9]^+$ reconhece qualquer sequência de dígitos decimais, ou seja, números naturais diferentes do conjunto vazio. Este tipo de expressão pode ser utilizado para definir *tokens* numéricos inteiros, sendo uma parte essencial no funcionamento do analisador léxico de linguagens de programação.

2.2.3 Analisador Sintático

O analisador sintático, também conhecido como *parser*, opera sobre o fluxo de *tokens* gerado pelo analisador léxico, verificando se a sequência segue as regras gramaticais da linguagem. O analisador sintático solicita um *token* ao analisador léxico, que o gera a partir do código fonte. O *token* recebido é avaliado dentro das regras sintáticas e eventualmente passa a compor a tabela de símbolos (AHO; SETHI; ULLMAN, 2007). Quando encontra construções que violam a sintaxe, o *parser* reporta erros sintáticos e, em alguns casos, tenta realizar a recuperação de erros para permitir que a análise prossiga (FISCHER; CYTRON; LEBLANC, 2010). A partir dessa análise, caso a sintaxe siga as regras gramaticais da linguagem, é construída uma árvore sintática que será utilizada nas próximas fases do compilador (COOPER; TORCZON, 2012).

Para a análise sintática, utiliza-se uma gramática livre de contexto (GLC), composta por regras formais que descrevem como os *tokens* podem ser agrupados. Essas regras envolvem símbolos terminais (os próprios *tokens*) e não terminais (variáveis que representam agrupamentos de *tokens*), permitindo definir a estrutura hierárquica de uma linguagem (BLAUTH, 2010). No entanto, uma das principais dificuldades no projeto de gramáticas é garantir que não haja ambiguidade (situação em que uma mesma cadeia de entrada pode gerar mais de uma árvore de derivação). Por exemplo, a Figura 2 demonstra uma AST ambígua para a expressão $A + B - C$. Uma gramática ambígua permite diferentes formas de agrupamento dos operadores, resultando em árvores de sintaxe abstrata (ASTs) distintas e, conseqüentemente, comportamentos diferentes na execução do programa. Como as fases posteriores da tradução associarão significado com a forma detalhada da árvore sintática, múltiplas árvores implicam múltiplos significados possíveis para um único programa — o que é uma característica indesejável para uma linguagem de programação. Nessas situações, o compilador não teria como inferir automaticamente qual estrutura é a correta, o que evidencia a importância de projetar gramáticas bem definidas e livres de ambiguidade (COOPER; TORCZON, 2012).

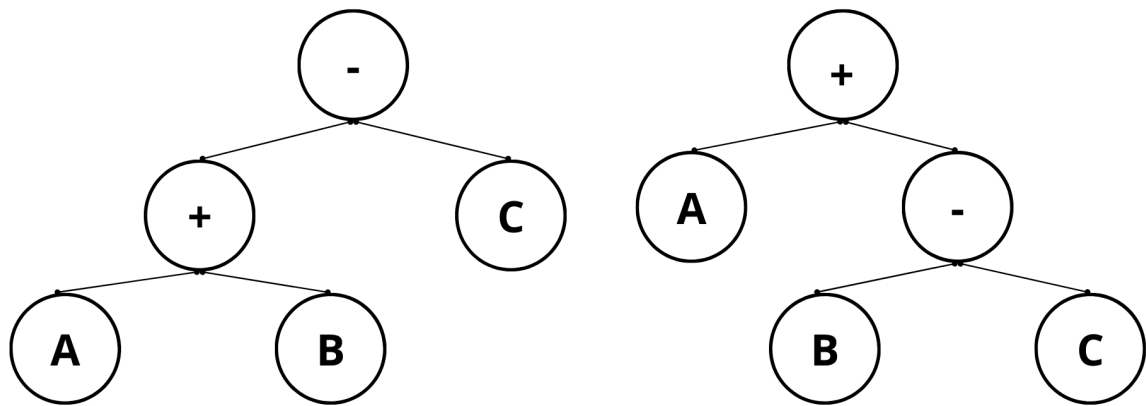


Figura 2 – Exemplo de árvore ambígua de derivação para a expressão $A + B - C$.

2.2.4 Analisador Semântico

A análise semântica é responsável por garantir que as construções do programa, mesmo estando sintaticamente corretas, também façam sentido lógico dentro do contexto da linguagem. É nessa etapa que o compilador verifica aspectos como a compatibilidade entre tipos de dados e o uso apropriado de identificadores. Um dos pontos centrais dessa fase é a verificação de tipos, onde o compilador analisa se os operadores estão sendo aplicados sobre operandos válidos conforme as regras definidas pela linguagem de programação. Essa verificação é essencial para evitar inconsistências que, embora passem pela análise sintática, podem resultar em comportamentos incorretos durante a execução (AHO; SETHI; ULLMAN, 2007).

Como dito por Costa et al. (2023), a função do analisador semântico é identificar erros que não podem ser detectados apenas pela análise sintática, utilizando a tabela de símbolos como suporte para garantir que a árvore gerada pelo *parser* (AST) esteja em conformidade com as regras semânticas da linguagem. Essa verificação é essencial para garantir que o programa resultante seja seguro, funcional e livre de inconsistências. Esse processo envolve percorrer cada nó da AST e validar sua estrutura e significado, assegurando que as operações representadas sejam logicamente válidas (FISCHER; CYTRON; LEBLANC, 2010).

Entre as validações mais comuns estão: compatibilidade de tipos em atribuições, verificação de parâmetros em chamadas de função, uso de constantes, e a exigência de que variáveis e funções sejam declaradas antes de seu uso.

2.2.5 Middle-End

De acordo com Fischer, Cytron e LeBlanc (2010), em um compilador, o termo *front-end* refere-se às fases responsáveis por analisar o código-fonte, enquanto o *back-end* lida com a geração do código de saída, geralmente na forma de código de máquina ou *assembly*. Entre essas duas etapas, existe um conjunto intermediário de fases conhecido como *middle-end* (COOPER; TORCZON, 2012). O *middle-end* permite aplicar otimizações e transformações na representação intermediária (IR) do código, de forma independente da linguagem de entrada e da arquitetura de destino (FISCHER; CYTRON; LEBLANC, 2010). A tarefa do otimizador é transformar o programa em IR, produzido pelo *front-end*, de uma forma que melhore a qualidade do código produzido pelo *back-end* (COOPER; TORCZON, 2012).

As otimizações realizadas no middle-end, ou passes de otimização, podem variar amplamente. No caso do LLVM, os passes de otimização são divididos em:¹

1. Passes de análise (*Analysis Passes*): calculam informações que podem ser usadas por outros passes, ou para fins de depuração ou visualização do programa. Por exemplo, o passe “*instcount*”, que apenas conta quantas instruções diferentes existem no programa.
2. Passes de transformação (*Transformer Passes*): podem usar (ou invalidar) os passes de análise. Todos os passes de transformação modificam o programa de alguma forma. Por exemplo, o passe “*dce*”, *dead code elimination*, remove partes do código que nunca são usadas, como variáveis que são criadas mas nunca lidas.
3. Passes de utilidade (*Utility Passes*): oferecem funções de apoio que não analisam nem modificam o código, mas ajudam em outras tarefas. Por exemplo, um passe que dá nomes para instruções sem nome é útil para facilitar a leitura ou comparação de versões do código. Como exemplo de passe temos o “*verify*” que age como um revisor ortográfico: ele verifica se o código está escrito corretamente segundo as regras internas do compilador. Ele não muda o código, mas avisa se há erros ou coisas malfeitas.

2.2.6 Back-end

A etapa do *back-end* (ou gerador de código) de um compilador só entra em ação após todas as fases de compilação anteriormente citadas e tem como função converter a representação intermediária (IR), gerada pelo *front-end*, em código objeto ou código de máquina compatível com a arquitetura de destino (COOPER; TORCZON, 2012). As exigências que são tradicionalmente impostas sobre o *back-end* são as seguintes: O código

¹ Passes do LLVM: <<https://llvm.org/docs/Passes.html>>

objeto precisa ser correto e de alta qualidade, significando que faz bom uso dos recursos da máquina-alvo (AHO; SETHI; ULLMAN, 2007).

Embora certos aspectos da geração de código variem conforme a arquitetura da máquina-alvo e o sistema operacional utilizado, há elementos fundamentais que são comuns à maioria dos compiladores. Entre eles, destacam-se a gestão de memória, a seleção de instruções adequadas à arquitetura, a alocação eficiente de registradores e a determinação da ordem de avaliação das expressões. Esses elementos desempenham papel central no processo de geração de código. Como dito anteriormente, a entrada para essa etapa é composta pela representação intermediária (IR) gerada pelo *front-end* do compilador, bem como pelas informações contidas na tabela de símbolos. Esta tabela é essencial para o mapeamento correto dos objetos de dados durante a execução, pois fornece os dados necessários para localizar os endereços associados aos identificadores presentes na IR. A saída é o programa objeto (AHO; SETHI; ULLMAN, 2007).

Assim o *back-end* converte a representação intermediária em código de máquina adaptado à arquitetura de destino (FISCHER; CYTRON; LEBLANC, 2010). Essa etapa ganha importância especial quando o alvo são microcontroladores, cujas limitações de memória e processamento exigem geração de código altamente otimizada, como veremos na seção a seguir.

2.3 Microcontroladores

Segundo Hussain et al. (2016), um microcontrolador pode ser entendido como um pequeno computador encapsulado em um único chip. Ele reúne, em um mesmo circuito integrado, uma unidade de processamento (CPU), memória RAM, algum tipo de memória não volátil (como ROM ou Flash) e interfaces de entrada e saída. Ao contrário dos computadores de uso geral, que são projetados para executar diversas aplicações, os microcontroladores são desenvolvidos com foco em tarefas específicas, operando geralmente em sistemas embarcados com uma única finalidade. Produtos controlados automaticamente, como sistemas de controle automático de motor, controles remotos, ferramentas elétricas, brinquedos e máquinas de escritório, ou seja, fotocopiadoras e impressoras que são comumente usadas, estão sendo programados usando microcontroladores. Dois exemplos de microcontroladores de baixo custo são:

- AVR: Elaborado pela Microship Technology, com uma arquitetura flexível e de baixo consumo de energia, incluindo o Event System, recursos analógicos inteligentes e periféricos digitais avançados. A Figura 3 apresenta um exemplo de microcontrolador da família AVR, desenvolvido pela *Microchip Technology*, é o modelo AVR16DD14. Ele conta com 14 pinos configuráveis para entrada e saída, opera com frequência

máxima de 24 MHz e possui 2 kB de memória SRAM, além de 16 kB de memória Flash para armazenamento do programa² (Microchip Technology Inc., 2024).

- STM32: A família STM32, da *STMicroelectronics*, é composta por microcontroladores de 32 bits baseados no núcleo ARM Cortex-M, conhecidos por sua alta performance, suporte a tempo real, baixo consumo e operação em baixa voltagem. Um exemplo é o STM32F103C8T6, que utiliza o núcleo ARM Cortex-M3, operando a até 72 MHz, com 128 kB de memória Flash, 20 kB de SRAM e até 37 portas de entrada/saída. Ele também conta com diversos periféricos conectados a dois barramentos APB, oferecendo boa flexibilidade para aplicações embarcadas. A Figura 4 mostra uma placa com esse microcontrolador em destaque (STMicroelectronics, 2023).³

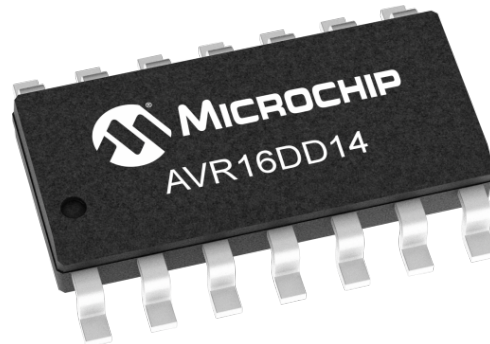


Figura 3 – Um dos microcontroladores da linha AVR, desenvolvido pela Microchip Technology (empresa que incorporou a antiga Atmel), é o AVR16DD14. Esse modelo conta com 14 portas de entrada e saída, opera com frequência máxima de 24 MHz, e dispõe de 2 kB de memória SRAM e 16 kB de memória Flash para armazenamento do código (Microchip Technology Inc. (2024)).

2.4 Linguagem Específica de Domínio e a *Robotics Language*

Linguagens específicas de domínio (DSLs) são linguagens de programação desenvolvidas com foco em um conjunto restrito de problemas, oferecendo maior expressividade e simplicidade dentro daquele contexto em comparação às linguagens de propósito geral. Por serem projetadas com base no conhecimento técnico de um domínio específico (como bancos de dados, estatística, sistemas embarcados ou mesmo aplicações web), as DSLs tornam a escrita e a leitura do código mais intuitivas e alinhadas à lógica do problema que se busca resolver (MERNIK; HEERING; SLOANE, 2005).

A implementação de sistemas computacionais voltados a domínios específicos tem se tornado cada vez mais complexa, exigindo a integração de múltiplas áreas do

² Site da Microchip sobre o AVR16DD20: <<https://www.microchip.com/en-us/product/AVR16DD20>>

³ DataSheet sobre STM32F103x8 e STM32F103xB <<https://www.st.com/resource/en/datasheet/stm32f103r8.pdf>>

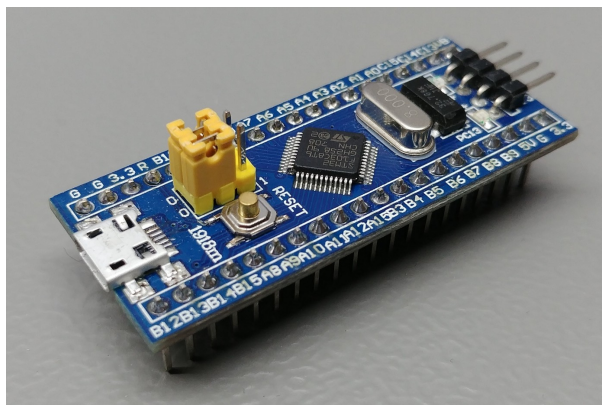


Figura 4 – A imagem apresenta uma placa que utiliza um microcontrolador STM32, desenvolvido pela STMicroelectronics. O modelo em destaque é o STM32F103C8T6, que opera com frequência máxima de 72 MHz, possui 20 kB de memória SRAM, 128 kB de memória Flash e oferece até 37 portas de entrada e saída. Além do microcontrolador, a placa inclui outros componentes essenciais para seu funcionamento, como o oscilador, o circuito de alimentação e os conectores laterais que permitem o acesso às portas de I/O. [STMicroelectronics \(2023\)](#)

conhecimento. Um exemplo claro é o desenvolvimento de sistemas web modernos, que demanda atenção simultânea a aspectos como usabilidade, segurança, persistência de dados e regras de negócio. Para lidar com essas diferentes preocupações de forma mais estruturada e independente da tecnologia de codificação utilizada, engenheiros de software vêm recorrendo às Linguagens Específicas de Domínio (DSLs), como aponta [Fowler \(2010\)](#). Essas linguagens são amplamente adotadas para modelar e codificar funcionalidades de um domínio específico ([IUNG et al., 2020](#)), no caso deste trabalho, no domínio dos microcontroladores.

A *Robotics Language* é uma linguagem de programação desenvolvida com foco no domínio de microcontroladores aplicados à robótica e IoT. Seu principal diferencial é a abstração das particularidades do hardware diretamente no compilador e na biblioteca padrão, permitindo que o desenvolvedor escreva um único código que funcione em diferentes plataformas sem precisar de adaptações manuais ou diretivas específicas ([OLIVEIRA, 2024](#)).

Além de facilitar a portabilidade, a linguagem realiza uma análise semântica mais rica, prevenindo erros que ocorreriam em tempo de execução, ainda na etapa de compilação. Sua construção foi feita utilizando ferramentas clássicas de compiladores como o Flex⁴ (versão 2.6.4) para a análise léxica, Bison⁵ (versão 3.8.2) para a análise sintática e o LLVM⁶ para o *back-end*, este último um *framework* moderno e modular que permite tanto a otimização quanto a geração de código para diferentes arquiteturas de microcontroladores.

⁴ Site do Flex: <https://github.com/westes/flex>

⁵ Site do Bison: <https://www.gnu.org/software/bison/manual/>

⁶ Site do LLVM: <https://www.llvm.org/>

O ecossistema de ferramentas da linguagem inclui suporte à depuração em simulador e realce de sintaxe no *Visual Studio Code* (a IDE ideal para o desenvolvimento) por meio da extensão *RobCmpSyntax*, o que torna o desenvolvimento mais visível e intuitivo, especialmente em ambientes educacionais. Ao utilizar a análise semântica do compilador, é possível evitar erros recorrentes que costumam surgir durante o desenvolvimento de firmware em linguagens de propósito geral, como C ou C++, justamente por essas linguagens não serem adaptadas às particularidades do domínio de aplicação.

Embora as DSLs, como a *Robotics Language*, simplifiquem o desenvolvimento ao abstrair detalhes de hardware e otimizar a portabilidade, seu potencial pode ser ampliado com técnicas de Inteligência Artificial, conforme evidenciado no trabalho de [Wei et al. \(2025\)](#). Modelos de IA, especialmente os de aprendizado de máquina, podem explorar o código gerado por DSLs para identificar padrões e aplicar otimizações automáticas, unindo a expressividade da linguagem à capacidade adaptativa de algoritmos inteligentes.

2.5 Inteligência Artificial

A inteligência artificial (IA) busca capacitar sistemas computacionais a realizar tarefas que, tradicionalmente, associamos à mente humana. Isso inclui tanto habilidades comumente consideradas “inteligentes”, como o raciocínio lógico, quanto funções como visão ou controle motor, que também envolvem processos cognitivos importantes ([MORANDÍN-AHUERMA, 2022](#)).

Em vez de representar um único tipo de habilidade, nossa inteligência abrange um conjunto complexo e estruturado de capacidades distintas voltadas ao processamento de informações. De forma análoga, a IA combina diferentes métodos e abordagens para lidar com uma ampla gama de desafios, adaptando-se conforme o tipo de problema a ser resolvido ([BODEN, 2017](#)). Além disso, as IAs são baseadas em algoritmos e tecnologias de aprendizagem automático para que a máquina tenha a capacidade de realizar habilidades cognitivas e tarefas sozinhas, seja de maneira autônoma ou semi-autônoma ([MORANDÍN-AHUERMA, 2022](#)).

2.5.1 Aprendizado de Máquina

Como explicado por [Zhou \(2021\)](#), o aprendizado de máquina é uma abordagem dentro da computação que permite que sistemas melhorem seu desempenho a partir da experiência, utilizando métodos computacionais. Nesse contexto, a “experiência” se traduz em dados, e o principal objetivo dessa área é desenvolver algoritmos capazes de extrair padrões desses dados para construir modelos preditivos (resultado aprendido a partir dos dados).

O aprendizado de máquina pode ser categorizado em três tipos principais: supervisionado, não supervisionado e por reforço (LUDERMIR, 2021). No aprendizado supervisionado, o algoritmo recebe um conjunto de exemplos de treinamento rotulados (com a resposta esperada) e aprende a associar as entradas às saídas corretas, tornando-se capaz de prever resultados em novos dados cujo rótulo é desconhecido. Já no aprendizado não supervisionado, o algoritmo trabalha com dados não rotulados e procura por padrões ocultos ou agrupamentos (*clustering*) nesses dados, sem conhecimento prévio das categorias, descobrindo estruturas subjacentes de forma autônoma. Por fim, no aprendizado por reforço, um agente aprende a tomar decisões através de recompensas e punições recebidas ao interagir com um ambiente; o sistema reforça ações corretas e desencoraja as incorretas, visando maximizar uma recompensa cumulativa ao longo do tempo (RUSSELL; RUSSELL; NORVIG, 2020).

2.5.2 Treinamento supervisionado

No aprendizado supervisionado, são utilizados exemplos em que se conhece tanto a entrada quanto a saída corretas – como uma lista de pares (entrada, saída), tipo $((x_1, y_1), (x_2, y_2), \dots, (x_n, y_n))$ (MAHESH et al., 2020). Cada um desses pares foi gerado por alguma regra que relaciona entrada e saída; a regra exata pode ser desconhecida. A tarefa principal, então, é descobrir uma fórmula ou método capaz de imitar o comportamento dessa regra original o máximo possível. Em outras palavras, queremos que essa fórmula ou método consiga prever corretamente a saída y para uma nova entrada x (LUDERMIR, 2021). Alguns autores chamam essa fórmula ou método de função h , ou função hipótese, pois representa uma suposição sobre como o mundo funciona, construída com base nos exemplos que se tem.

Para avaliar corretamente um modelo de aprendizado, é necessário separar os dados de entrada em dois conjuntos: um para treinamento e outro para teste. O conjunto de treinamento serve como base para o modelo aprender, ou seja, é a partir desses exemplos que ele tenta identificar padrões. Já o conjunto de teste é usado posteriormente para verificar se o modelo consegue fazer boas previsões em dados que ele nunca viu antes, medindo assim seu desempenho (LUDERMIR, 2021).

Por exemplo, considere o cenário de uma instituição de ensino em que se deseja prever o desempenho dos estudantes. Um modelo de aprendizado de máquina poderia ser treinado com dados históricos de alunos, incluindo variáveis como notas em disciplinas anteriores, frequência às aulas e envolvimento em atividades acadêmicas. Com esses exemplos rotulados – por exemplo, dados de alunos formados indicando quais foram aprovados ou reprovados – o algoritmo aprenderia padrões que relacionam essas variáveis ao sucesso ou dificuldade acadêmica. Após o treinamento, o modelo seria capaz de prever quais alunos atuais estão sob risco de baixo desempenho ou reprovação, com base em seus dados

mais recentes, permitindo que a escola ou os professores realizem intervenções pedagógicas antecipadas. Esse exemplo ilustra uma aplicação típica de aprendizado supervisionado na educação, em que o sistema aprende com experiências passadas (dados de alunos anteriores) para tomar decisões preditivas no presente.

2.5.3 *Large Language Models* (LLMs)

Modelos de linguagem como o *Code Llama*⁷ e o Chat GPT⁸ (ZHAO et al., 2024) são sistemas computacionais capazes de entender e produzir texto em linguagem natural. Eles têm a capacidade de prever a probabilidade de certas sequências de palavras ou até mesmo gerar novos textos com base em uma entrada fornecida, o que os torna ferramentas poderosas e versáteis em diversas aplicações (CHANG et al., 2024).

Modelos de Linguagem de Grande Escala, conhecidos como LLMs, são versões avançadas dos modelos de linguagem, caracterizados por possuírem uma quantidade enorme de parâmetros e uma capacidade de aprendizado bastante sofisticada. O principal componente por trás do funcionamento desses modelos é o mecanismo de autoatenção presente na arquitetura *Transformer*, que se tornou a base para diversas tarefas em Processamento de Linguagem Natural (PLN). Os *Transformers* trouxeram uma mudança significativa para a área de PLN por conseguirem lidar de forma eficiente com dados sequenciais, permitindo paralelização no treinamento e capturando relações de longo alcance dentro dos textos (CHANG et al., 2024).

O diferencial da arquitetura *Transformer*, utilizada como base na maioria dos LLMs atuais, está no mecanismo de autoatenção. Em vez de depender de estruturas sequenciais como redes recorrentes, o *Transformer* é capaz de analisar todas as partes de uma sequência de entrada ao mesmo tempo, identificando relações entre palavras mesmo quando elas estão distantes no texto. Isso torna o treinamento mais rápido, facilita a paralelização e melhora a capacidade do modelo de capturar padrões complexos na linguagem (VASWANI et al., 2017).

Apesar de seu grande potencial, o treinamento de LLMs exige um volume enorme de recursos computacionais e de dados. Um exemplo disso é o *Code Llama*, cujo treinamento demandou cerca de 1,4 milhão de horas de GPU A100. Além disso, preparar e organizar conjuntos de dados com centenas de bilhões de *tokens* é uma tarefa bastante complexa. Esses custos acabam sendo um obstáculo para muitos pesquisadores, tornando inviável a reprodução ou expansão desses modelos em contextos com recursos mais limitados (CUMMINS et al., 2025).

Os LLMs, ao demonstrarem capacidade de compreender e transformar código, como apresentado no trabalho de tornam-se candidatos promissores para apoiar a otimização

⁷ Site do Code Llama: <<https://www.llama.com/code-llama>>

⁸ Site do ChatGPT: <<https://openai.com/index/chatgpt>>

em compiladores, como apresentado no trabalho de [Cummins et al. \(2025\)](#). Com essa base, vamos entender sobre o processo de otimização em compiladores na próxima seção.

2.6 Otimização em compiladores

A função do otimizador é aplicar transformações sobre o código em representação intermediária (IR), gerado pelo *front-end*, com o objetivo de melhorar a qualidade do código final que será emitido pelo *back-end*. Essa ideia de “melhora” pode variar bastante, dependendo do contexto. Na maioria dos casos, significa tornar o código executável mais rápido. No entanto, em outras situações, a otimização pode estar voltada para a redução do consumo de energia ou para diminuir o uso de memória. Todas essas metas fazem parte do domínio da otimização dentro de um compilador ([COOPER; TORCZON, 2012](#)).

2.6.1 Otimização Clássica – sem Inteligência Artificial

Até recentemente, encontrar uma tradução ótima era descartado como sendo algo difícil demais de se alcançar e um esforço irrealista ([WANG; O’BOYLE, 2018](#)). [Faustino et al. \(2021\)](#) nos dizem que compiladores disponibilizam ao desenvolvedor algumas sequências de otimização pré-configuradas, cada uma voltada para um objetivo específico, como melhorar o desempenho do código ou reduzir seu tamanho. No caso do LLVM, por exemplo, o otimizador do *middle-end*, conhecido como Opt, oferece três níveis padrão voltados para desempenho: *-O1*, *-O2* e *-O3*. Além disso, existem também dois níveis focados na redução do tamanho do código gerado: *-Os* e *-Oz* ([DENG et al., 2025](#)).

Cada uma dessas sequências ativa uma série de etapas de compilação, que podem incluir tanto análises quanto transformações sobre o código intermediário. Para se ter uma ideia, o nível *O1* no LLVM executa 229 passes de otimização; já o *-O2* aciona 277, e o *O3*, 281. Mesmo as sequências voltadas à economia de espaço não são simples: o Opt-*Os* realiza 264 passes, enquanto o Opt-*Oz* executa 260 ([FAUSTINO et al., 2021](#)).

2.6.2 Otimização Com Inteligência Artificial

Embora o uso de aprendizado de máquina (ML) para otimizações em compiladores já tenha sido bastante explorado em pesquisas acadêmicas, sua adoção em compiladores industriais (que exigem alto grau de robustez e confiabilidade) ainda é bastante limitada. Até o momento, essas técnicas não se consolidaram como parte integrante de compiladores amplamente utilizados na prática ([TROFIN et al., 2021](#)). Apesar dos avanços, muitos dos métodos propostos ainda falham em reproduzir até mesmo análises básicas de fluxo de dados (que são essenciais para orientar decisões de otimização mais eficazes) ([CUMMINS et al., 2021](#)).

O aprendizado de máquina pode ser aplicado dentro do compilador para construir um modelo capaz de tomar decisões de otimização automaticamente, independentemente do programa fornecido como entrada. Esse processo é dividido em duas etapas principais: aprendizado e aplicação. Na fase de aprendizado, o modelo é treinado com base em dados coletados de programas anteriores, enquanto na etapa de aplicação o modelo é utilizado para prever boas decisões em novos programas ainda não vistos. Para que isso seja possível, é fundamental representar os programas de forma sistemática por meio de propriedades mensuráveis, conhecidas como *features*. Essas *features* são essenciais, pois o aprendizado de máquina depende justamente de características numéricas extraídas do código para construir um modelo que generalize bem e consiga orientar o compilador em diferentes contextos (WANG; O'BOYLE, 2018).

Entre as abordagens exploradas em aprendizado de máquina para representação de programas, uma alternativa promissora tem sido a modelagem do código como um grafo. Nessa representação, instruções individuais são conectadas por relações de controle, dados e chamadas, permitindo que o modelo compreenda cada instrução em relação ao seu contexto local dentro do grafo. Essa forma de raciocínio relacional possibilita a criação de representações latentes mais expressivas, aproveitando a estrutura do programa como um todo. A ideia se aproxima do funcionamento das IRs já utilizadas por compiladores e lembra os métodos clássicos de análise de fluxo de dados (CUMMINS et al., 2021).

Por fim, Trofin et al. (2021) nos apresenta uma aplicação real de aprendizado de máquina no contexto de compiladores, o MLGO, uma estrutura projetada para integrar técnicas de ML de forma sistemática ao compilador LLVM. Como estudo de caso, foi explorada a substituição da heurística tradicional de *inlining* voltada para redução de tamanho por modelos treinados com algoritmos de aprendizado, especificamente *Policy Gradient* e *Evolution Strategies*. Essa abordagem conseguiu reduzir em até 7% o tamanho do código gerado, superando o desempenho da otimização -Oz padrão do LLVM. Além disso, o modelo mostrou boa capacidade de generalização, apresentando resultados positivos tanto em diferentes alvos reais quanto em versões futuras dos mesmos programas, mesmo após meses de evolução no desenvolvimento. Esse tipo de integração mostra como o uso de técnicas de ML pode complementar e, em alguns casos, aprimorar estratégias já consolidadas no *pipeline* de compilação.

2.6.3 Otimização com *Large Language Models*

Há um interesse crescente em modelos de linguagem de grande porte (LLMs) para tarefas de engenharia de software, incluindo geração, tradução e teste de código (CUMMINS et al., 2025). Desenvolvedores desejam uma solução universal que transforme programas de entrada em versões semanticamente equivalentes, mas mais eficientes, sem esforço manual. Por isso, encontrar automaticamente uma boa sequência de passes é

fundamental para melhorar a eficiência das otimizações feitas pelo compilador. Para que esse processo seja viável na prática, o algoritmo precisa ser capaz de gerar sequências eficazes em um tempo aceitável e funcionar bem com diferentes tipos de programas (DENG et al., 2025).

Uma alternativa promissora dentro do uso de modelos de linguagem para otimização de compiladores é o *CompilerDream*, apresentado por Deng et al. (2025), uma abordagem baseada em aprendizado por reforço com modelo de mundo. Diferentemente de métodos tradicionais que utilizam algoritmos de busca ou aprendizado com reforço sem modelo, o *CompilerDream* constrói uma simulação precisa do comportamento do compilador e treina um agente que aprende a aplicar sequências eficazes de passes de otimização. Em experimentos, o modelo demonstrou capacidade de reduzir o tamanho do código em diversos conjuntos de dados, superando o nível de otimização -Oz do LLVM em praticamente todos os *benchmarks* avaliados, com exceção de dois casos. Ele também apresentou desempenho superior ao algoritmo PPO e à busca aleatória, especialmente dentro de orçamentos de tempo semelhantes. Um dos destaques é sua generalização sem treinamento específico no domínio: por exemplo, no conjunto NPB (NASA Parallel Benchmarks, um conjunto de programas desenvolvido originalmente pela NASA para avaliar o desempenho de sistemas de computação paralela), o modelo alcançou uma redução adicional de 3% no tamanho do código mesmo com dados escassos, e nos testes com programas em Fortran e Objective-C também obteve bons resultados — chegando a reduzir o código em até 2,87 vezes em casos específicos.

Outra proposta recente e relevante é a do modelo com *feedback* gerado pelo compilador, apresentado por Grubisic et al. (2024), que utiliza LLMs para otimização de código em LLVM-IR. Nesse modelo, o LLM não apenas sugere passes de otimização, mas também prevê a contagem de instruções do código antes e depois da otimização. Em seguida, essas sugestões são validadas com compilação real, e o resultado é devolvido ao modelo como *feedback*, permitindo uma nova tentativa mais precisa. Essa abordagem permitiu um ganho adicional de 0,53% em relação ao nível de otimização -Oz, superando os 2,87% obtidos pelo modelo base. O modelo com *feedback* também demonstrou desempenho superior em exemplos onde o modelo original cometia erros, especialmente com poucas inferências. Entre as variantes avaliadas, o modelo *Fast feedback* foi o mais eficaz, destacando-se pela eficiência na iteração e pela capacidade de detectar e corrigir instruções incorretas geradas anteriormente.

Recentemente, Cummins et al. (2025) propuseram os modelos *LLM Compiler* e *LLM Compiler FTD*, desenvolvidos especificamente para tarefas relacionadas a compiladores. Esses modelos são derivados do *Code Llama*, mas foram adaptados para compreender representações intermediárias (IRs), código *assembly* e estratégias de otimização de compilação. O *LLM Compiler* foi treinado com um impressionante volume de 546 bilhões

de *tokens* de dados especializados em LLVM-IR e *assembly*, passando por um ajuste fino orientado por instruções para interpretar o comportamento do compilador. O *LLM Compiler FTD*, por sua vez, vai além ao incorporar mais 164 bilhões de *tokens* voltados para tarefas específicas como ajuste de flags e desassemblagem, totalizando 710 bilhões de *tokens* no processo de treinamento. Esse treinamento em múltiplas etapas permite que os modelos obtenham resultados robustos: em tarefas de otimização de tamanho de código, o *LLM Compiler FTD* atinge 77% do desempenho de uma busca por autotuning (melhor compilação obtida a partir da execução exaustiva de um conjunto amplo de sequência de passes de otimização), mas sem precisar de compilações adicionais. Já na tarefa de compilação reversa do código *assembly* x86_64 e ARM para LLVM-IR, o modelo atinge uma taxa de 14% de reconstruções exatas. Dado o nível de especialização alcançado, os modelos *LLM Compiler* se mostram como um interessante ponto de partida para o desenvolvimento deste trabalho, que busca adaptar essa abordagem ao contexto de microcontroladores e ambientes com restrições severas de recursos.

3 Trabalhos Relacionados

3.1 Introdução

Para levantar os trabalhos relacionados, foi adotada uma revisão narrativa da literatura (NRL), partindo de buscas iniciais em bases científicas (Google Scholar) com termos de busca voltados a compiladores, otimização de código e aprendizado de máquina. Seguiu-se explorando as referências e citações de tais artigos, para identificar o conjunto de trabalhos mais recente e relacionado.

Para a escolha de trabalhos correlatos, foram elegidos aqueles que, direta ou indiretamente, aportam a aplicação de métodos para otimização de tamanho de código no pipeline de compilação (em especial no LLVM), seja por ordenação de passes, *autotuning* ou por modelos de Inteligência Artificial.

3.1.1 Trabalhos Analisados

Nas subseções abaixo são descritos três trabalhos, ressaltando suas contribuições e limitações quanto ao escopo deste estudo.

3.1.2 New Optimization Sequences for Code-Size Reduction for the LLVM-Compilation Infrastructure

A ideia central do trabalho de [Faustino et al. \(2021\)](#) é substituir as sequências padrão de redução de tamanho do LLVM (-Os com 264 passes e -Oz com 260 passes) por sequências muito mais curtas (12–15 passes) obtidas por busca sistemática, mas que preservem boa eficiência na redução de binário e reduzam o tempo de compilação. Para descobrir essas sequências, os autores geram candidatos com um algoritmo genético (YaCoS¹), reduzem-nos por poda ([PURINI; JAIN, 2013](#)) e removem duplicatas, construindo uma “matriz de otimização” a partir de 15.000 funções do AnghaBench ([SILVA et al., 2021](#)) e 10.044 sequências de otimização; cada célula da matriz guarda o tamanho do código LLVM-IR resultante ao aplicar uma sequência a uma função. A avaliação é feita em nível de programa, nos 16 *benchmarks* do *SPEC CPU2017*², medindo tamanho final em número de instruções LLVM-IR após a criação do executável.

Uma das sequências propostas (Lim) gera binários 2,3% menores que -Os e apenas 2,5% maiores que -Oz, enquanto compila mais rápido: 140 s com Lim vs 224 s (-Os) e 210

¹ Disponível em <https://github.com/ComputerSystemsLaboratory/YaCoS>

² Disponível em: <https://www.spec.org/cpu2017/>

s (-Oz), o que corresponde a $1,4\text{--}1,6\times$ e $1,3\text{--}1,5\times$ mais rápido, respectivamente. Em casos específicos, como x264, as sequências Sum e Lim superam -Oz em 11% e -Os em 16% no tamanho do binário. Ao aplicar todas as sequências por função e reter o melhor resultado, a combinação vence os níveis padrão do LLVM em 7 de 16 programas do SPEC.

Em resumo, a pesquisa de [Faustino et al. \(2021\)](#), sem o uso de Inteligência Artificial, encontra uma lista com sequência de passes curtas que realmente mostram ter impacto significativo sobre os padrões de otimização já existentes no LLVM. Porém por ser uma busca exaustiva, é pesada, e não se adapta especificamente a cada programa.

3.1.3 CompilerDream: Learning a Compiler World Model for General Code Optimization

A ideia central do estudo de [Deng et al. \(2025\)](#) é treinar um modelo de mundo do compilador, isto é, um modelo que simula como os passes de otimização se comportam quando aplicados ao código. Em cima desse simulador, os autores treinam um agente de aprendizado por reforço que aprende quais passes aplicar e em que ordem para melhorar o código (por exemplo, reduzir tamanho do binário ou otimizar métricas de desempenho). Ou seja, o sistema cria um “laboratório virtual” do compilador e ensina um agente a planejar sequências de otimização eficazes antes de gastar tempo com experimentos reais e lentos no compilador “de verdade”.

Seguindo essa linha, os resultados de [Deng et al. \(2025\)](#) (avaliados como média geométrica da redução da contagem de instruções no LLVM-IR sobre o “-Oz”) mostram que, no *autotuning/CompilerGym*³, o agente treinado no modelo de mundo atinge $1,068\times$ em *cBench* (coleção de programas C). No cenário de *value prediction* (escolha de sequências dentre 50 candidatas), o método supera o estado da arte *Coreset-NVP* ([LIANG et al., 2023](#)) em 3/4 suítes: *cBench* $1,038\times$ vs $1,028\times$, *MiBench* $1,017\times$ vs $1,003\times$ (*MiBench* é uma suíte clássica de embarcados), NPB $1,140\times$ vs $1,085\times$ e empata em *CHStone* $1,101\times$ vs $1,101\times$ (*CHStone* reúne 12 programas C usados em *High-Level Synthesis*). Já no teste *zero-shot end-to-end*, o agente do *CompilerDream* supera o -Oz em todos, exceto dois *benchmarks*; justamente BLAS (rotinas padrão de álgebra linear) e Linux (*kernel*) aparecem como casos “já muito otimizados”, o que explica a menor margem de ganhos nesses conjuntos.

Em síntese, o estudo de [Deng et al. \(2025\)](#), nos mostra que é possível, com uso de Inteligência Artificial, automatizar a ordenação de passes visando redução de tamanho de código, porém, os experimentos também não são realizados para MCUs/STM32, portanto não replicável para ambientes com recursos de *hardware* limitados.

³ Disponível em: <<https://github.com/facebookresearch/CompilerGym>>

3.1.4 LLMCompiler: Foundation Language Models for Compiler Optimization

A ideia central de (CUMMINS et al., 2025) é treinar modelos de linguagem “fundacionais” para compiladores em 546 bilhões de *tokens* de LLVM-IR e *assembly*, e depois especializá-los para emular otimizações e escolher passes que reduzem tamanho de código (além de um segundo alvo de levantamento de *assembly* \rightarrow IR). O tamanho de código é medido de duas formas: contagem de instruções no IR e tamanho binário (soma das seções *.TEXT* + *.DATA* após gerar o objeto; *.BSS* fica de fora por não afetar o tamanho em disco). Para gerar dados e treinar o modelo a “pensar como o opt”, os autores aplicam listas aleatórias de passes (amostradas de um conjunto de 167 passes) a programas desotimizados, e fazem *fine-tuning* para que o LLM preveja o IR/[*assembly*] e o tamanho resultante.

Além disso, o modelo alcança 77% do potencial de um *autotuner* extensivo, mas sem precisar de milhares de compilações por programa. Em suma, o LLM Compiler mostra que modelos fundacionais especializados em IR/*assembly* podem ordenar passes e escolher flags para reduzir tamanho de código de forma *zero-shot*, com ganhos médios de 4% sobre o -Oz.

3.1.5 Resumo Comparativo

A tabela [tabela 1](#) nos mostra o resumo comparativo entre os trabalhos analisados nas seções anteriores, com a utilização de critérios elaborados durante a revisão da literatura.

Tabela 1 – Comparativo entre trabalhos relacionados

Trabalhos	Critérios					
	MCUs	Modelos	LLM	Fine-tuning	CPU	Otimiz. p/ tamanho
(FAUSTINO et al., 2021)	–	–	–	–	+	+
(DENG et al., 2025)	–	+	–	+	+	+
(CUMMINS et al., 2025)	–	+	+	+	+	+
Este trabalho	+	+	+	+	+	+

“+” indica presença do critério; “–” indica ausência.

Todos os trabalhos apresentam o critério **otimização para tamanho** de forma explícita, o que diferencia é o caminho, um utiliza de heurísticas e engenharia de passes (FAUSTINO et al., 2021), enquanto os outros utilizam modelos treinados que priorizam métricas de tamanho (CUMMINS et al., 2025; DENG et al., 2025). Da mesma forma, todos os trabalhos possuem o critério **CPU**, pois em todos os trabalhos, o *pipeline* é pensado e executado para ambientes de CPUs de propósito geral.

Quanto ao critério **Modelos**, só é presente naqueles trabalhos em que foi utilizado modelos de inteligência artificial na otimização para redução de tamanho. Neste caso Faustino et al. (2021) não contém este critério pois realizam busca sistemática para

descobrir sequências curtas que reduzem tamanho e tempo de compilação em LLVM. Porém, Cummins et al. (2025) e Deng et al. (2025) utilizam respectivamente, um modelo de linguagem e modelo de mundo com um agente (um programa que aprende por tentativa e erro) para redução de tamanho de código.

O critério **Fine-tuning**, é adotado como qualquer adaptação paramétrica de um modelo base guiada por um sinal de tarefa ou domínio. Nessa leitura, somente Cummins et al. (2025) e Deng et al. (2025) pontuam positivamente. Cummins et al. (2025), após expor o modelo a bilhões de *tokens* inicia um *fine-tuning* supervisionado por instruções, usando entradas e saídas desejadas (A variante FTD adiciona uma etapa extra focada em tarefas, aprofundando essa especialização). Deng et al. (2025) realizam o *fine-tuning* por reforço de um agente dentro de um modelo de mundo que imita o compilador. O agente ajusta sua política (sua regra de decisão) para aplicar passes que reduzem o código.

Por fim, apenas no trabalho de Cummins et al. (2025) é atingido o critério **LLM**, que é a utilização de LLMs para reduzir o tamanho em LLVM-IR. Isso nos mostra que a literatura permanece centrada em CPUs. A minha contribuição é trazer esse conjunto de técnicas para o domínio de MCUs, mantendo o foco em otimização para tamanho, demonstrando sua efetividade sob as restrições de hardware embarcado e atingindo o critério de **MCUs**(utilização destas técnicas em MCUs).

4 Metodologia e Experimentos

4.1 Classificação da Pesquisa

A presente pesquisa se caracteriza, quanto à natureza, como aplicada, pois visa o desenvolvimento de uma solução prática com potencial impacto em ambientes de compilação, com foco especial na arquitetura STM32, especificamente no microcontrolador STM32F103C8T6. Com relação aos objetivos, é exploratória, pois investiga o uso de modelos de linguagem de grande porte (LLMs) em tarefas de otimização de código voltadas para sistemas embarcados. Quanto aos procedimentos, trata-se de uma pesquisa experimental, uma vez que implementa, adapta e valida algoritmos em ambiente de teste controlado. A abordagem adotada é quantitativa, visto que os resultados serão analisados por métricas objetivas, como a contagem de instruções do código gerado. Por fim, quanto às fontes, a pesquisa também é classificada como bibliográfica, uma vez que se fundamenta em artigos e manuais técnicos previamente publicados.

4.2 Aspectos Metodológicos Gerais

A metodologia será composta por etapas sequenciais, conforme apresentado na [Figura 5](#). Nossa metodologia começa pela preparação dos dados, em seguida a utilização de um *parser* específico, depois o treinamento de modelo e por fim a avaliação do modelo treinado. Primeiramente, selecionamos os conjuntos de dados de programas em C, em seguida utilizamos um conversor(*parser*) para converter esses mesmos programas em C para a ROBL.

Após a conversão, compilamos os códigos em ROBL, mas agora aplicando sequências de otimizações LLVM provenientes da pesquisa de [Faustino et al. \(2021\)](#), e somente então os dados foram organizados em pares de entrada (*prompt*) e saída (resposta/*label*) para treinamento do modelo. Partimos então, para o ajuste fino do modelo *LLM Compiler*, adaptando-o à tarefa de otimização de código específico da plataforma STM32. Finalmente, realizamos a avaliação do modelo treinado, com códigos não utilizados no treinamento, comparando a saída do modelo com otimizações convencionais. Todas estas etapas serão descritas nas subseções subsequentes.

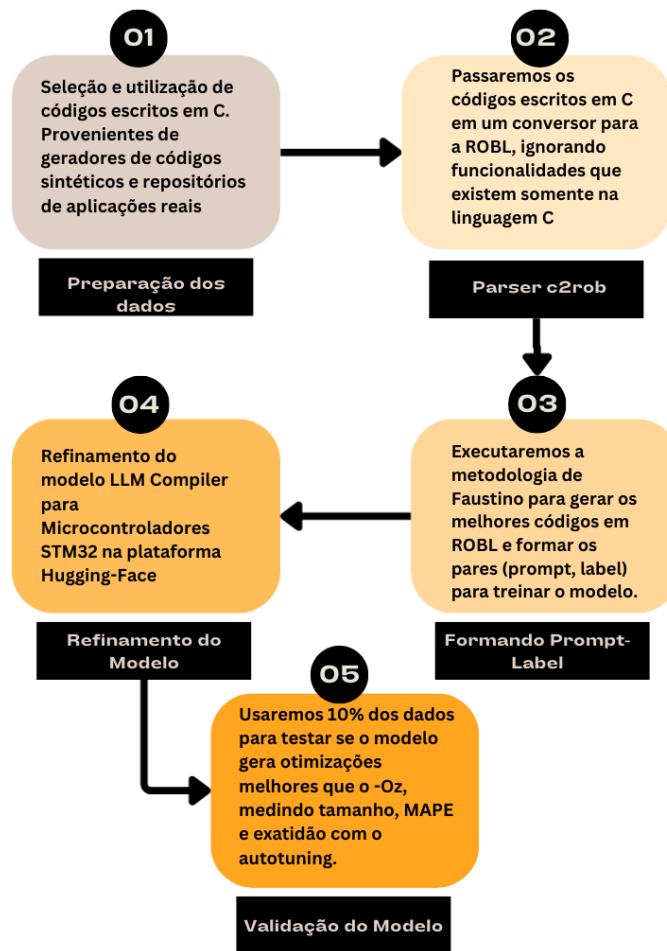


Figura 5 – Fluxograma de etapas metodológicas.

4.3 Conjunto de Dados

Para compor a base de entrada do pipeline experimental, utilizamos dois conjuntos de programas da linguagem C, visando abranger tanto códigos sintéticos aleatórios quanto códigos oriundos de aplicações reais.

O primeiro conjunto foi gerado pelo Csmith (YANG et al., 2011), uma ferramenta que produz programas aleatórios em C. O Csmith é comumente empregado para *stress testing* de compiladores, garantindo uma ampla diversidade de construções nos programas de entrada. Neste trabalho, a variabilidade dos códigos foi limitada pelas restrições de recursos na ROBL, como ausência de ponteiros, por exemplo. O Csmith possui um conjunto de parâmetros que pode ser usado para não emitir alguns recursos nos programas gerados, como o `-no-pointers` que previne a geração de ponteiros nos programas.

O segundo conjunto de programas é o AnghaBench (SILVA et al., 2021), uma suíte que contém aproximadamente um milhão de funções em C, extraídas de repositórios públicos do GitHub. Diferentemente do código sintético do Csmith, o AnghaBench provê

códigos representativos de padrões e estruturas encontrados em softwares do mundo real.

Essa combinação de *datasets* visou maximizar a diversidade do conjunto de treino. Códigos aleatórios exercitam aspectos incomuns da linguagem, enquanto códigos reais fornecem exemplos fielmente extraídos de aplicações práticas. Para utilizar estes *datasets* no nosso contexto, usaremos um conversor (`c2rob`) descrito na próxima seção.

4.4 Conversor de código C para RobCmp: `c2rob`

Para viabilizar a pesquisa, desenvolveu-se um conversor capaz de traduzir programas escritos em C para a sintaxe da ROBL, chamado `c2rob`.¹ A implementação foi realizada com auxílio das ferramentas Flex e Bison, escolhidas por sua capacidade de construir analisadores léxicos e sintáticos customizados.

O `c2rob` lê o código fonte em C, proveniente dos *datasets* descritos na subseção anterior, e o transforma em um código equivalente na sintaxe da ROBL, preservando a lógica e o comportamento do programa original.

Como nem todos os recursos da linguagem C estão presentes na ROBL, uma etapa adicional foi empregada para selecionar programas possíveis de conversão. A própria falha de tradução, apresentada pelo `c2rob` como erro de sintaxe, foi um indicador de incompatibilidade. Para garantir uma quantidade suficiente de programas corretos, criamos um *dataset* de aproximadamente 80.000 códigos C, e fizemos com que o `c2rob` ignorasse os arquivos que apresentassem erro de sintaxe. Os programas corretos foram compilados com a *flag* `-O1`, que aplica otimizações básicas e reduz o tamanho do código gerado, tornando-o mais adequado às limitações de contexto do modelo, ou seja, permite a criação de *prompts* menores mais que ainda possuem oportunidades de otimização.

4.5 Formação dos Dados de Treinamento

Para a criação dos dados de treinamento, é necessário que os pares de *prompt-label*, possuam em seu label uma versão compilada que se aproxime em eficiência da *flag* `-Oz`. Para isso foi empregado um processo de compilação, aproveitando uma lista de passes dos resultados do trabalho de Faustino et al. (2021), conhecida como *Optimization Cache*² que funcionam como um ótimo caso geral para os códigos, assim como a *flag* `-Oz`.

A lista de sequências de passes do trabalho de Faustino et al. (2021) não eram devidamente aplicáveis ao contexto atual do LLVM-20, usado neste trabalho, pois foi realizada usando a versão 10 do LLVM (lançada em meados de 2020). Inicialmente, apenas 30% das sequências eram utilizáveis. Portanto, mostrou-se necessário realizar a conversão

¹ Disponível em: <<https://github.com/thborges/c2rob>>

² Disponível em: <<https://zenodo.org/records/4416117>>

dos passes do LLVM-10 para o LLVM-20. Após esta conversão, conseguiu-se utilizar 100% da lista de passes para realizar a compilação.

Com os códigos devidamente compilados, procedeu-se à organização dos dados em pares de *prompt-label* para alimentar o treinamento supervisionado do modelo de linguagem. Em cada par, o *prompt* consiste de um comando inicial textual em inglês, seguido do código em LLVM IR para a plataforma alvo (ROBL compilado para LLVM IR, otimizado com a *flag -O1*), enquanto o *label* corresponde à melhor versão compilada do mesmo código em *assembly* e os respectivos passes de otimização empregados, seguindo o padrão adotado para o treinamento do modelo original (CUMMINS et al., 2025).

A lista de sequências de passes do trabalho de Faustino et al. (2021) não eram devidamente aplicáveis ao contexto atual do LLVM-20, usado neste trabalho, pois foi realizada usando a versão 10 do LLVM (lançada em meados de 2020). Inicialmente, apenas 30% das sequências eram utilizáveis. Portanto, mostrou-se necessário realizar a conversão dos passes do LLVM-10 para o LLVM-20. Após esta conversão, conseguiu-se utilizar 100% da lista de passes para realizar a compilação, composta por **1289 sequências únicas** de passes LLVM.

Com os códigos devidamente compilados, procedeu-se à organização dos dados em pares de *prompt-label* para alimentar o treinamento supervisionado do modelo de linguagem. Em cada par, o *prompt* consiste de um comando inicial textual em inglês, seguido do código em LLVM-IR para a plataforma alvo (ROBL compilado para LLVM-IR, otimizado com a *flag -O1*), enquanto o *label* corresponde à melhor versão compilada do mesmo código em *assembly* e os respectivos passes de otimização empregados, seguindo o padrão adotado para o treinamento do modelo original (CUMMINS et al., 2025).

O esquema de treinamento e inferência é apresentado na Figura 6. O *prompt* contém um texto em inglês perguntando qual os passes necessários para se reduzir o tamanho do arquivo objeto para a arquitetura stm32, seguido do código IR otimizado apenas com a flag -O1. A resposta (*label*) contém os passes de otimização necessários para reduzir o tamanho do arquivo objeto e o valor em decimal, da soma dos campos estruturais do mesmo arquivo, seguido do código assembly já otimizado com as sequências de Faustino et al. (2021). Durante a inferência, extraímos a lista de passes de otimização da resposta, que é, então, usada no compilador para produzir o código otimizado que foi usado em outras validações descritas na seção 5.1.

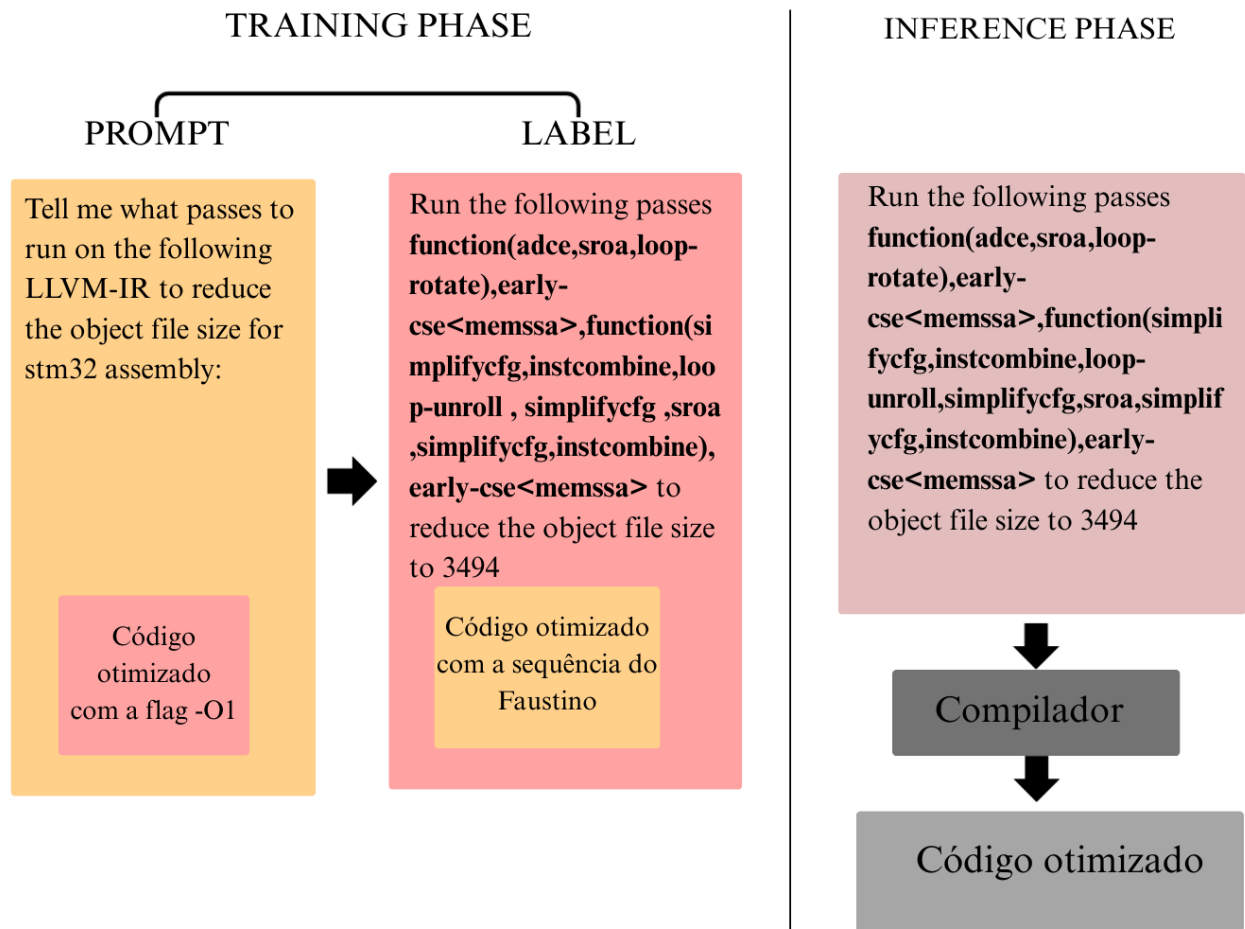


Figura 6 – Esquema de Treinamento e Inferência do LLM Compiler para STM32. Figura adaptada de Cummins et al. (2023)

4.6 Caracterização do Dataset quanto a Sequência de Passes

A Figura 7 apresenta a análise do conjunto de dados utilizado em relação à melhor otimização obtida utilizando a lista de sequências de passes do trabalho de Faustino et al. (2021). Observa-se que, em 56,2% dos programas, as sequências de passes propostas por Faustino et al. (2021) produzem códigos menores do que a *flag* padrão -Oz, enquanto em 43,6% dos casos o resultado é equivalente. Em apenas 0,2% a sequência é pior que o -Oz. Portanto, o conjunto de passes apresenta otimizações significativas que o modelo refinado pode capturar e aprender, mesmo não explorando todo o espaço de busca. A vantagem em relação ao -Oz é que a lista de passes é significativamente menor (dezenas *vs* centenas de passes).

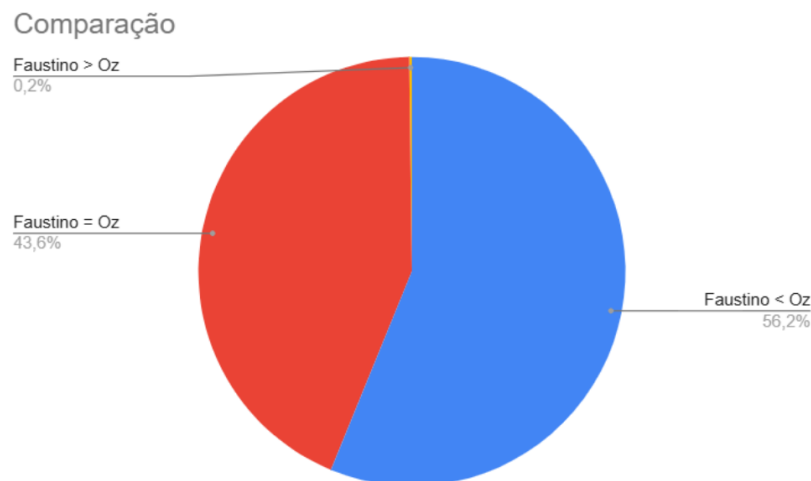


Figura 7 – Comparação de tamanho entre os códigos otimizados pelas sequências do (FAUSTINO et al., 2021) e os códigos otimizados pela *flag* -Oz.

4.7 Treinamento do Modelo

Com os *prompts* e *labels* preparados, procedemos ao refinamento do modelo para que ele aprenda a realizar otimizações de código da arquitetura STM32.

Devido ao volume de exemplos e ao tamanho do modelo, o treinamento exigirá recursos computacionais de alto desempenho. Optou-se por utilizar o ambiente de treinamento oferecido pela plataforma *Hugging Face*, que disponibiliza infraestrutura escalável com suporte a GPUs otimizadas, além de treinamento de *fine-tuning* automatizado (*auto-train*³), que dispensa codificação, através de sua interface de usuário. O *Hugging Face* permite o treinamento de modelos diretamente na nuvem, por meio de créditos computacionais, e oferece ferramentas integradas como monitoramento de desempenho, versionamento de experimentos e integração com repositórios. Essa alternativa se mostrou mais viável e flexível para o escopo do projeto.

O treinamento então foi realizado com 1,907 pares de *prompt-label*, e seu treinamento durou aproximadamente 60 horas. Para realizar o treinamento do modelo, deixamos a maioria dos parâmetros de treinamento nos padrões do próprio *hugging face Auto-Train*. De parâmetros similares ao de Cummins et al. (2025), conseguimos utilizar somente o *Cosine schedule* e também o *optimizer* AdamW, sem os valores β_1 e β_2 apresentados na mesma pesquisa, além disso, houve necessidade de alterar o valor do parâmetro *model max length* para 8,192 tokens. Estas limitações se devem ao fato de optarmos por utilizar a interface de usuário para treinamento do modelo, onde temos pouca liberdade de manipulação destes parâmetros, salvo o *model max length*, que precisamos alterar devido a GPU que escolhemos não suportar treinar o modelo com os 16,384 tokens como entrada. Mais detalhes sobre o ambiente de treinamento serão fornecidos na próxima seção.

³ Documentação do *auto-train*, disponível em: <<https://huggingface.co/docs/autotrain/index>>

4.8 Especificações de Hardware e Software

Quanto aos recursos utilizados, o *c2rob* foi desenvolvido e testado com auxílio do Visual Studio Code versão 1.101.0, no Ubuntu 24.04 LTS, executado em um *desktop* Lenovo ThinkCentre M93p, com processador Intel® Core™ i5-4570 (4 núcleos), 32GB de RAM e gráficos integrados Intel® HD Graphics 4600. Essa infraestrutura local foi utilizada principalmente nas etapas de geração, conversão, inferência experimental e manipulação dos dados, enquanto o treinamento do modelo propriamente dito ocorreu em ambiente de alto desempenho.

O ambiente de alto desempenho empregado, foi o *hugging face auto-train*, utilizando de uma GPU A10G Large, com 12vCPU, 46GB de memória, acelerador Nvidia A10G e 24GB de VRAM.

4.9 Resultados: Refinamento do Modelo

A partir do *pipeline* desenvolvido neste trabalho, foi possível construir um conjunto de 1,907 pares de *prompt-label*, para treinamento supervisionado, nos quais o *prompt* corresponde ao código em ROBL compilado para LLVM IR, otimizado apenas com a *flag* -O1, e o *label* representa a melhor sequência de passes de otimização, que resultou no menor código objeto para a arquitetura STM32. Esse conjunto foi gerado a partir de programas escritos originalmente em linguagem C, convertidos automaticamente para ROBL, compilados com diferentes sequências de passes e avaliados quanto ao tamanho gerado para o microcontrolador STM32F103C8T6.

O modelo *LLM Compiler* foi então refinado com esses pares, resultando em uma versão treinada⁴ para o domínio de programas em ROBL-alvo STM32. Embora o número de amostrar utilizadas no ajuste fino tenha sido reduzido, testes exploratórios de inferência indicaram que o modelo refinado é capaz de sugerir sequências de passes compatíveis com aquelas observadas no treinamento. Podemos observar isso no anexo E, que nos mostra o código ROB utilizado com a finalidade de criar o código IR para realizar inferência e o resultado apresentado pelo modelo.

No total foram realizados 5 testes de inferência exploratórios, sendo três do *benchmark* do *Csmith* (Anexos A, B e C) e dois do *benchmark* do *AnghaBench* (Anexos D e E). Para cada um deles, o código LLVM IR foi fornecido ao modelo por meio do *script* de inferência⁵, registrando-se as respostas em terminal. Nos testes com programas do *Csmith* e em dois dos três programas do *AnghaBench*, o modelo não recuperou as sequências de

⁴ Versão treinada do *LLM Compiler*, disponível em: <<https://huggingface.co/Cal-mfbc5446/LlmCompiler-Stm32FineTuningFinal>>

⁵ Código de inferência disponível em: <https://github.com/Alisson-Teles/Pipeline-training/blob/main/STM32/Inferencia/llm_compiler_demo.py>

passes fornecidas durante o processo de *auto-tunning*, o que sugere generalização ainda limitada, compatível com o tamanho reduzido do conjunto de treinamento. Em apenas um dos códigos (Anexo E) do *AnghaBench* observou-se a geração de uma sequência de passes estruturalmente semelhante àquela empregada durante o treinamento, indicando que o modelo é capaz de, em alguns casos, reproduzir padrões de otimização previamente observados.

A partir do *pipeline* desenvolvido neste trabalho, foi possível construir um conjunto de 1,907 pares de *prompt-label*, para treinamento supervisionado, nos quais o *prompt* corresponde ao código em ROBL compilado para LLVM-IR, otimizado apenas com a *flag* -O1, e o *label* representa a melhor sequência de passes de otimização, que resultou no menor código objeto para a arquitetura STM32. Esse conjunto foi gerado a partir de programas escritos originalmente em linguagem C, convertidos automaticamente para ROBL, compilados com diferentes sequências de passes e avaliados quanto ao tamanho gerado para o microcontrolador STM32F103C8T6.

O modelo *LLM Compiler* foi então refinado com esses pares, resultando em uma versão treinada⁶ para o domínio de programas em ROBL-alvo STM32. Embora o número de amostras utilizadas no ajuste fino tenha sido reduzido, testes exploratórios de inferência indicaram que o modelo refinado é capaz de sugerir sequências de passes compatíveis com aquelas observadas no treinamento. Podemos observar isso no anexo E, que nos mostra o código ROB utilizado com a finalidade de criar o código IR para realizar inferência e o resultado apresentado pelo modelo.

No total foram realizados 5 testes de inferência exploratórios, sendo três do *benchmark* do *Csmith* (Anexos A, B e C) e dois do *benchmark* do *AnghaBench* (Anexos D e E). Para cada um deles, o código LLVM-IR foi fornecido ao modelo por meio do *script* de inferência⁷, registrando-se as respostas em terminal. Nos testes com programas do *Csmith* e em dois dos três programas do *AnghaBench*, o modelo não recuperou as sequências de passes fornecidas durante o processo de *auto-tunning*, o que sugere generalização ainda limitada, compatível com o tamanho reduzido do conjunto de treinamento. Em apenas um dos códigos (Anexo E) do *AnghaBench* observou-se a geração de uma sequência de passes estruturalmente semelhante àquela empregada durante o treinamento, indicando que o modelo é capaz de, em alguns casos, reproduzir padrões de otimização previamente observados.

⁶ Versão treinada do *LLM Compiler*, disponível em: <<https://huggingface.co/Cal-mfbc5446/LlmCompiler-Stm32FineTuningFinal>>

⁷ Código de inferência disponível em: <https://github.com/Alisson-Teles/Pipeline-training/blob/main/STM32/Inferencia/llm_compiler_demo.py>

4.10 Resultados: Ferramentas para Geração do Dataset de Treinamento

O principal resultado deste trabalho é a implementação de um pipeline reprodutível⁸ que vai desde a coleta e conversão de códigos fonte em C para ROBL, passando pela seleção da melhor sequência de passes, até a construção dos pares de *prompt-label* e o ajuste fino do modelo. Este pipeline está disponível publicamente para acesso de todos.

O principal resultado deste trabalho é a implementação de um *pipeline* reprodutível⁹ que vai desde a coleta e conversão de códigos fonte em C para ROBL, passando pela seleção da melhor sequência de passes, até a construção dos pares de *prompt-label* e o ajuste fino do modelo. O pipeline começa com o processo de compilação exaustiva do *script* “`rodar_seq_unica.sh`”, que, ao receber como entrada o arquivo “`sequencias_unicas.txt`” (arquivo que contém as 1289 sequências de otimização do LLVM-IR), e o diretório contendo os arquivos LLVM-IR compilados em ROBL para STM32 realiza os seguintes pontos:

1. Aplicação dos passes presentes dentro de `sequencias_unicas.txt` ao arquivo LLVM-IR do código de entrada;
2. Geração do LLVM-IR otimizado com a melhor sequência de passes (aquela que produziu o melhor arquivo objeto);
3. Geração do melhor arquivo objeto correspondente (aquele que possui o menor tamanho);
4. Geração para STM32 *assembly*;
5. Extração e armazenamento do DEC (tamanho em bytes do melhor código objeto);
6. Armazenamento dos artefatos referentes à melhor sequência parcial (arquivos `.s`, `.o` e `.ll` e `.txt`), onde o `melhor.txt` contém a sequência que produziu o menor código;
7. Salva os melhores DEC de todos os códigos em um `melhores.csv` (apenas quando não executado de forma paralela);
8. Gera um diretório com os melhores artefatos para cada código em que as sequências foram aplicadas. Todos os artefatos são salvos em um subdiretório com o nome do código que foi otimizado.

Devido ao grande volume do dataset esperado, criamos um *script* que consegue realizar a execução paralela do código `rodar_seq_unica.sh`, este artefato permite a

⁸ Acesso do pipeline reprodutível: <<https://github.com/Alisson-Teles/Pipeline-training.git>>

⁹ Acesso do pipeline reprodutível: <<https://github.com/Alisson-Teles/Pipeline-training.git>>

execução em múltiplas *threads* (quantas você houver disponível em seu computador, no nosso caso foram nove). Porém, quando utilizado a execução paralela, o DEC de todos os códigos não é salvo, para resolver este problema, criamos o `coleta_melhores.py`, que recebe como entrada o diretório criado pelo `rodar_seq_unica.sh`, e coleta o DEC de cada `melhor.o` presente dentro de cada subdiretório e salvando em um `melhores.csv`.

Após gerar todos os dados necessários, utilizamos o código `promptLavel.py` para criação dos pares de *prompt-label*. O *script* realiza varredura em todos os diretórios que armazenam os arquivos LLVM-IR, as sequências consideradas ótimas (`melhor.txt`) e os respectivos arquivos de *assembly* otimizados, integrando essas informações e gerando como saída um arquivo JSONL. Nesse arquivo, cada linha é estruturada em uma coluna “*messages*”, seguida de “*content*”, no formato esperado pelo serviço de *autotrain* do *Hugging Face*, de modo que cada linha corresponde a um par de dados.

Para completar esse par de dados, o campo “*content*”, possui duas variações, uma pertencendo ao “*user*” e outra ao “*assistant*”, sendo respectivamente representadas pelo *prompt* e *label*, exatamente como apresentado no campo de “*Training Phase*” da figura 6.

Esse processo produz um conjunto de dados organizado e padronizado, já pronto para ser utilizado na etapa de *fine-tuning* do *LLM Compiler*. Dessa forma, assegura-se tanto a consistência em relação à metodologia original quanto a aderência ao domínio específico da arquitetura STM32.

Portanto os resultados apresentados nesse trabalho também *scripts*:

- O script `rodar_seq_uniq.sh` aplica, para cada programa LLVM-IR, as 1289 sequências de otimização únicas derivadas de (FAUSTINO et al., 2021). Para cada sequência, o código é compilado para STM32, sendo gerados e armazenados os arquivos resultantes (`.s`, `.o`, `.ll`). Ao final, para cada programa, identifica-se a sequência que produz o menor arquivo objeto, caracterizando o “melhor caso”.
- O script `parallel-run.py` atua distribuindo a execução do `rodar_seq_uniq.sh` entre múltiplos núcleos da CPU, possibilitando o processamento simultâneo de diversos programas e reduzindo de forma significativa o tempo total necessário para a conclusão das compilações.
- O script `coleta_melhores.py` percorre todas os subdiretórios de resultados geradas pelas compilações e, para cada programa, seleciona o arquivo objeto `melhor.o` e registra seu tamanho em *bytes*. Esses dados são então agregados em um único arquivo `melhores.csv`, estruturado como uma tabela que relaciona o nome de cada programa ao respectivo tamanho após otimização.
- O script `promptLavel.py` é encarregado da construção dos pares de treinamento, reunindo o IR não otimizado, o *assembly* otimizado, a sequência de passes considerada

ótima e o tamanho final do objeto. Essas informações são organizadas no formato JSONL adotado pelo *LLM Compiler*.

4.11 Discussão e Limitações

Este estudo foi conduzido sobre restrições de tempo e de orçamento computacional, o que impactou diretamente o escopo do projeto. A etapa de geração de dados, que envolve a compilação exaustiva dos mesmos programas sobre diferentes sequências de passes se mostrou custosa em termos de tempo de CPU e armazenamento. Em função destas limitações, optou-se por restringir o *auto-tuning* a um subconjunto controlado de combinações de passes e a um número reduzido de programas, priorizando a viabilidade da coleta e a completude do *pipeline* mesmo que isso implicasse em não explorar de forma exaustiva todo o espaço de combinações de passes de otimização.

Como consequência direta deste cenário, o modelo *LLM Compiler* foi refinado com 1,907 pares *prompt-label*, número inferior a estimativa inicial deste projeto e também a números observados em trabalhos correlatos, que utilizam ordens de grandeza superiores de dados e infraestrutura de processamento especializada para treinar modelos para otimização de código. Essa limitação de escala restringe a capacidade de generalização do modelo, especialmente para programas que se afastam do padrão observado no conjunto de treinamento.

Os resultados obtidos, ainda que limitados em escala, sugerem que a abordagem de refinamento de LLMs voltados a compiladores é aplicável ao contexto de microcontroladores STM32 e à linguagem específica de domínio Robotics Language. O fato de o modelo ser capaz de recuperar, para um programa não utilizado no treinamento, sequências de passes compatíveis com aquelas descobertas pelo [Faustino et al. \(2021\)](#) indica que ele consegue internalizar, ao menos de forma esporádica, padrões de otimização associados à redução de tamanho do código objeto. Esse comportamento está em linha com observações de trabalhos recentes que exploram LLMs especializados em IRs de compiladores e *assembly* para tarefas de otimização([CUMMINS et al., 2025](#); [DENG et al., 2025](#)).

Por outro lado, a ausência de uma avaliação quantitativa abrangente, baseada em métricas como redução média de *bytes* no objeto ou comparação sistemática com a sequência padrão -Oz, impede de afirmar que o modelo refinado supera consistentemente as estratégias tradicionais em cenários reais de desenvolvimento embarcado. Trabalhos voltados à descoberta de novas sequências compactas de passes para redução de código mostram que, mesmo em arquiteturas de propósito geral, o ganho em relação às *flags* padrão tende a ser modesto e depende fortemente da seleção de *benchmarks* e da profundidade da busca([CUMMINS et al., 2025](#); [FAUSTINO et al., 2021](#)). No presente estudo, essa exploração foi necessariamente restrita pelo tempo disponível e pelo custo computacional.

Outro ponto a ser destacado é que o conjunto de programas utilizados (Códigos C convertidos para ROBL) tende a abranger exemplos curtos (como é o caso do *AnghaBench*) ou com pouca variabilidade (caso do *Csmith*). Isso significa que o modelo foi exposto principalmente a padrões de códigos com pouca generalização, o que limita sua capacidade de lidar com projetos mais complexos. Apesar disso, o *pipeline* proposto estabelece uma base sólida para a qual o conjuntos de dados mais amplos e diversos podem ser agregados em trabalhos futuros, permitindo avaliar de forma mais robusta o potencial da abordagem em cenários de maior complexidade.

O conjunto de programas utilizados (Códigos C convertidos para ROBL) tende a abranger exemplos curtos (como é o caso do *AnghaBench*) ou com pouca variabilidade (caso do *Csmith*). Isso significa que o modelo foi exposto principalmente a padrões de códigos com pouca generalização, o que limita sua capacidade de lidar com projetos mais complexos. Apesar disso, o *pipeline* proposto estabelece uma base sólida para a qual o conjuntos de dados mais amplos e diversos podem ser agregados em trabalhos futuros, permitindo avaliar de forma mais robusta o potencial da abordagem em cenários de maior complexidade.

Outra limitação relevante, em relação ao tamanho do contexto de LLMs em geral, diz respeito ao tamanho médio dos pares de *prompt-label* que ficou em torno de 50,000 tokens. Esse volume de contexto é substancialmente maior do que o utilizado por Cummins et al. (2025), que treina o modelo com uma janela de 16,384 *tokens*. Na prática, isso significa que o conjunto construído neste trabalho não pôde ser explorado integralmente com a arquitetura de LLM utilizada nesta pesquisa, sendo realizado o truncamento dos dados após exceder o limite de contexto. Por outro lado, optou-se por manter os *prompts* completos justamente porque já existem modelos com janelas de 128,000 *tokens*¹⁰, o que preserva a utilidade das ferramentas de construção do *dataset* para uso em trabalhos futuros.

Em última análise, o cenário apresentado na figura 7 configura uma ameaça à validade externa dos experimentos, na medida em que as conclusões obtidas tendem a se restringir a situações muito específicas, caracterizadas por programas curtos e com pouca variabilidade. Em contextos mais realistas, envolvendo códigos de maior porte, com múltiplos módulos, bibliotecas externas e uso intensivo de periféricos, é pouco provável que o mesmo padrão de ganho se repita de forma consistente. Assim, embora os resultados indiquem que o modelo é capaz de reproduzir sequências competitivas em um cenário controlado, não se pode afirmar que o comportamento observado será mantido quando aplicado a sistemas embarcados de complexidade superior, o que deve ser explicitado como limitação e ameaça à generalização dos resultados.

¹⁰ Modelo Gemma 3 possui 128,000 de janela de contexto. Documentação disponível em: <https://ai.google.dev/gemma/docs/core?hl=pt-br>

5 Conclusão e trabalhos futuros

Este trabalho teve como objetivo refinar o modelo *LLM Compiler* para o contexto de otimização de código alvo voltado a microcontroladores STM32, tomando como base programas escritos em Robotics Language e o compilador Robcmp. Para isso, foi proposto e implementado um pipeline que abrangeu a conversão de código C para ROBL, a geração de diferentes códigos otimizados com diferentes sequências, a seleção do menor código com a melhor sequência para cada programa com base no tamanho do código objeto produzido e, por fim, a construção de pares *prompt-label* para treinamento supervisionado do modelo.

Consideramos que a principal contribuição deste trabalho é a infraestrutura de geração de datasets e o método de integração entre a Linguagem específica de domínio Robotics Language. Tal infraestrutura poderá ser reutilizada em trabalhos futuros, aumentando a quantidade e melhorando a diversidade dos programas bases, resultando assim em datasets melhores para o refinamento de modelos com o propósito de otimização de programas para microcontroladores.

Embora o modelo refinado tenha sido treinado com um conjunto pequeno, cerca de 1,907 pares, os experimentos de inferência realizados indicaram que ele é capaz de reproduzir, para códigos específicos, as sequências de passes observadas no processo de *auto-tunning*, demonstrando viabilidade técnica da abordagem e confirmando a adequação do fluxo de preparação de dados proposto.

5.1 Trabalhos Futuros

Como continuidade natural deste trabalho, a primeira linha de investigação futura consiste em ampliar e aperfeiçoar o conjunto de dados de treinamento. Isso inclui tanto o número de programas usados na criação do *dataset* de treinamento, quanto a adoção de *datasets* mais representativos de aplicações reais em sistemas embarcados, evitando a predominância de códigos muito curtos e com pouca variabilidade em seus conteúdos.

Uma segunda vertente importante de trabalhos futuros diz respeito à validação do modelo. Para a validação do modelo treinado, devem ser reservados 10% do total de amostras de código LLVM IR nunca vistas durante o treinamento. O objetivo é verificar a capacidade de generalização do modelo treinado, sua eficácia em realizar otimizações sobre códigos voltados à arquitetura STM32 e também realizar uma média de quantos dos códigos realmente fornecem sequências que reduzirão o tamanho do código. A principal métrica de desempenho será a redução percentual no tamanho do código final em comparação com o que seria gerado pelo compilador usando a *flag* padrão *-Oz*, replicando a abordagem

descrita por Cummins et al. (2025). Para isso, deve ser usada a métrica **MAE** (Erro Absoluto Médio), que avalia o quão precisa foi a estimativa do modelo sobre o tamanho do código gerado após otimização. Isso ajuda a entender se o modelo está prevendo bem o impacto das mudanças que sugere.

Uma segunda vertente importante de trabalhos futuros diz respeito à validação do modelo. Para a validação do modelo treinado, devem ser reservados 10% do total de amostras de código LLVM-IR nunca vistas durante o treinamento. O objetivo é verificar a capacidade de generalização do modelo treinado, sua eficácia em realizar otimizações sobre códigos voltados à arquitetura STM32 e também realizar uma média de quantos dos códigos realmente fornecem sequências que reduzirão o tamanho do código. A principal métrica de desempenho será a redução percentual no tamanho do código final em comparação com o que seria gerado pelo compilador usando a *flag* padrão -Oz, replicando a abordagem descrita por Cummins et al. (2025). Para isso, deve ser usada a métrica **MAE** (Erro Absoluto Médio), que avalia o quão precisa foi a estimativa do modelo sobre o tamanho do código gerado após otimização. Isso ajuda a entender se o modelo está prevendo bem o impacto das mudanças que sugere.

Por fim, nossa ultima vertente está relacionada ao aproveitamento integral dos pares *prompt-label*, cerca de 50,000 *tokens*. Devido à limitação da janela de contexto do modelo utilizado neste estudo, exatamente 16,384, não foi possível explorar todo o conteúdo disponível em cada exemplo, impondo truncamentos e perda de informação. Com o avanço recente de modelos que suportam janelas de 64,000 e 128,000 *tokens*, o *dataset* construído poderá ser reutilizado sem truncamentos, explorando todo o conteúdo em cada exemplo.

Referências

- AHO, A. V.; SETHI, R.; ULLMAN, J. D. *Compiladores: Princípios, técnicas e ferramentas. LTC, Rio de Janeiro, Brasil*, p. 219–276, 2007. Citado 3 vezes nas páginas 19, 20 e 22.
- ARDUINO. 2025. Disponível em: <<https://www.arduino.cc/>>. Citado na página 13.
- AVR16/Datasheet. 2024. Citado na página 12.
- BLAUTH, P. *Linguagens formais e autômatos. Artmed Editora SA*, 2010. Citado 2 vezes nas páginas 18 e 19.
- BODEN, M. A. *Inteligencia artificial*. [S.l.]: Turner, 2017. Citado na página 25.
- CHANG, Y. et al. A survey on evaluation of large language models. *ACM transactions on intelligent systems and technology*, ACM New York, NY, v. 15, n. 3, p. 1–45, 2024. Citado na página 27.
- COOPER, K.; TORCZON, L. *Engineering a Compiler*. Morgan Kaufmann, 2012. (Morgan Kaufmann). ISBN 9780120884780. Disponível em: <<https://books.google.com.br/books?id=CGTOIAEACAAJ>>. Citado 6 vezes nas páginas 7, 17, 18, 19, 21 e 28.
- COSTA, R. H. P. et al. *Compiladores*. 2023. Citado na página 20.
- CUMMINS, C. et al. Programl: A graph-based program representation for data flow analysis and compiler optimizations. In: PMLR. *International Conference on Machine Learning*. [S.l.], 2021. p. 2244–2253. Citado 2 vezes nas páginas 28 e 29.
- CUMMINS, C. et al. *Large Language Models for Compiler Optimization*. arXiv, 2023. ArXiv:2309.07062 [cs]. Disponível em: <<http://arxiv.org/abs/2309.07062>>. Citado 2 vezes nas páginas 7 e 40.
- CUMMINS, C. et al. LLM Compiler: Foundation Language Models for Compiler Optimization. In: *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*. Las Vegas NV USA: ACM, 2025. p. 141–153. ISBN 979-8-4007-1407-8. Disponível em: <<https://dl.acm.org/doi/10.1145/3708493.3712691>>. Citado 14 vezes nas páginas 13, 14, 15, 27, 28, 29, 30, 34, 35, 39, 41, 46, 47 e 49.
- DENG, C. et al. Compilerdream: Learning a compiler world model for general code optimization. In: *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*. [S.l.: s.n.], 2025. p. 486–497. Citado 7 vezes nas páginas 13, 28, 30, 33, 34, 35 e 46.
- FAUSTINO, A. et al. New optimization sequences for code-size reduction for the llvm compilation infrastructure. In: *Proceedings of the 25th Brazilian Symposium on Programming Languages*. [S.l.: s.n.], 2021. p. 33–40. Citado 12 vezes nas páginas 7, 28, 32, 33, 34, 36, 38, 39, 40, 41, 45 e 46.
- FISCHER, C.; CYTRON, R.; LEBLANC, R. *Crafting a Compiler*. Addison-Wesley, 2010. (Crafting a compiler with C). ISBN 9780136067054. Disponível em: <https://books.google.com.br/books?id=G4Y_AQAIAAJ>. Citado 6 vezes nas páginas 17, 18, 19, 20, 21 e 22.

- FOLEISS, J. H. et al. Scc: Um compilador c como ferramenta de ensino de compiladores. In: *WEAC2009-Workshop Educação em Arquitetura de Computadores*. [S.l.: s.n.], 2009. p. 15–22. Citado na página 17.
- FOWLER, M. *Domain-specific languages*. [S.l.]: Pearson Education, 2010. Citado na página 24.
- GEORGIOU, K. et al. Less is more: Exploiting the standard compiler optimization levels for better performance and energy consumption. In: *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*. New York, NY, USA: Association for Computing Machinery, 2018. (SCOPES '18), p. 35–42. ISBN 9781450357807. Disponível em: <<https://doi.org/10.1145/3207719.3207727>>. Citado na página 13.
- GRUBISIC, D. et al. Compiler generated feedback for large language models. *arXiv preprint arXiv:2403.14714*, 2024. Citado na página 30.
- HONG, S.; PARK, G.; KIM, J.-S. Automated deep-learning model optimization framework for microcontrollers. *ETRI Journal*, Wiley Online Library, v. 47, n. 2, p. 179–192, 2025. Citado na página 12.
- HUGGING. 2025. Disponível em: <<https://huggingface.co/docs/hub/en/index>>. Citado na página 14.
- HUSSAIN, A. et al. Programming a microcontroller. *Int. J. Comput. Appl*, v. 155, n. 5, p. 21–26, 2016. Citado na página 22.
- INTEL®™. 2025. Disponível em: <<https://www.intel.com/content/www/us/en/products/sku/208921/intel-core-i71165g7-processor-12m-cache-up-to-4-70-ghz-with-ipu/specifications.html>>. Citado na página 12.
- IUNG, A. et al. Systematic mapping study on domain-specific language development tools. *Empirical Software Engineering*, Springer, v. 25, p. 4205–4249, 2020. Citado na página 24.
- LIANG, Y. et al. *Learning compiler pass orders using coreset and normalized value prediction*. 2023. 20746–20762 p. Citado 2 vezes nas páginas 13 e 33.
- LUDERMIR, T. B. Inteligência artificial e aprendizado de máquina: estado atual e tendências. *Estudos Avançados*, SciELO Brasil, v. 35, p. 85–94, 2021. Citado na página 26.
- MAHESH, B. et al. Machine learning algorithms-a review. *International Journal of Science and Research (IJSR)*. [Internet], v. 9, n. 1, p. 381–386, 2020. Citado na página 26.
- MASLOV, S.; GANESH, K. Code size optimization using the Intel® C/C++ Compiler. 2014. Citado na página 12.
- MERNIK, M.; HEERING, J.; SLOANE, A. M. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, ACM New York, NY, USA, v. 37, n. 4, p. 316–344, 2005. Citado na página 23.
- Microchip Technology Inc. *AVR16DD14 - 8-bit AVR Microcontroller with 16 KB Flash and eXtreme Low Power Technology*. 2024. Disponível em: <<https://www.microchip.com/en-us/product/AVR16DD14>>. Citado na página 23.

- MORANDÍN-AHUERMA, F. What is artificial intelligence? 2022. Citado na página 25.
- OLIVEIRA, T. B. d. *Robcmp: The Robotics Compiler*. 2024. <<https://github.com/thborges/robcmp/>>. Disponível em: <<https://github.com/thborges/robcmp/>>. Acesso em: 20 maio 2025. Citado 2 vezes nas páginas 15 e 24.
- PURINI, S.; JAIN, L. Finding good optimization sequences covering program space. *ACM Transactions on Architecture and Code Optimization (TACO)*, ACM New York, NY, USA, v. 9, n. 4, p. 1–23, 2013. Citado na página 32.
- RUSSELL, S.; RUSSELL, S.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. Pearson, 2020. (Pearson series in artificial intelligence). ISBN 9780134610993. Disponível em: <<https://books.google.com.br/books?id=koFptAEACAAJ>>. Citado na página 26.
- SILVA, A. F. D. et al. Anghabench: A suite with one million compilable c benchmarks for code-size reduction. In: IEEE. *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. [S.l.], 2021. p. 378–390. Citado 2 vezes nas páginas 32 e 37.
- STMicroelectronics. *STM32F103R8 - ARM Cortex-M3 32-bit MCU with 64 or 128 Kbytes Flash, 72 MHz CPU, motor control, USB and CAN*. 2023. Disponível em: <<https://www.st.com/resource/en/datasheet/stm32f103r8.pdf>>. Citado 3 vezes nas páginas 12, 23 e 24.
- TROFIN, M. et al. *MLGO: a Machine Learning Guided Compiler Optimizations Framework*. 2021. Disponível em: <<https://arxiv.org/pdf/2101.04808>>. Citado 2 vezes nas páginas 28 e 29.
- VASWANI, A. et al. Attention is all you need. *Advances in neural information processing systems*, v. 30, 2017. Citado na página 27.
- WANG, Z.; O’BOYLE, M. Machine learning in compiler optimization. *Proceedings of the IEEE*, v. 106, n. 11, p. 1879–1901, 2018. Citado 4 vezes nas páginas 12, 13, 28 e 29.
- WEI, A. et al. Improving parallel program performance with llm optimizers via agent-system interfaces. In: *Forty-second International Conference on Machine Learning*. [S.l.: s.n.], 2025. Citado na página 25.
- YANG, X. et al. Finding and understanding bugs in c compilers. In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. [S.l.: s.n.], 2011. p. 283–294. Citado na página 37.
- ZHAO, H. et al. Explainability for large language models: A survey. *ACM Transactions on Intelligent Systems and Technology*, ACM New York, NY, v. 15, n. 2, p. 1–38, 2024. Citado na página 27.
- ZHOU, Z.-H. *Machine learning*. [S.l.]: Springer nature, 2021. Citado na página 25.

Anexos

ANEXO A – Código 00668.rob, e Resultado da inferência

Este anexo apresenta um dos códigos em ROBL utilizado para criar o LLVM-IR e realizar a inferência, bem como o resultado apresentado pela inferência experimental.

A.1 Código em ROBL

```

1 use csmith;
2 __undefined = int64(0);
3
4 int8 func_1();
5
6 int8 func_1() {
7     l_4 = int8(9);
8     l_5 = {int32(0)};
9     l_38 = uint32(0x68CDE333);
10    l_45 = int16((-7));
11    l_72 = int8((-1));
12    l_88 = int32(0x97BD6D72);
13    l_95 = uint64(0x9A48E5C97493B576);
14    i = int32(0);
15
16    i = 0;
17    while i < 1 {
18        l_5[i] = (-1);
19        i++;
20    }
21
22    l_5[0] &= (safe_div_func_uint8_t_u_u(l_4, 0x1E));
23
24    if (((((((safe_mul_func_int32_t_s_s(l_5[0], l_5[0])) < l_5[0]) <= 0x1B)
25        & 6U) < 0x764276038591B68D) != 0) and (3 != 0)) != 0 {
26        l_8 = uint16(65535U);
27        l_25 = int32(0x7EA1B148);
28
29        l_4 = 0;
30        while (l_4 <= 0) {

```

```

30         l_24 = uint32(0x57617C0E);
31         i = int32(0);
32
33         if l_5[l_4] != 0 {
34             i = int32(0);
35             l_8 &= l_5[l_4];
36             return l_5[l_4];
37         } else {
38             l_23 = {int32((-2)), (-2), (-2), (-2), (-2)};
39             i = int32(0);
40             l_25 = (safe_mul_func_int16_t_s_s((
                    safe_mul_func_int16_t_s_s(((
                    safe_mod_func_uint32_t_u_u((((
                    safe_sub_func_int16_t_s_s(((
                    safe_add_func_uint32_t_u_u(((
                    safe_add_func_uint32_t_u_u((
                    safe_sub_func_uint32_t_u_u(((l_8 < l_8) | l_23
                    [3]), l_5[l_4])), l_8)) != 0) and ((-9) != 0)),
                    (-3))) != l_5[l_4]) ^ l_24), 0U)) > l_5[0])
                    != 0) or (l_24 != 0)), l_23[2])) & (-7)), l_8))
                    , l_5[0]));
41             l_5[l_4] = ((0x42796A450C197911 != 0) or (l_25 !=
                    0));
42         }
43
44         l_4 += 1;
45     }
46
47     l_5[0] |= l_8;
48
49     l_4 = (-14);
50     while (l_4 > 17) {
51         l_32 = int64((-1));
52         l_5[0] = ((safe_lshift_func_uint64_t_u_u(((((((
                    safe_div_func_int8_t_s_s(l_32, l_32)) ^ (-8)) | 0U) !=
                    l_32) > l_5[0]) != 0) and (l_32 != 0)), 21)) ^ l_5
                    [0]);
53         l_4 = safe_add_func_uint8_t_u_u(l_4, 8);
54     }
55
56     } else {
57         l_36 = uint32(1U);

```

```

58         l_46 = int32(8);
59         l_60 = {uint8(250U), 0U, 0U, 250U, 0U, 0U};
60         l_64 = int64((-4));
61         i = int32(0);
62
63         if (safe_div_func_uint32_t_u_u((~l_5[0]), l_36)) != 0 {
64             l_37 = int32(5);
65             l_39 = {int32(0), 0, 0, 0, 0};
66             l_42 = uint32(0x2DA72DF2);
67             i = int32(0);
68
69             l_36 = 0;
70             while (l_36 <= 0) {
71                 i = int32(0);
72                 l_5[l_36] = ((l_5[l_36] > l_4) & l_37);
73                 l_39[0] = ((l_37 != 0) and (l_38 != 0));
74                 l_5[0] = (((safe_lshift_func_int16_t_s_u(l_5[0],
75                     l_39[0])) < l_5[l_36]) & 0x71);
76                 l_5[0] = (l_39[0] ^ l_42);
77                 l_36 += 1;
78             }
79
80             if (safe_add_func_uint16_t_u_u((l_5[0] >= 9U), 0x6C34))
81                 != 0 {
82                 l_39[3] = l_5[0];
83                 l_45 ^= 0;
84                 l_46 ^= 0x971708C9;
85             } else {
86                 l_51 = uint32(18446744073709551615U);
87                 l_46 = ((safe_sub_func_uint64_t_u_u((
88                     safe_mul_func_uint16_t_u_u((8U <= 0
89                     x3943CDC182152669), 0x2DBC)), 0
90                     x293E9FC65F31FD02)) ^ l_39[3]);
91                 l_39[0] = l_51;
92                 l_5[0] = (((safe_mul_func_int16_t_s_s(((
93                     safe_add_func_int64_t_s_s((1 == l_36), l_51))
94                     == l_37), l_46)) ^ l_38) != 0) and (0x1071 !=
95                     0));
96             }
97         } else {

```

```

102         l_61 = uint32(18446744073709551610U);
103         l_83 = int32(0x03AF45EF);
104
105         if (safe_lshift_func_int32_t_s_s(l_38, l_5[0])) != 0 {
106             l_65 = uint8(1U);
107             l_61 = (safe_sub_func_uint8_t_u_u(l_60[2], l_45));
108             l_5[0] = (safe_sub_func_uint32_t_u_u(l_64, l_5[0])
109                 );
110             l_65--;
111             l_5[0] &= (-8);
112         } else {
113             l_68 = uint32(0xB7D65D20);
114             l_68 = 0;
115         }
116
117         if (safe_mul_func_uint32_t_u_u((0xB7 == 0x49), 0x14E7E319
118             )) != 0 {
119             l_71 = uint64(0x4DD42852A3B4A04E);
120             l_71 |= 0xE569BB21;
121         } else {
122             l_72 ^= l_61;
123         }
124
125         l_46 = (0x4C8A);
126
127         l_72 = 0;
128         while (l_72 == 6) {
129             l_78 = int64(0xB8759D8E95A6083B);
130             l_5[0] &= (safe_lshift_func_uint32_t_u_u((l_60[2]
131                 ^ l_61), l_61));
132             l_78 = l_38;
133             l_5[0] = (safe_div_func_uint32_t_u_u(0x90F62EB7,
134                 2));
135             l_83 = (safe_mul_func_uint16_t_u_u((6 > l_46), 1U)
136                 );
137             l_72 = safe_add_func_uint64_t_u_u(l_72, 1);
138         }
139     }
140 }

```



```
168         l_45 += 1;
169     }
170
171     l_91 ^= ((safe_rshift_func_uint16_t_u_s(0U, l_94)) ^ 0
172             x4C20F6EB);
173     l_72 += 1;
174 }
175     l_4++;
176 }
177
178     l_5[0] = (((l_45 <= l_45) ^ l_4) != l_95) > l_95);
179     return l_38;
180 }
181
182
183 int32 main() {
184     print_hash_value = int32(0);
185     platform_main_begin();
186     crc32_gentab();
187     func_1();
188     platform_main_end(crc32_context ^ 0xFFFFFFFFU, print_hash_value);
189     return 0;
190 }
```

Código 00668.rob

A.2 Resultado da Inferência

Saída gerada no terminal:

```
1 The LLVM-IR will have instruction count 33 and binary size 130 bytes:
2 <code>; Module ID = ' '
3 source_filename =
```

ANEXO B – Código 08016.rob, usado para criar o IR para inferência

Este anexo apresenta um dos códigos em ROBL utilizado para criar o LLVM-IR e realizar a inferência, bem como o resultado apresentado pela inferência experimental.

B.1 Código em ROBL

```

1 use csmith;
2 __undefined = int64(0);
3
4 int64 func_1();
5
6 int64 func_1() {
7     l_2 = {uint32(0)};
8     l_3 = int32((-1));
9     i = int32(0);
10
11     i = 0;
12     while i < 1 {
13         l_2[i] = 0x3342EB90;
14         i++;
15     }
16
17
18     l_3 = 0;
19     while (l_3 >= 0) {
20         l_4 = int32(0xAED23B61);
21
22         l_4 = 0;
23         while (l_4 >= 0) {
24             l_9 = int32((-1));
25             i = int32(0);
26             l_9 = (safe_sub_func_int64_t_s_s((
27                 safe_sub_func_int32_t_s_s(l_2[l_3], l_2[l_3])), 0
28                 x8F8F576C5165352F));
29             return l_3;
30             l_4 -= 1;

```

```
29         }
30
31         l_3 -= 1;
32     }
33
34     return l_2[0];
35 }
36
37
38 int32 main() {
39     print_hash_value = int32(0);
40     platform_main_begin();
41     crc32_gentab();
42     func_1();
43     platform_main_end(crc32_context ^ 0xFFFFFFFFU, print_hash_value);
44     return 0;
45 }
```

Código 08016.rob

B.2 Resultado da Inferência

Saída gerada no terminal:

```
1 The assembly will have instruction count 7 and binary size 142
2 <code> .text
3 .file "."
4 .globl S0:init #-- Begin function S0:init
5 .p2align 5,0x90
6 .type S0:init
```

ANEXO C – Código 45071.rob, usado para criar o IR para inferência

Este anexo apresenta um dos códigos em ROBL utilizado para criar o LLVM-IR e realizar a inferência, bem como o resultado apresentado pela inferência experimental.

C.1 Código em ROBL

```

1 use csmith;
2 __undefined = int64(0);
3
4 type S0{
5     f0 = uint6(0);
6 }
7
8
9 S0 func_1();
10
11 S0 func_1() {
12     l_4 = int32(0xF09CC25F);
13     l_5 = int32((-6));
14     l_6 = int32(1);
15     l_7 = int32(0xFAF425F5);
16     l_8 = {int32((-1)), 0x7CC42477, (-1), (-1), 0x7CC42477, (-1), (-1), 0
        x7CC42477, (-1)};
17     l_9 = int8(1);
18     l_10 = uint64(0x6FEC1B16EB2D3650);
19     l_13 = S0();
20     l_13.f0 = -0;
21
22     i = int32(0);
23     l_5 ^= (safe_lshift_func_int8_t_s_u((((0U | l_4) >= 1U) <= l_4) < 0),
        4));
24     l_10--;
25     return l_13;
26 }
27
28

```

```
29 int32 main() {
30     print_hash_value = int32(0);
31     platform_main_begin();
32     crc32_gentab();
33     func_1();
34     platform_main_end(crc32_context ^ 0xFFFFFFFFU, print_hash_value);
35     return 0;
36 }
```

Código 45071.rob

C.2 Resultado da Inferência

Saída gerada no terminal:

```
1
2
3 <code> .text
4
5 ‘‘S0:crc32\_gentab’’:
6
7         push\{r7, lr\}
8
9         mov r7,sp
10
11        bl 0xad8 <S0:crc32\_gentab+0x6> @ imm = \#-0x4
```

ANEXO D – Código

extr_.....gdbstub.c_gdb_cmd_memwrite, usado para criar o IR para inferência

Este anexo apresenta um dos códigos em ROBL utilizado para criar o LLVM-IR e realizar a inferência, bem como o resultado apresentado pela inferência experimental.

D.1 Código em ROBL

```

1
2 type kgdb_state{
3     dummy = int32(0);
4 }
5
6
7 int32 error_packet(int32 _un0, int32 _un1);
8 remcom_out_buffer = int32(0);
9
10 int32 strcpy(int32 _un2, char _un3);
11
12 int32 write_mem_msg(int32 _un4);
13
14 void gdb_cmd_memwrite(kgdb_state ks) {
15     err = int32(write_mem_msg(0));
16
17     if err != 0 {
18         error_packet(remcom_out_buffer, err);
19     } else {
20         strcpy(remcom_out_buffer, "OK");
21     }
22
23 }
```

Código extr_.....gdbstub.c_gdb_cmd_memwrite

D.2 Resultado da Inferência

Saída gerada no terminal:

```
1
2 <|im\_end|>
3
4 <|im\_start|>assistant
```

ANEXO E – Código

extr_libcoreproxy.c_compress_hint_to_enum usado para criar o IR para inferência

Este anexo apresenta um dos códigos em ROBL utilizado para criar o LLVM-IR e realizar a inferência, bem como o resultado apresentado pela inferência experimental.

E.1 Código em ROBL

```

1  H20_COMPRESS_HINT_AUTO = char(0);
2  H20_COMPRESS_HINT_DISABLE = char(0);
3  H20_COMPRESS_HINT_ENABLE = char(0);
4  H20_COMPRESS_HINT_ENABLE_BR = char(0);
5  H20_COMPRESS_HINT_ENABLE_GZIP = char(0);
6
7  int32 H20_STRLIT(char _un0);
8
9  int32 h2o_lcstris(char _un1, uint32 _un2, int32 _un3);
10
11 char compress_hint_to_enum(char val, uint32 len) {
12
13     if h2o_lcstris(val, len, H20_STRLIT("on")) != 0 {
14         return H20_COMPRESS_HINT_ENABLE;
15     }
16
17
18     if h2o_lcstris(val, len, H20_STRLIT("off")) != 0 {
19         return H20_COMPRESS_HINT_DISABLE;
20     }
21
22
23     if h2o_lcstris(val, len, H20_STRLIT("gzip")) != 0 {
24         return H20_COMPRESS_HINT_ENABLE_GZIP;
25     }
26
27
28     if h2o_lcstris(val, len, H20_STRLIT("br")) != 0 {

```

```
29         return H20_COMPRESS_HINT_ENABLE_BR;
30     }
31
32     return H20_COMPRESS_HINT_AUTO;
33 }
```

Código *extr_libcoreproxy.c_compress_hint_to_enum.rob*

E.2 Resultado da Inferência

Saída gerada no terminal:

```
1
2 <|im\_end|>
3
4 <|im\_start|>assistant
5
6 Run the following passes function(agressive-instcombine),early-cse<memssa>,
   function(sroa,simplify,gvn) to reduce the object file size to 259
7
8 [code]
9
10 /
```