Reescrita de Padrões na Representação Intermediária

Pedro Henrique Medeiros Hermes[1], **Wesley Attrot**[1]

[1]Departamento de Computação – Universidade Estadual de Londrina (UEL)
Caixa Postal 10.011 – CEP 86057-970 – Londrina – PR – Brasil

{pedro.henrique5, wesley}@uel.br

Abstract. *To build software capable of being executed by a specific machine, it is necessary to have a translation process performed by a compiler. Through intermediate representations, a compiler can apply machine-independent or machine-dependent optimizations, as well as techniques for pattern recognition and substitution. The study introduces the basic concepts of compilation and presents two approaches for replacing segments of the intermediate representation with architecture instructions.*

Resumo. *Para se construir um software que seja capaz de ser executado por uma máquina específica é necessário que haja algum processo de tradução desempenhado por um compilador. Através das representações intermediárias um compilador é capaz de aplicar otimizações independentes, ou dependentes, de máquina, além de aplicar técnicas de reconhecimento e substituição de padrões. O estudo apresenta os conceitos básicos da compilação e introduz duas abordagens de substituição de trechos da representação intermediária por instruções de arquitetura.*

1. Introdução

Os softwares de computador são ferramentas essênciais ao funcionamento do mundo moderno. Os códigos estão sempre atuando desde grandes sistemas bancários até aplicações simples, como a programação de uma máquina de lavar [4, 8]. No entanto, as máquinas não compreendem o mundo como os humanos, por isso é necessário que um elemento intermediário converta as informações para um formato compreensível por elas. Esse intermediário recebe o nome de compilador.

Um compilador é um programa de computador capaz de traduzir códigos de uma linguagem para outra. Sua principal aplicação é garantir que uma determinada máquina-alvo seja capaz de interpretar instruções descritas com alto nível de abstração [4]. Para atingir seu objetivo, o compilador deve conhecer a estrutura sintática e especificações da linguagem de origem, bem como a natureza da arquitetura para a qual se deseja traduzir o código. O processo de compilação é comumente dividido em duas etapas: *front-end* e *back-end*.

O *front-end*, também descrito como fase de análise, é o responsável por receber o código-fonte como entrada e garantir que o mesmo está em conformidade com as especificações da linguagem em que foi escrito [4, 8]. Para isso, são utilizadas técnicas de análise léxica, sintática e semântica, derivadas da teoria das linguagens formais. Como resultado, o *front-end* produz uma representação intermediária equivalente ao código original.

Já o *back-end*, ou fase de síntese, desempenha o papel mais custoso e crucial da compilação: a geração de código. Os diferentes processos do *back-end* são o seletor de código, o alocador de registradores e o escalonador de instruções, que, através da representação intermediária produzida na fase de análise, executam algoritmos complexos de otimizações específicas de arquitetura para, enfim, entregar um código baixo nível capaz de ser compreendido por uma máquina.

Como etapa opicional, mas amplamente utilizada, existe o *middle-end*, ou otimizador. O *middle-end* é responsável por interpretar uma determinada representação intermediária e identificar trechos de código que podem ser substituídos por outras expressões que produzem o mesmo resultado, porém de forma mais eficiente. Diferente das otimizações do *back-end*, essa etapa não possui conhecimento nenhum sobre a arquitetura para a qual o código será traduzido, por isso realiza operações que são independentes de máquina.

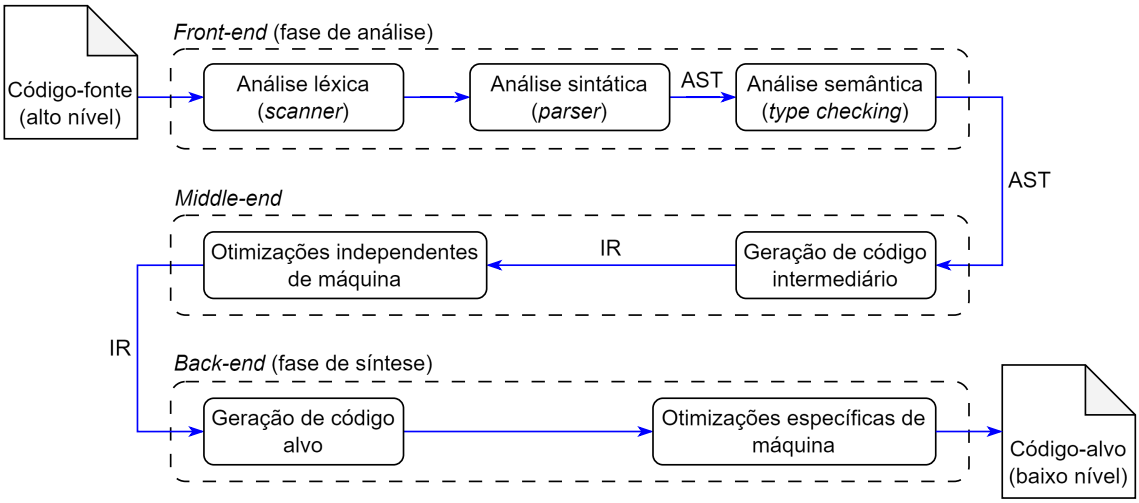

Figura 1. Etapas da compilação. Adaptado de [4, 8]

Uma representação intermediária, ou IR - *Intermediate Representation*, é a estrutura de dados principal dos processos de síntese do compilador [4] e, por isso, devem ser cuidadosamente implementadas de forma a não prejudicar o desempenho da compilação. O conhecimento derivado sobre o código-fonte não pode ser representado literalmente, mas sim de forma simbólica. Uma AST - *Abstract Syntax Tree* é a IR gerada pelo *front-end* e representa as operações do código-fonte apenas pela sua sintaxe. A Figura 3 mostra um exemplo de AST.

Sem as IRs seria completamente inviável para o compilador executar qualquer operação de otimização, visto que seria muito custoso, do ponto de vista computacional, iterar sobre um código-fonte extenso diversas vezes. Por isso, foram criados diversos tipos de IRs com propósitos diferentes e aplicações em várias etapas da fase de síntese, como árvores sintáticas abstratas, grafos acíclicos direcionados, grafo de fluxo de controle, etc [4].

Por fim, com o uso da IR é possível aplicar técnicas de reconhecimeto e substituição de padrões por códigos mais eficientes, que estão disponíveis através do conjunto de instruções da arquitetura. O presente trabalho visa explorar o algoritmo *Source*

Matching and Rewriting [9] e a linguagem *Twig* [1], que buscam resolver o problema supracitado.

2. Fundamentação Teórico-Metodológica e Estado da Arte

Com as IRs é possível reduzir o trabalho necessário para se desenvolver um tradutor para uma máquina específica, visto que, se já existir um *front-end* para a linguagem fonte, é preciso apenas construir o *back-end*, partindo da IR previamente gerada. Logo, existem $m \times n$ compiladores, onde m são *front-ends* e n são *back-ends*.

2.1. Representações Intermediárias

Árvore sintática Uma árvore sintática é uma representação intermediária de alto nível e tipo gráfico que contém a derivação gramatical completa do código-fonte. Os nós pai representam os símbolos não-terminais (variáveis), enquanto as folhas representam os símbolos terminais (os *tokens* identificados pelo *scanner*).

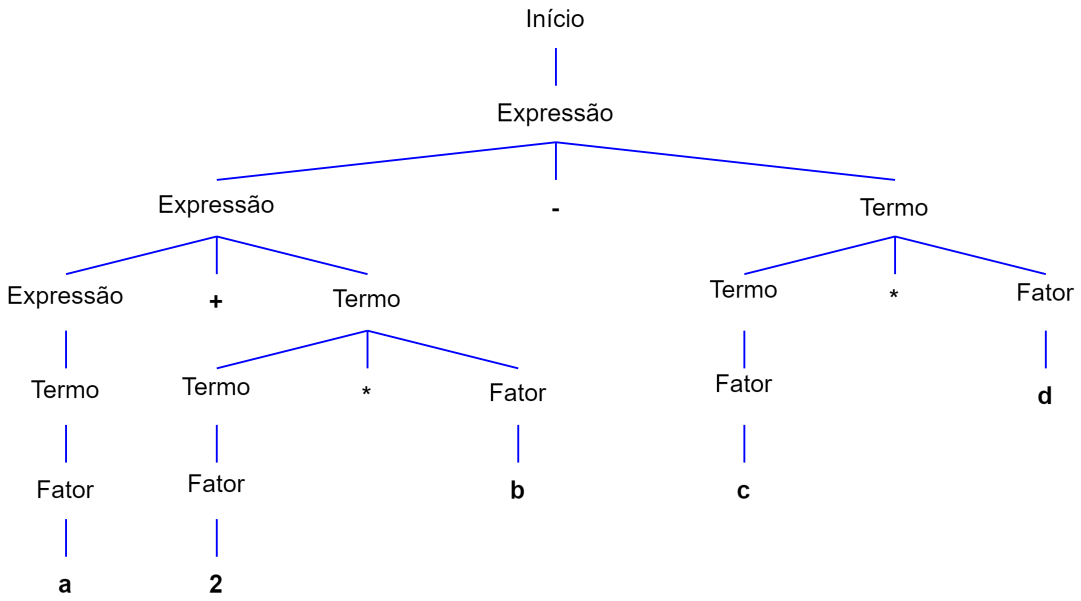

Figura 2. Exemplo de árvore sintática. Adaptado de [4]

O uso das árvores sintáticas está concentrado principalmente no *front-end*. Em relação aos processos do *back-end*, uma IR desse tipo não é ideal pois retém muita informação que não é relevante ao gerador de código ou otimizador, por exemplo. A exclusão de informação excessiva, ou compressão da árvore sintática, pode ser realizada através do emprego de abstrações.

Uma AST - *Abstract Syntax Tree* é uma representação comprimida da árvore sintática que oculta as variáveis gramaticais, reduzindo, assim, a densidade da estrutura e recursos de hardware necessários para armazená-la. A Figura 3 representa a conversão da árvore sintática da Figura 2 em uma AST.

Grafo acíclico direcionado Grafos são estruturas mais flexíveis e ainda capazes de representar a mesma informação de uma árvore. Explorando tal característica, é possível

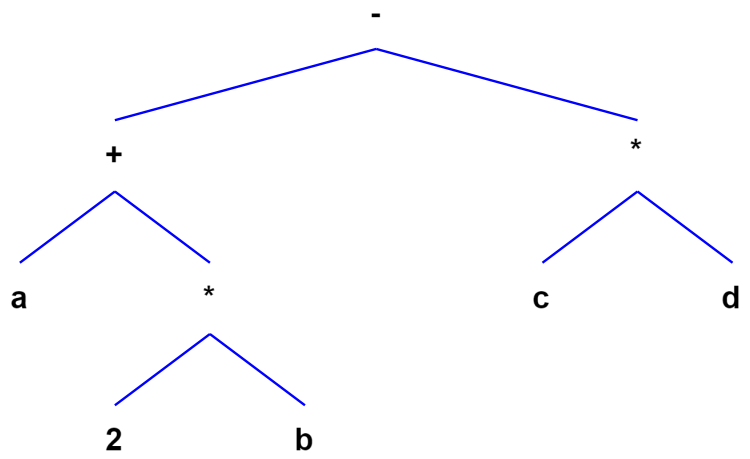

Figura 3. Exemplo de árvore sintática abstrata. Adaptado de [4]

converter uma AST em um DAG - *Directed Acyclic Graph* e comprimir ainda mais a representação sintática através do compartilhamento de nós pai.

Ao representar a expressão $a \times 2 + a \times 2 \times b$ através de uma AST, a operação $a \times 2$ derivaria duas sub-árvores, enquanto no DAG a mesma expressão seria representada uma única vez. A Figura 4 contém o exemplo de um DAG.

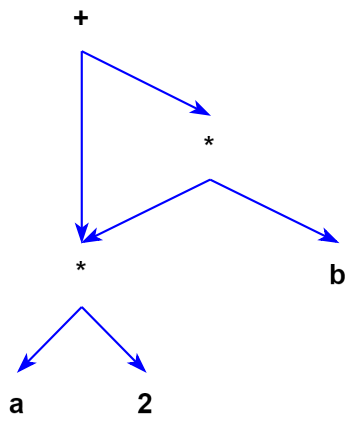

Figura 4. Exemplo de IR como grafo acíclico direcionado. Fonte: [4]

Grafo de fluxo de controle Os grafos de fluxo de controle, ou CFG - *Control-Flow Graph*, são responsáveis por representar o fluxo dos diferentes blocos de comando do código-fonte, onde os blocos de comando são descritos como nós e o controle é descrito pelas arestas. Se uma estrutura de repetição é representada em CFG, o grafo exibe um ciclo que deixa explícito o fluxo da aplicação; daí seu benefício comparado à uma árvore, que, por conta de sua rigidez, é incapaz de representar ciclos facilmente.

É comum que um CFG seja implementado através da mesclagem de diferentes tipos de IRs. Enquanto o grafo representa apenas o fluxo da aplicação, uma AST, por

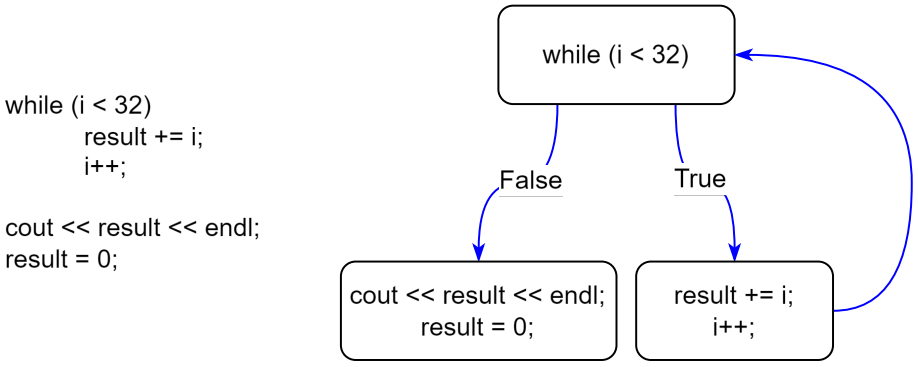

Figura 5. Representação do código-fonte como grafo de fluxo de controle. Adaptado de [4]

exemplo, descreve as operações dentro dos blocos de comando de forma sintática. Se necessário, o projetista pode optar por adotar um DAG como representação para os blocos de comando. Tal abordagem resulta em uma IR híbrida.

Aproveitando as facilidades providas pelas IRs e o conhecimento do conjunto de instruções da arquitetura para a qual o código será traduzido, é possível introduzir alguma outra etapa de otimização que vise tirar proveito de tais instruções pré-definidas. O emprego dessa técnica trás benefícios na performance do programa resultante.

2.2. Instruções Específicas de Arquitetura

O conjunto de instruções, ou ISA - *Instruction Set Architecture*, descreve uma série de operações definidas pelo fabricante do processador, microprocessador ou microcontrolador, sendo instruções específicas de arquitetura, como x86, ARM, MIPS, etc [2]. Os comandos estão disponíveis ao programador através de linguagem baixo nível, com pouca abstração, chamada *assembly*.

MMX A tecnologia MMX foi desenvolvida pela Intel e integrada pela primeira vez aos processadores da família Pentium. O objetivo era maximizar o desempenho dos chips no processamento de aplicações multimídia, que anteriormente requeriam máquinas distintas designadas para essas atividades [5]. Com o conjunto de instruções do MMX, foram introduzidas 57 novas operações pré-definidas, oito registradores e quatro tipos de dados que estão presentes em processadores de diferentes fabricantes, como a AMD, por exemplo.

SSE - *Streaming SIMD Extensions* O SSE é uma tecnologia presente em diversas arquiteturas de diferentes fabricantes que visa aumentar o desempenho, principalmente, de aplicações 3D [7]. Diante da característica que os processadores mais antigos tinham de processarem apenas um elemento de dados por ciclo e a demanda por processamento gráfico, a adição do SSE ao conjunto de instruções solucionou esse problema, pois passou a permitir que os chips lidassem com múltiplos elementos de dados.

Posteriormente foi criado o SSE2 que visava melhorar os benefícios que o SSE anterior já entregava. Essa atualização adicionou mais 144 instruções ao conjunto do SSE

e dobrou a taxa de *bits* suportada pelas instruções de tipo inteiro, introduzidas pelo MMX, de 64 para 128 *bits*. As principais aplicações beneficiadas foram MPEG-2, MP3 e gráficos 3D.

Outras atualizações incluem SSE3, SSE4, SSE4.1 e SSE4.2

AES - *Advanced Encryption Standard* Com o objetivo de acelerar a computação dos algoritmos complexos do padrão AES, em 2010 a Intel adicionou aos seus processadores um novo conjunto de instruções denominado AES-NI que é capaz de executar as etapas mais custosas de encriptação e decriptação via hardware, aumentando a performance do sistema em até 10 vezes [6].

As instruções `aesenc`, `aesenclast`, `aesdec` e `aesdeclast` são destinadas à aceleração de encriptação e decriptação, enquanto as instruções `aeskeygenassist` e `aesimc` são destinadas à geração de chaves.

Como todo algoritmo de criptografia o AES possui seus pontos de vulnerabilidade. Um dos possíveis ataques, conhecido como *side channel attack*, usa as tabelas de consulta (*lookup tables*) para recuperar informações sobre a chave gerada pelo AES ao introduzir um código espião que realiza inúmeras operações de leitura na tabela [3]. O emprego do AES-NI minimiza a chance de ataque ao tornar desnecessário o uso da *lookup table*, pois todo o processo é realizado via hardware, sem interferência de nenhum software.

Há outros inúmeros conjuntos de instruções não mencionados aqui.

3. Objetivos

O intuito da pesquisa é explorar formas de se reconhecer determinados padrões nas representações intermediárias e substituí-los por outros códigos mais eficientes, providos pelo conjunto de instruções da arquitetura alvo.

4. Procedimentos metodológicos/Métodos e técnicas

Serão analisados o algoritmo *Source Matching and Rewriting* [9] e a linguagem *Twig* [1] para compreensão do problema e como foram desenvolvidas as soluções, a fim de produzir uma nova abordagem.

5. Cronograma de Execução

As atividades planejadas para serem executadas durante a realização dessa pesquisa são as seguintes:

1. Pesquisa bibliográfica;
2. Aprimoramento dos conhecimentos de algoritmos específicos;
3. Implementação de algoritmo para reconhecimento de padrões;
4. Testes de *benchmark*;
5. Análise de resultados;
6. Escrita do TCC;

A Tabela 1 apresenta os detalhes do cronograma.

	Mar.	Abr.	Mai.	Jun.	Jul.	Ago.	Set.
Pesquisa bibliográfica	X	X					
Aprimoramento dos conhecimento de algoritmos específicos		X	X	X			
Implementação de algoritmo para reconhecimento de padrões			X	X	X		
Testes de *benchmark*						X	X
Análise de resultados							X
Escrita do TCC			X	X	X	X	X

Tabela 1. Cronograma de execução

6. Resultados esperados

É esperado que o algoritmo desenvolvido durante o TCC apresente resultados melhores, ou pelo menos comparáveis, aos algoritmos usados como referência.

7. Espaço para assinaturas

Londrina, *4 de Março de 2024*.

_____ _____

Pedro Henrique Medeiros Hermes Wesley Attrot

Referências

[1] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Trans. Program. Lang. Syst.*, 1989.

[2] Instruction Set Architecture (ISA). Disponível em: `https://www.arm.com/glossary/isa`. Acessado em: 25 de Fevereiro de 2024.

[3] Michael Neve e Kris Tiri. On the complexity of side-channel attacks on AES-256 – methodology and quantitative results on cache attacks. Cryptology ePrint Archive, Paper 2007/318, 2007.

[4] Keith D. Cooper e Linda Torczon. *Construindo Compiladores*. Elsevier, Campus, 2nd edition, 2014.

[5] Randall Hyde. *The Art of Assembly Language Programming*. No Starch Press, 2nd edition, 2010.

[6] Intel Advanced Encryption Standard Instructions (AES-NI). Disponível em: `https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-encryption-standard-instructions-aes-ni.html`. Acessado em: 27 de Fevereiro de 2024.

[7] Intel Instruction Set Extensions Technology. Disponível em: `https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html`. Acessado em: 25 de Fevereiro de 2024.

[8] Ivan Ricarte. *Introdução à Compilação*. Elsevier, Campus, 1st edition, 2008.

[9] Hervé Yviquel e Guido Araujo Vinicius Espindola, Luciano Zago. Source Matching and Rewriting for MLIR Using String-Based Automata. *ACM Trans. Archit. Code Optim.*, 2023.