UNIVERSIDADE FEDERAL DA FRONTEIRA SUL
CAMPUS CHAPECÓ
CURSO DE CIÊNCIA DA COMPUTAÇÃO

COMPARAÇÃO ENTRE CFGS EM DIFERENTES ETAPAS DO PROCESSO DE COMPILAÇÃO

LUCAS ARSEGO DE MELLO

CHAPECÓ
2018

LUCAS ARSEGO DE MELLO

COMPARAÇÃO ENTRE CFGS EM DIFERENTES ETAPAS DO PROCESSO DE COMPILAÇÃO

Trabalho de conclusão de curso de graduação apresentado como requisito para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.
Orientador: Dr. Emílio Wuerges

CHAPECÓ
2018

Bibliotecas da Universidade Federal da Fronteira Sul - UFFS

Mello, Lucas Arsego de
Comparação entre CFGs em diferentes etapas do processo de compilação / Lucas Arsego de Mello. -- 2018.
37 f.:il.

Orientador: Emílio Wuerges.
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal da Fronteira Sul, Curso de Ciência da Computação, Chapecó, SC , 2018.

1. Passos de Otimização. 2. Compiladores. 3. Similaridade de CFGs. I. Wuerges, Emílio, orient. II. Universidade Federal da Fronteira Sul. III. Título.

Elaborada pelo sistema de Geração Automática de Ficha de Identificação da Obra pela UFFS com os dados fornecidos pelo(a) autor(a).

LUCAS ARSEGO DE MELLO

COMPARAÇÃO ENTRE CFGS EM DIFERENTES ETAPAS DO PROCESSO DE COMPILAÇÃO

Trabalho de conclusão de curso de graduação apresentado como requisito para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.

Orientador: Prof. Emílio Wuerges

Aprovado em: 07\12\2018

BANCA EXAMINADORA:

Emílio Wuerges - UFFS

Dr. Denio Duarte - UFFS

Dr. Bráulio Adriano de Mello - UFFS

RESUMO

Compiladores são responsáveis por traduzir um código em uma linguagem específica para um código equivalente em outra linguagem alvo, neste processo ele realiza uma série de otimizações. O objetivo deste trabalho é identificar os passos do processo de compilação independente de máquina que causam as principais mudanças no fluxo de controle do sistema. Para analisarmos o fluxo de controle do sistema, utilizamos um modo de representação em forma de grafos, chamado de Grafo de Fluxo de Controle (CFG). A fim de atingirmos esse objetivo, foi necessária a realização de comparações entre as CFG's de todos estes passos realizados pelo compilador, essa comparação foi feita segundo a quantidade de nós e arestas da CFG. Como resultado, foi identificado o provável passo de otimização que causou as principais mudanças no fluxo de controle do sistema, diminuindo significativamente o numero de nós e arestas da CFG's em sua execução. Além disso, detalhamos esse passo e mostramos as funcionalidades que causaram essas mudanças.

Palavras-chave: Control Flow Graph; Similarity CFGs; Compilation process; Optimization methods..

ABSTRACT

Compilers are responsible for representing a code in a language task for a code that can be used as a process language. The objective of this work is to identify the steps of the machine independent compilation process as main changes in the control flow of the system. For the analysis of the control flow of the system, a representation model in the form of graphs, called Control Flow Chart (CFG), is used. In order to achieve the objective, a comparison of comparisons between the CFGs of all these works performed by the compiler was necessary, the number was compared with the number of nodes and the predictions of the CFG. As a result, it was identified the optimization process that caused important changes in the system flow, reducing the number of nodes and the guidelines of the CFG in its execution. In addition, we detail the step and show the features that caused those changes.

Keywords: Topic Modeling; Topics; Coherence Measures, Metrics; Sensitivity.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

PGO	*Profile-Guided Optimization*
CFG	*Control Flow Graph*
WCET	*Worst-Case Execution Time*
ARM	*Advanced RISC Machine*
ISA	*Instruction Set Architecture*
IoT	*Internet of Things*
LTO	*Link-Time Optimization*
SSA	*Static Single Assignment*
GVN	*Global Value Numbering*

SUMÁRIO

1 INTRODUÇÃO

Compiladores são programas que lêm um código escrito em uma linguagem *a* e traduz essa entrada em um código equivalente em outra linguagem alvo. Como apresentado na Figura 1.1, um importante passo desse processo de tradução é o relato ao usuário da presença de erros no arquivo fonte (SETHI; ULLMAN; S. LAM, 2008).

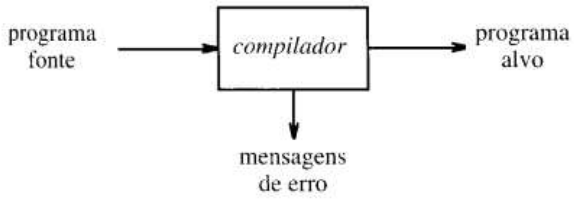

Figura 1.1: Compilador. Fonte:(SETHI; ULLMAN; S. LAM, 2008)

Durante o processo de compilação e nas etapas de otimizações, é necessário encontrar pontos críticos, para isso, os compiladores usam de técnicas para extrair informações sobre o código fonte afim de otimizar melhor o código. Uma técnica de otimização muito utilizada por compiladores é a PGO (*Profile-Guided Optimization*), essa técnica se utiliza de perfis criados por informações coletadas de uma compilação prévia do programa, sendo retiradas através de instruções colocadas em lugares específicos do código fonte original. Após a criação dos perfis é feito uma nova compilação utilizando o PGO e os perfis para realizar melhores otimizações (WICHT et al., 2014).

No entanto, um dos problemas é que este modo de otimização deixa o seu uso limitado a algumas aplicações, impossibilitando o uso dele em sistemas que não disponibilizam de muito tempo para sua compilação.

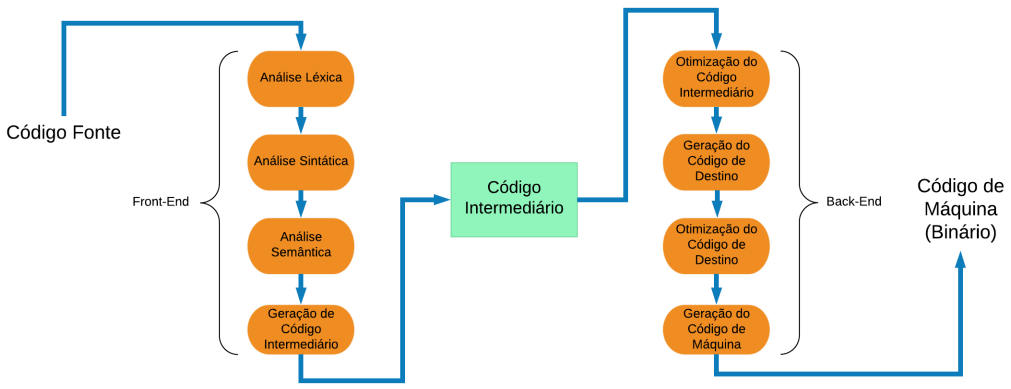

Figura 1.2: Etapas de Compilação

Em vista disso, buscamos investigar em quais etapas de compilação existem possibilidades de realizar melhores otimizações, utilizando para a busca, a similaridade entre CFGs (*Control Flow Graphs*). Na Figura 1.2 são mostradas todas as etapas do processo de compilação, sendo retiradas as CFGs dos seguintes momentos:

- Código Intermediário;
- Entre as etapas do Back-End;

Nesse sentido, a pesquisa se mostra relevante, ao verificar a importância de encontrar os pontos críticos do processo de compilação e contribuir para uma melhor solução do sistema de otimização dos compiladores, oportunizando aos casos que não possuem tempo disponível para esperar, uma compilação utilizando *PGO* em seu método de otimização.

Tradicionalmente as otimizações feitas pelos compiladores eram feitas apenas de arquivo em arquivo, foi nesse momento que surgiu um método de otimização que primeiro faz a união dos códigos intermediários de cada arquivo, para posteriormente otimizar o resultado dessa união utilizando informações globais geradas na compilação de cada arquivo. Este método foi chamado de *LTO (Link-Time Optimization)*, vale salientar que a sua criação trouxe um impacto grande na evolução das técnicas de otimizações.

Nas versões mais atuais dos compiladores existem *flags* que controlam a utilização de técnicas de otimização, sendo que alguns deles disponibilizam a aplicação de ambas as técnicas *PGO* e *LTO* ou podem escolher somente uma delas. Nos casos em que os compiladores utilizam as duas, primeiro é compilado o código empregando as técnicas do *LTO* somada aos fatores de utilização do *PGO* como a instrumentação do código. Feito isso, a segunda etapa fará a simulação deste código gerado e em seguida compilará novamente utilizando as técnicas do *PGO*.

Devido as desvantagens encontradas na técnica *PGO* como a dupla compilação e a instrumentação do código, bem como a utilização desta otimização mais agressiva feita pelo *LTO* de unir os códigos intermediários e otimizar o código resultante (podendo gerar mudanças mais drásticas no código final em relação a uma otimização tradicional), acreditamos que encontrar estes pontos críticos dentro do processo de compilação, possa ajudar a tornar mais precisa as técnicas já existentes, sem a necessidade da dupla compilação ou a instrumentação do código.

A similaridade de CFGs pode ser utilizada para encontrar pontos críticos no processo de compilação, como também, na utilização de outras aplicações, como: na busca por detecção de

malwares(onde é necessário fazer a comparação de trechos de código em relação a uma base de dados) (KINABLE; KOSTAKIS, 2011); na avaliação automatizada de tarefas ou *softwares* (VUJOŠEVIĆ-JANIČIĆ et al., 2013); e na engenharia de software para obter informações sobre alterações entre diferentes versões de um sistema (APIWATTANAPONG; ORSO; HARROLD, 2004).

A utilização de um *Control Flow Graphs* para as comparações faz sentido pois ela consegue mostrar toda a estrutura do código fonte, bem como as ligações entre os blocos básicos gerados.

Assim, o objetivo geral deste estudo é encontrar em qual momento do processo de compilação existem as maiores mudanças no código do programa. Para operacionalizar tal objetivo, elencamos os seguintes objetivos específicos:

- Identificar quais passos de otimizações feitos pelo compilador mudam mais o CFG.
- Explicar o porque esses passos mudam o CFG.

Desse modo, o presente estudo foi organizado da seguinte forma: Num primeiro momento, buscou-se apresentar o embasamento teórico, afim de mostrar os elementos que foram tomados como base para a realização do projeto. Posteriormente, foi detalhado os passos desenvolvidos para identificar os passos que mais mudam o CFG e por fim, foi apresentado o detalhamento dos passes críticos encontrados.

2 REFERENCIAL TEÓRICO

Neste capítulo será apresentado todas as técnicas e argumentos teóricos utilizados para a execução da pesquisa. Este trabalho abrange diversas áreas da computação, como grafos e compiladores. Cada seção deste capítulo, foi dividio em assuntos específicos que serão utilizados no desenvolvimento da pesquisa.

2.1 Control Flow Graph

Um *Control Flow Graph*(CFG) tem como principal objetivo representar o grafo de fluxo de controle de um código fonte. O ponto a destacar, é que um CFG bem construído, traz consigo todas as dependências do código de entrada, isto é importante pois em situações de comparação entre CFGs ou de uso da CFG para a retirada de informações do código não manter a estrutura do código fonte, trará resultados errados (SETHI; ULLMAN; S. LAM, 2008).

Definição 1 *(Control Flow Graph). O Control Flow Graph é um grafo direcionado* $CFG(V_{CFG}; E_{CFG})$ *como representação do fluxo de controle de um arquivo fonte. Onde* V_{CFG} *é o conjunto de nós(blocos básicos), contendo uma única entrada e uma única saída.* E_{CFG} *representa o conjunto de arestas(ligações) sendo que um elemento deste conjunto representa a ligação entre um* V_n *a outro. O* CFG *contem um* V_i *único como entrada e um* V_j *único com saída. Sendo que cada elemento de* V_{CFG} *pode ter mais de uma aresta de saída e entrada (SETHI; ULLMAN; S. LAM, 2008).*

As Figuras 2.3 e 2.2, apresentamos dois exemplos de CFGs geradas pelo código da Figura 2.1, sendo que cada uma das CFGs tem um único bloco básico para a entrada($BLOCKi\ Entry$) do código e um único bloco básico para a saída($BLOCKi\ Exit$) do código.

Segundo SETHI; ULLMAN; S. LAM (2008) um bloco básico é definido por um conjunto de instruções sem pontos de entrada, exceto a primeira instrução e sem desvios, exceto, possivelmente, a última instrução do bloco.

2.2 Profile-Guided Optimization

A otimização baseada em perfis*(Profile-Guided Optimization(PGO))* trazem resultados excelentes na compilação de um programa. As informações de caminhos de execuções que são

geradas através das suas técnicas, auxiliam os compiladores a gerarem um código melhor e com uma melhor distribuição na memória cache (WICHT et al., 2014).

As implementações mais utilizadas nos compiladores, vem de um sistema de dupla compilação, que utiliza informações retiradas de uma execução do código pra criar *profiles*(perfil), sendo utilizados em uma segunda compilação como informação para as otimizações. Geralmente a informação coletada é a frequência de execuções das borda entre os blocos básicos. Nas versões disponibilizadas pelos compiladores, esses *profiles* são criados de forma dinâmica, fazendo com que ao longo de sua execução de criação, possam ocorrer alterações, podendo resultar em uma aceleração relevante em aplicativos que usam uma intensiva entrada e saída de dados.

Como foi apontado por WICHT et al. (2014), essas implementações utilizam de um modelo baseado em instrumentação do código, onde são inseridas instruções em locais específicos. Durante a execução, são criados contadores que utilizam das instruções para se incrementarem e gerarem os perfis. Por fim o código é compilado novamente utilizando o perfil gerado para realizar as otimizações.

Desvantagens desta metodologia são apontadas por WICHT et al. (2014), sendo elas:

- Instrumentação de código podem diminuir a sua velocidade, BALL; LARUS (1994) demonstra que um binário instrumentado pode ser muito mais lento, podendo ter entre 9% e 105% de sobrecarga;
- A criação de *profiles* através de uma execução inicial do código, gera um modelo de dupla compilação, ficando inconveniente. De fato, para aplicativos com tempo longo de compilação, ter este tempo dobrado pode diminuir a sua produtividade;
- Um pequeno conjunto de informações podem ser coletadas. Deixando informações que poderiam ser úteis fora deste conjunto, como por exemplo, informações sobre problemas de memória ou previsão de ramificações;
- Por existir um forte acoplamento entre as duas construções, fica necessário a utilização das mesmas opções de otimização na primeira e segunda compilação. Sem isso, o CFG de ambas as compilações podem não corresponder e os dados de criação de perfis podem não ser usados com a sua precisão máxima;
- A medida que novas instruções são inseridas no programa, podem alterar a qualidade do perfil gerado, podendo alterar as decisões de otimização.

Analisando estas desvantagens, percebemos que a utilização desta técnica, fica limitada a sistemas pequenos ou que não tenham problemas com o tempo de construção do seu *software*. Essas informações mostram a necessidade de encontrarmos novos meios de otimizações para sistemas que não consigam utilizar esta técnica, ficando com a utilização de otimizações que geram um impacto pequeno ao seu produto final.

2.3 Similaridade de CFGs

A similaridade de *Control Flow Graph* é muito utilizada em diversas áreas como na busca por detecção de *malwares*, onde é necessário fazer a comparação de trechos de código em relação a uma base de dados KINABLE; KOSTAKIS (2011), pode ser na avaliação automatizada de tarefas ou *softwares* VUJOŠEVIĆ-JANIČIĆ et al. (2013), e também utilizado em engenharia de software para obter informações sobre alterações entre diferentes versões de um sistema (APIWATTANAPONG; ORSO; HARROLD, 2004).

Devido a todo o processo de compilação, propriedades que existiam no código fonte podem não aparecerem nos binários compilados, devido ás otimizações realizadas pelo compilador XU; SUN; SU (2009). Utilizar similaridade de CFGs com a intenção de encontrar qual momento o compilador faz essas mudanças que retiram informações do código binário, podem acarretar em mudanças no processo de otimização para que essas perdas não ocorram. Deixando um código binário otimizado e, ao mesmo tempo, disso mantendo as informações equivalente ao código fonte.

Existem algoritmos propostos para testar a similaridade entre CFGs, CHAN; COLLBERG (2014) apresentam, funções que retornam como resultado se as CFGs de entrada são similares ou não. Esses algoritmos podem se basear na analise da topologia das CFGs ou no conteúdo dos blocos básicos, ou em ambos, calculando uma pontuação de similaridade.

Além deste método podemos analisar a similaridade das CFGs através de formas mais simples, como a quantidade de blocos básicos, numero de ligações entre os blocos, e em casos mais complexos como a quantidade de ciclos na CFG.

Utilizar um método ou outro, vai do objetivo do seu trabalho, qualquer um dos métodos traz informações relevantes e encontra algum tipo de similaridade entre as CFGs.

2.3.1 Métricas

Para que as comparações entre as CFGs sejam realizadas neste trabalho, precisamos definir algumas métricas. Como apresentado anteriormente, alguns compiladores retiram informações das CFGs para a criação de *profiles*, quando executado a otimização baseada a perfis(PGO), uma delas é a frequência da execução de borda entre blocos básicos (WICHT et al., 2014).

Algumas métricas que poderão ser utilizadas na comparação entre CFGs são: a quantidade de blocos básicos de cada CFG e a quantidade de ligações entre blocos básicos (arestas). Inicialmente somente essas duas métricas serão utilizadas, tendo em vista que, para mostrar a sua configuração, a quantidade de nós e arestas do Grafo são de suma importância. Desta forma, percebemos que estas duas métricas inicias são suficientes para mostrar se existe ou não alguma mudança básica entre as CFGs.

2.4 CFGs do processo de compilação

O compilador tem o papel de transformar um código fonte de uma linguagem compilada em um programa semanticamente equivalente, porem em escrito em uma linguagem de código objeto. Este processo é dividido em duas principais etapas, a etapa de *front-end* que é responsável pelas análises léxica, sintática e semântica e a geração do código intermediário. A segunda etapa *back-end* é responsável pela otimização do código até a geração do código objeto(código binário).

Durante a primeira etapa não é possível fazer análises através de CFGs pois não existe uma estrutura sobre o código fonte, mas somente é feito a verificação se o código fonte está respeitando as regras da linguagem do programa. Já á partir da geração do código intermediário, podem começar a extração de CFGs durante as suas otimizações, porém a principal dificuldade é encontrar em quais momentos vão ser retiradas essa CFGs. Como cada tipo de compilador tem um tipo de estrutura de *back-end*, bem como de otimizações, não sendo possível mapear um caso genérico que atenda a todos os compiladores.

Cada compilador traz consigo uma serie de bibliotecas e funcionalidades, por padrão todo compilador tem uma *flag* que permite fazer a retirada da CFG durante o processo de compilação. Dentro dos diversos compiladores que existem, escolhemos dois para mostrarmos um exemplo de extração de CFG do código intermediário. Compiladores escolhidos foram o GCC

e Clang, para a extração de uma CFG de um código na linguagem C apresentado na Figura 2.1, o código apresentado faz a soma de dois números caso a uma condição desse como verdadeira. Os comandos de compilação executados foram:

- GCC : *gcc -fdump-tree-gimple test.c*
- Clang : *clang -cc1 -analyze -analyzer-checker=debug.DumpCFG test.c*

No comando do GCC, temos dois parâmetros, sendo o primeiro o *-fdump-tree-gimple* que identifica que deve-se gerar a CFG do código de entrada, o segundo parâmetro é o do aquivo de entrada. Já no comando do Clang, temos quatro parâmetros, sendo o primeiro o *-cc1* que indica que o front-end do compilador, e não o driver, deve ser executado, já o segundo parâmetro, *-analyze* inicia a análise estática, o terceiro parâmetro *-analyzer-checker=debug.DumpCFG* é o identificador para extrair a CFG do código de entrada e o ultimo parâmetro é o *teste.c* arquivo de entrada.

Com esses comandos é gerado como saída a CFG do código intermediário para o código de entrada. Este arquivo é formado por os blocos básicos gerados e as ligações que existem entre cada um deles. Figura 2.2 e 2.3 apresentam os exemplos de CFG geradas á partir do código de entrada da Figura 2.1 para os compiladores Clang e GCC, respectivamente.

test.c

```
int main(int argc, char *argv[]){
int a,b,c;
a = 3;
b = 5;
if(a>b){
  c = a+b;
}
return c;
}
```

Figura 2.1: Exemplo de código em linguagem C

Como pode-se visualizar nas Figuras 2.3 e 2.2 cada compilador monta a sua CFG com suas próprias configurações, deixando inviável a comparação entre CFGs de diferentes compiladores.

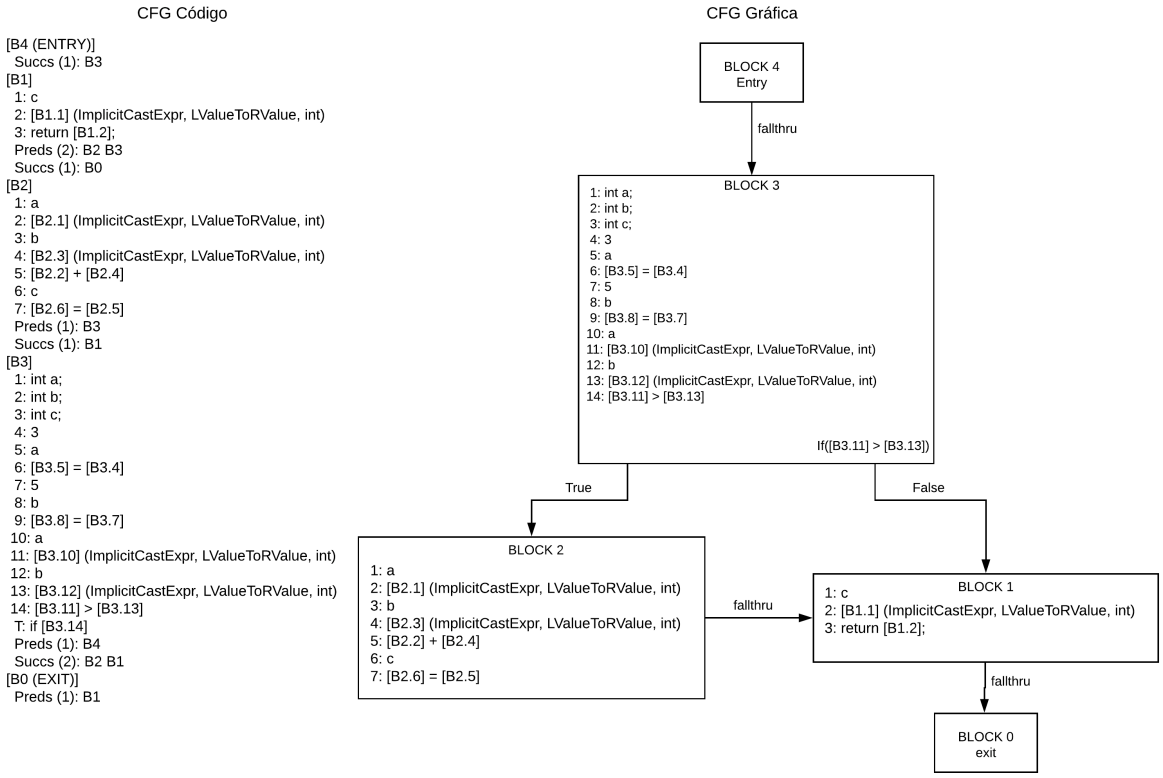

Figura 2.2: CFG gerada através do compilador Clang

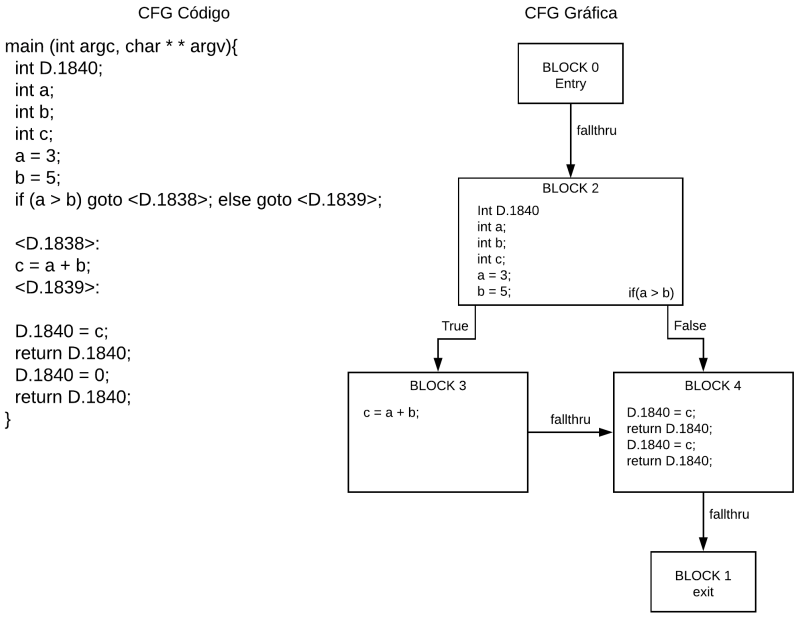

Figura 2.3: CFG gerada através do compilador GCC

2.5 CFG do código binário

Todo processo de compilação termina com a geração do código binário do programa, muitos programas são disponibilizados somente desta forma para o usuário, por este motivo existiram estudos de como realizar a reconstrução da CFG do programa a partir deste código.

Existem duas técnicas de construir a CFG á partir do binário, as técnicas estáticas e dinâmicas. As técnicas estáticas analisam a estrutura de código binários e o conjunto de instruções. Geralmente elas conseguem uma boa cobertura do código, mas em algum casos sofrem de baixa precisão pois é difícil resolver estaticamente os alvos indiretos das ramificações. Já as técnicas dinâmicas depende principalmente de execuções monitoradas do binário em um conjunto finito de execuções e podendo não garantir a cobertura total do código, sendo que elas também tem pouca escalabilidade na presença de *loops* de longa duração (XU; SUN; SU, 2009).

Em 2009 foi apresentada por XU; SUN; SU (2009) uma técnica nova e eficaz na geração de CFGs a partir do código binário. Essa técnica tenta resolver o problema com os alvos indiretos das ramificações através da exploração sistemática de ambas as direções das ramificações condicionais, fazendo isso através de execuções forçadas. Desta forma, a técnica explora de forma eficiente ambas as direções em cada ponto de ramificação, resolvendo os ramos indiretos dinamicamente para a construção de uma CFG precisa.

Outra técnica foi apresentada no mesmo ano por KINDER; ZULEGER; VEITH (2009), que é definida por uma interpretação abstrata para reconstruir a CFG, sendo parametrizada por um domínio abstrato. Sendo este domínio abstrato dado com uma representação de um CFG parcial. Para calcular os alvos de ramificações é utilizado uma formula que computa encontra a ramificação ideal.

Todas essas técnicas existem implementações que são possíveis de utilização, ficando a cargo do pesquisador fazer a escolhe de qual melhor atende o seu objetivo.

2.5.1 Gem5

Como mostrado anteriormente, dissertamos sobre algumas técnicas fáceis e precisas para realizar a extração do CFG á partir do código binário, porem para a sua construção é necessário os endereçamentos utilizados pelo código ao ser executado, para realizar a simulação de suas execuções precisaremos utilizar um simulador de sistema completo, pode-se usar o Gem5 BINKERT et al. (2011), devido ao seu suporte a diversas arquiteturas.

O Gem5 suporta a maioria das *Instruction Set Architecture*(Arquitetura do conjunto de instruções) comerciais(ARM, ALPHA, MIPS, Power, SPARC, e x86), deixando para o usuário escolher qual arquitetura convêm para seus estudos.

Uma arquitetura de conjunto de instruções(ISA), é a interface entre o *software* e o *hardware* do computador. Todo *software* escrito para uma ISA, poderá ser executado em um computador com outra arquitetura, porém depois da geração do código de máquina, este código só executará em um computador com a mesma ISA. A ISA define toda a arquitetura que do computador, em geral, ela define todos os tipos de dados suportados, como a memória principal e os registradores, a consistência de memória e modos de endereçamentos, o conjunto de instrução de máquina e o modelo de entrada/saída.

3 EXTRAÇÃO E ANÁLISE DAS CFGS

Este capítulo apresenta os passos utilizados para a constituição dos resultados finais desta pesquisa, os quais resultaram nas seguintes seções: extração da CFGs; análise geral das CFGs; análise dos passos que mudam o CFG e por fim, a explicação de cada um dos passos encontrados na etapa de análise.

3.1 Extração da CFGs

Na etapa de extração de CFGs foi preciso definir quais passos teríamos que rodar para simular uma compilação, para isso, utilizamos um comando do compilador que mostra os passos de otimizações realizados em uma compilação, como mostra o comando abaixo:

A Figura 3.1 mostra a saída dada pelo compilador na execução deste comando:

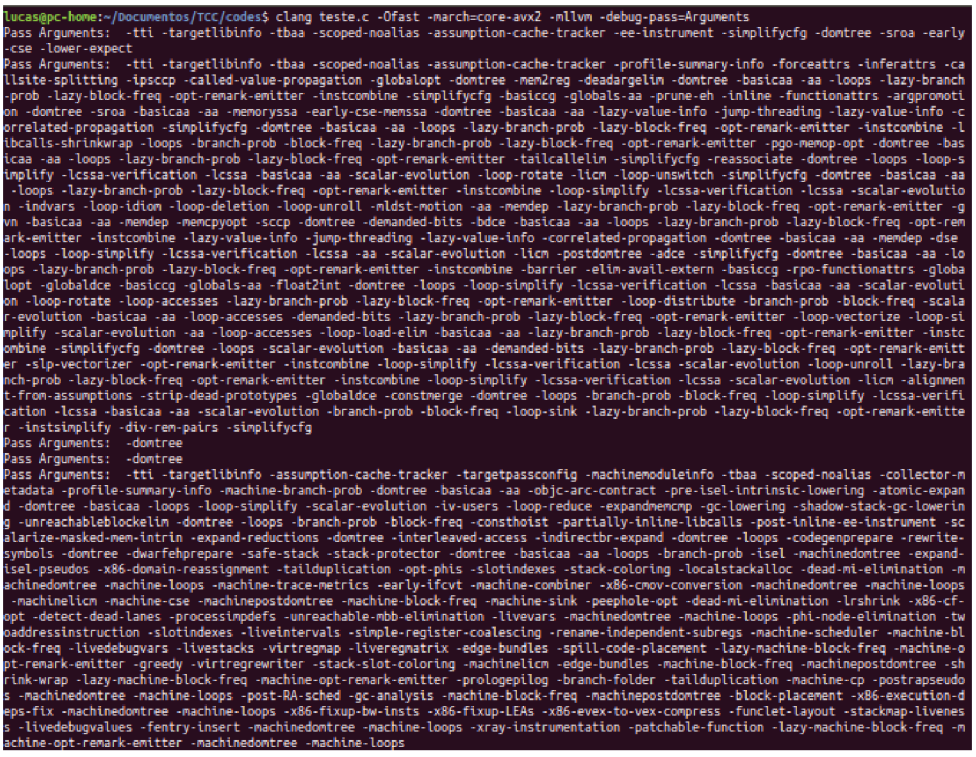

Figura 3.1: Passos do processo de compilação dados pelo compilado

Após analisar a saída, percebemos que existem três níveis de passos, sendo o último deles dependente da arquitetura de máquina, a qual não foi utilizada, pois o intuito é deixar os resultados mais amplos e sem qualquer dependência de arquitetura, podendo ser utilizados em trabalhos futuros em qualquer tipo de sistema. A lista selecionada tem 253 passos, tendo entre

eles passos de otimizações, análises e utilidades[1].

Depois dessa seleção foi necessário escolher qual seria o arquivo de entrada para a realização da pesquisa, nós utilizamos um código de um projeto da Universidade de Michigan chamado *MiBench2*[2], sendo ele um pacote de *benchmark* incorporado gratuito e comercialmente representativo. O arquivo selecionado foi uma implementação do algoritmo *Dijkstra("dijkstra.c")*, que tem a finalidade de encontrar o caminho mínimo de um vértice a outro em um grafo com pesos nas arestas, sendo que esse arquivo é formado por cinco funções(*"main", "dijkstra", "qcount", "dequeue"e "enqueue"*).

Após ter o arquivo selecionado utilizamos um comando para extrair o código intermediário desabilitando todos os passos:

```
clang -mllvm -disable-llvm-optzns dijkstra.c -emit-llvm -c -S
```

Tendo como arquivo de saída um código intermediário não otimizado (*"dijkstra.ll"*), sendo que para a realizar a sequência de passos escolhidas, precisamos retirar uma *flag* que impede a realização de passos nele(*"optnone"*), deixando o arquivo pronto para os próximos passos.

Como a lista de passos selecionada tem um tamanho relativamente grande(253 passos), e para não ser preciso extrair as CFGs manualmente, criamos um programa *"extractCFG.py"*[3] que faz de forma automatizada a execução dos comandos de extração, seguindo essa ordem para cada passo:

1. O primeiro comando é para fazer a utilização do passo no código intermediário, para isso utilizamos o seguinte comando:

   ```
   opt -march=x86 -mcpu=core-avx2 -men2reg arg1 -S arg2 -o arg3
   ```

   Tendo três argumentos necessários, o primeiro é o *arg1*, sendo o passo que será utilizado, o segundo é *arg2* que é o arquivo de entrada e o terceiro *arg3* é o arquivo de saída com o passo realizado.

2. O segundo comando é para fazer a extração da CFG do código intermediário é:

   ```
   opt -dot-cfg arg1
   ```

[1] https://llvm.org/docs/Passes.html
[2] https://github.com/impedimentToProgress/MiBench2
[3] https://github.com/LucasArsego/codesTCC

Que recebe somente um argumento como parâmetro *arg1*, sendo ele o arquivo do código intermediário ex:*"dijkstra.ll"*. A saída é um arquivo contendo a CFG para cada função do programa, com a extensão *".dot"*.

3.2 Análise geral das CFGs

Após extrair as CFGs, iniciamos a etapa de análise e, para isso, foram definidos duas métricas de comparação, sendo elas: a quantidade de blocos básicos(nós) e a quantidade de ligações entre os blocos básicos(arestas). A escolha destas métricas se deu pela simplicidade de suas implementações e ao testá-las percebemos que foram capazes de revelar quais passos causaram mudanças nas CFGs.

Para realizar as comparações foi preciso retirar as informações das CFGs utilizando as métricas escolhidas, para tanto, utilizamos uma biblioteca do Python[4] chamada Pydot[5], que possui funcionalidades para abrir e extrair as informações de um arquivo com extensão *".dot"*. Em relação a retirada da quantidade de nós e arestas de uma CFG, utilizamos as funções dessa biblioteca chamadas, *get_nodes()* e *get_edges()* respectivamente.

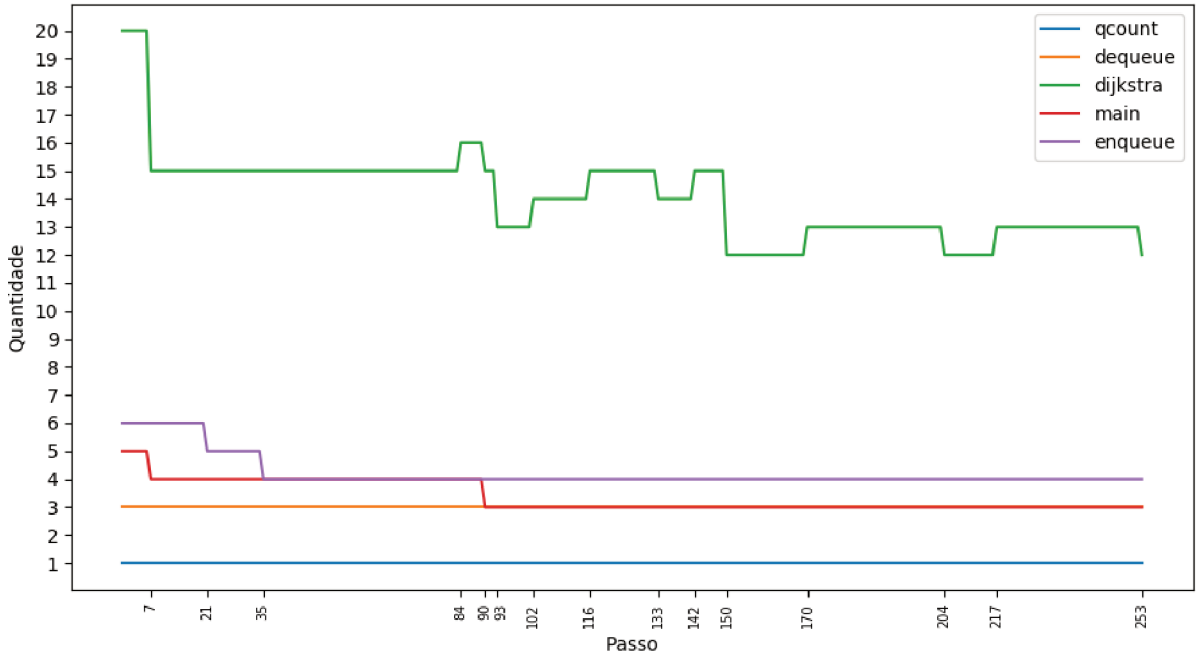

Figura 3.2: Gráfico da evolução da quantidade de Blocos Básicos

[4] https://www.python.org/
[5] https://pypi.org/project/pydot/

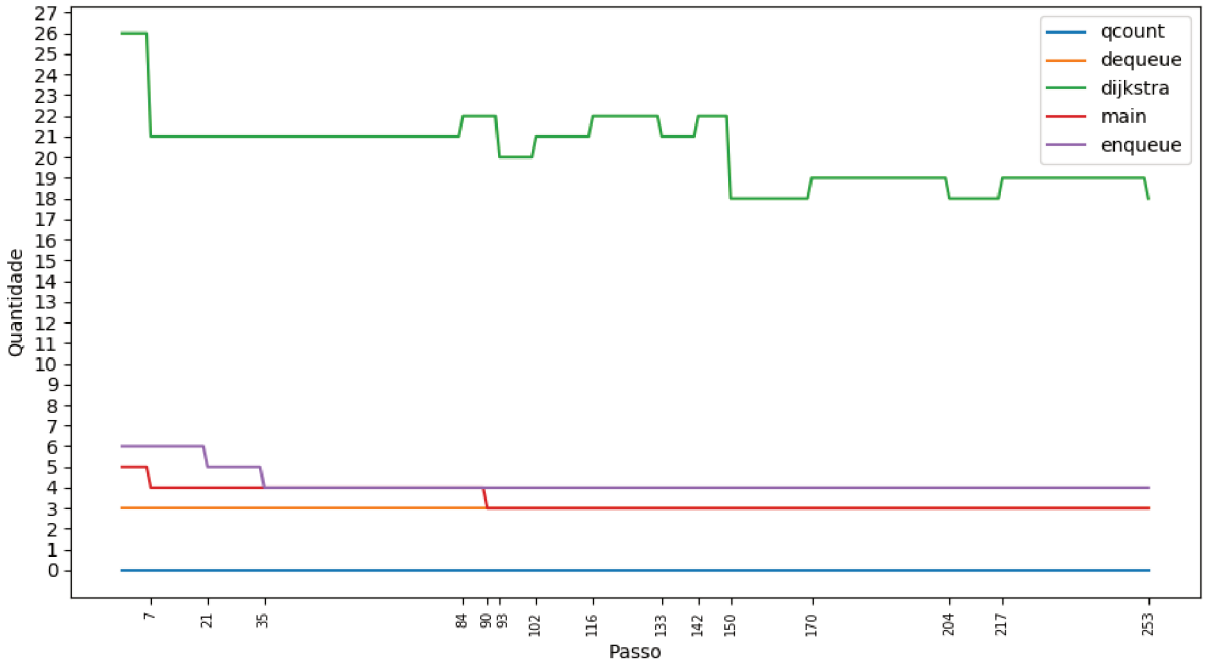

Figura 3.3: Gráfico da evolução da quantidade ligações entre os Blocos Básicos

A Figura 3.2 e 3.3 mostra o comportamento do número de Blocos Básicos e de Ligações entre eles durante todos os 253 passos que o compilador realiza.

Através desses dados desenvolvemos dois gráficos que mostram a evolução das mudanças ao longo dos 253 passos, separadas para cada uma das funções do programa. O gráfico da Figura 3.2 representa a evolução da quantidade de blocos básicos nas CFGs e o gráfico da Figura 3.3 representa a evolução na quantidade de ligações entre os blocos básicos nas CFGs, ambos mostram no seu eixo X o número da ordem de execução do passo que fez a mudança na CFG. Este número está relacionado na Tabela 3.1 junto com o nome do passo.

Ao analisar os gráficos das Figuras 3.2 e 3.3, foi evidenciado que algumas funções do programa não sofreram alterações, sendo elas as funções *"qcount"* e *"dequeue"*. Isso ocorre pela pouca complexidade de seu código, que traz pouco ou nenhum tipo de *loop*, desvio condicional ou incondicional para serem otimizados. Por não ocorrer mudanças, eliminamos elas do processo de análise.

Já as outras funções(*"dijkstra"*, *"main"*e *"enqueue"*) trouxeram alterações durante o processo de otimização, por isso retiramos os passos de cada uma delas, tanto na CFG da quantidade de Blocos Básicos, quanto na CFG da quantidade de Ligações entre os Blocos. Para apresentá-las foram geradas duas tabelas para cada função, sendo que ambas mostram o passo da alteração, a quantidade de Nós/Arestas anterior a execução do passo, a quantidade de Nós/Arestas depois da execução do passo, a diferença entre as duas quantidades, representada pelo Δ

Blocos Básicos		Ligações entre Blocos Básicos	
Passo	nº	Passo	nº
-simplifyCFG	7	-simplifyCFG	7
-ipsccp	21	-ipsccp	21
-simplifyCFG	35	-simplifyCFG	35
-loop-simplify	84	-loop-simplify	84
-loop-rotate	90		
-simplifyCFG	93	-simplifyCFG	93
-loop-simplify	102	-loop-simplify	102
-gvn	116	-gvn	116
-jump-threading	133	-jump-threading	133
-loop-simplify	142	-loop-simplify	142
-simplifyCFG	150	-simplifyCFG	150
-loop-simplify	170	-loop-simplify	170
-simplifiCFG	204	-simplifyCFG	204
-loop-simplify	217	-loop-simplify	217
-simplifyCFG	253	-simplifyCFG	253

Tabela 3.1: Passos por métrica que alteraram as CFGs

e o numero da ordem de execução do passo.

As Tabelas 3.2 e 3.3 representam as mudanças nas CFGs da função *Dijkstra*, a primeira mostra que a quantidade de Blocos Básicos ao longo do processo de otimização teve uma mudança significativa, visto que começou com 20 Blocos e terminou com 12. Os passos alterados de forma positiva, foram o *-simplifyCFG*, *-loop-rotate* e o *-jump-threading*, e os que alteraram de forma negativa foram os *-loop-simplify* e *-gvn*. Já a segunda, mostra a alteração na quantidade de Ligações entre os Blocos Básicos, começando com 26 e terminando com 18, os passos que alteraram de forma positiva foram os *-simplifyCFG* e o *-jump-threading*, e os que alteraram de forma negativa foram os passos *-loop-simplify* e *-gvn*.

A função *Enqueue* teve poucas alterações, como mostra as Tabelas 3.4 e 3.5, tanto no número de Blocos Básicos, e quanto no número de Ligações entre os Blocos, visto que em número obtivemos as mesmas mudanças, de 6 para 4.. Os passos que causaram isso, para ambas métricas, foram *-ipsccp* e *-simplifyCFG*, sendo os dois de forma positiva.

Passo	Qtd. Anterior	Qtd. Posterior	Δ	nº
-ipsccp	6	5	1	21
-simplifyCFG	5	4	1	35

Tabela 3.4: Passos que mudaram a quantidade de Blocos Básicos na CFG da função *Enqueue*

Passo	Qtd. Anterior	Qtd. Posterior	Δ	nº
-simplifyCFG	20	15	5	7
-loop-simplify	15	16	-1	84
-loop-rotate	16	15	1	90
-simplifyCFG	15	13	2	93
-loop-simplify	13	14	-1	102
-gvn	14	15	-1	116
-jump-threading	15	14	1	133
-loop-simplify	14	15	-1	142
-simplifyCFG	15	12	3	150
-loop-simplify	12	13	-1	170
-simplifyCFG	13	12	1	204
-loop-simplify	12	13	-1	217
-simplifyCFG	13	12	1	253

Tabela 3.2: Passos que mudaram a quantidade de Blocos Básicos na CFG da função *Dijiksta*

Passo	Qtd. Anterior	Qtd. Posterior	Δ	nº
-simplifyCFG	26	21	5	7
-loop-simplify	21	22	-1	84
-simplifyCFG	22	20	2	93
-loop-simplify	20	21	-1	102
-gvn	21	22	-1	116
-jump-threading	22	21	1	133
-loop-simplify	21	22	-1	142
-simplifyCFG	22	18	4	150
-loop-simplify	18	19	-1	170
-simplifyCFG	19	18	1	204
-loop-simplify	18	19	-1	217
-simplifyCFG	19	18	1	253

Tabela 3.3: Passos que mudaram a quantidade de ligações entre Blocos Básicos na CFG da função *Dijiksta*

Passo	Qtd. Anterior	Qtd. Posterior	Δ	nº
-ipsccp	6	5	1	21
-simplifyCFG	5	4	1	35

Tabela 3.5: Passos que mudaram a quantidade de ligações entre Blocos Básicos na CFG da função *Enqueue*

Já as Tabelas 3.6 e 3.7 representam os passos que causaram alterações nas CFGs da função *Main*, para ambas métricas as alterações foram causadas de forma positiva pelos mesmos passos, *-simplifyCFG* e *-loop-rotate*, tendo ocorrido uma diminuição de dois Blocos Básicos e de duas Ligações entre os Blocos, saindo ambas de 5 para 3.

Passo	Qtd. Anterior	Qtd. Posterior	Δ	nº
-simplifyCFG	5	4	1	7
-loop-rotate	4	3	1	90

Tabela 3.6: Passos que mudaram a quantidade de Blocos Básicos na CFG da função *Main*

Passo	Qtd. Anterior	Qtd. Posterior	Δ	nº
-simplifyCFG	5	4	1	7
-loop-rotate	4	3	1	90

Tabela 3.7: Passos que mudaram a quantidade de ligações entre Blocos Básicos na CFG da função *Main*

Por último, temos a Tabela 3.8 que mostra o ganho de cada uma das funções após passar por todos os passos. Para as funções(*Enqueue* e *Main*) os resultados foram iguais para ambas as métricas, já para a função *Dijikstra* o ganho em relação ao número de Blocos Básicos ficou melhor do que em relação ao numero de Ligações entre os blocos.

	Blocos Básicos	Ligações entre Blocos
Dijiksta	40,00%	30,77%
Enqueue	33,33%	33,33%
Main	40,00%	40,00%

Tabela 3.8: Ganho de cada função em relação a cada métrica

3.3 Análise dos passos que mudam o CFG

Nesta etapa, primeiramente foi realizado análise da documentação de cada um dos passos(*-simplifyCFG, -loop-simplify, -jump-threading, -gvn, -ipsccp e -loop-rotate*) que causaram alguma alteração durante o processo de otimização, para assim perceber suas funcionalidades e verificar se existe passos que são utilizados como preparação para outro, os quais mesmo fazendo alterações na CFG, servem para facilitar a execução de outro passo.

Isso foi evidenciado com relação ao passo *-ipsccp(Interprocedural Sparse Conditional Constant Propagation)*, que de modo geral é o principal passo dentre todos os outros estudados, devido a quantidade de alterações e de otimizações que ele pode gerar no código intermediário. Por isso escolhemos esse passo para mostrarmos mais detalhadamente o seu funcionamento, que será feito no Capítulo 4.

Antes de descrever sobre cada um dos passos, será trato sobre um formato de representação intermediária chamado *Static Single Assignment*(Atribuição Única Estática), que é muito

utilizado durante o processo de otimização, ficando mais fácil o entendimento dos passos que utilizam-se dessa representação.

Os compiladores escolhem essas representações intermediárias(RI), como estrutura de dados para montar seu código. Essas escolhas podem afetar diretamente no poder e eficiência da suas otimizações. CYTRON et al. (1991) fala que uma má escolha dessa estrutura pode inibir a otimização ou retardar a compilação até um ponto em que as otimizações se tornem indesejáveis.

3.3.1 *Static Single Assignment(SSA)*

Em 1988, ROSEN; WEGMAN; ZADECK (1988) apresentou um modelo de RI chamado *Static Single Assignment(SSA)*, sendo sua representação em forma de um Grafo de Fluxo de Controle(CFG). Segundo (CYTRON et al., 1989), existem duas etapas para fazer a tradução de um programa para o modelo SSA. A primeira delas, é a inserção de instruções especiais de atribuições chamadas de $\phi - functions$ em determinados pontos do programa. Na segunda, cada variáveis *V* recebe novos nomes V_i para vários inteiros i. Cada vez em que *V* é mencionado, é feita a substituição por uma menção dos novos nomes de V_i, deixando o código sem repetições de variáveis, onde cada linha é atribuída a uma variável diferente.

Na Figura 3.4 temos um exemplo de tradução de uma representação básica do programa para uma representação no modelo *SSA*, onde no exemplo (B) da figura, está a primeira etapa da tradução, adicionando a instrução que mostra que a variável V vai ser substituída por um conjuntos de variáveis V_i. Em seguida, no exemplo (C) está a segunda etapa, onde é feita a substituição de todas as menções da variável V por uma variável V_i.

```
if P
then do V ← 4
     end
else do V ← 6
        if Q then return
     end

... ← V + 5
```
(a)

```
if P
then do V ← 4
     end
else do V ← 6
        if Q then return
     end
V  ← φ(V1, V2)
... ← V + 5
```
(b)

```
if P
then do V1 ← 4
     end
else do V2 ← 6
        if Q then return
     end
V  ← φ(V1, V2)
... ← V + 5
```
(c)

Figura 3.4: Exemplo de transformação para o modelo *SSA*. Fonte: (CYTRON et al., 1991)

Apesar desse modelo ser criado em 1988, sua utilização só começou a ser viável em 1989, quando CYTRON et al. apresentou um algoritmo eficiente para a criação desta estrutura de dados, mostrando evidências analíticas e experimentais que a sua solução é linear no tama-

nho do programa. Esse assunto será melhor explicado no capitulo 4 referente ao assunto de Propagação de Constantes.

3.3.2 Simplificação do CFG (*-simplifyCFG*)

O passo *-simplifyCFG* tem como objetivo fazer a eliminação de informações redundantes ou de códigos mortos. No site do projeto *LLVM* são especificados todos os passos feitos nessa otimização, um deles é a eliminação de todos os nós(blocos básicos) sem predecessores, ou seja, os nós que não são possíveis alcançar em nenhuma sequência a partir do nó raiz. Outra etapa é a de união de dois nós, que ocorre com seu predecessor *se e somente se* o seu predecessor tiver apenas um sucessor. O terceiro passo realizado é a eliminação dos $\phi - nodes$ com um único predecessor, esses $\phi - nodes$ são criados na tradução do código para o formato *SSA*, então nesta etapa a otimização é realizada a exclusão desses nós criados. O último passo é a eliminação dos blocos básicos que contém apenas um ramo incondicional (LLVM, 2007).

3.3.3 Canonização de *loops* naturais (*-loop-simplify*)

Este passo de otimização executa transformações para simplificar o formato dos *loops* naturais, deixando as próximas análises e otimizações mais simples e eficazes. Inicialmente é feita a inserção de um pré-cabeçalho nos *loops* garantindo que haja uma única borda de entrada não crítica de fora do *loop* para o seu cabeçalho, outra inserção feita, é a de um bloco básico de saída no *loop*, a qual garante que todos blocos que estão fora do *loop* tenham um predecessor dentro do *loop* estejam ligados a um mesmo bloco básico(sendo dominados pelo cabeçalho do *loop*) (LLVM, 2007).

O próprio projeto (LLVM, 2007) mostra que estes blocos criados vão ser limpados na execução posterior do passo *-simplifycfg*, mostrando que o uso desse passo não deve piorar o código gerado. A modificação do fluxo de controle é clara neste passo, porém ele traz a atualização das informações de *loop*.

3.3.4 Rotação no Loop (*-loop-rotate*)

A otimização *-loop-rotate* tem como objetivo realizar simplificações nos *loops* do programa e quando aplicada a otimização produz um programa eficiente com caminhos de comunicação uniforme, além de descobrir se existe paralelismo no *loop* examinado (WOLFE, 1989).

3.3.5 Salto de segmentação(*-jump-threading*)

Esse passo tenta encontrar segmentos distintos de fluxo de controle que passam por um bloco básico, analisando blocos que possuem múltiplos predecessores e múltiplos sucessores. Á medida que um ou mais dos predecessores do bloco puderem comprovar que sempre causam um salto para um dos sucessores, nós direcionamos a aresta de seu predecessor para o sucessor ao duplicar o conteúdo deste bloco (LLVM, 2007).

3.3.6 Numeração de valor global (*-gvn*)

Este passo executa a numeração de valor global para eliminar instruções totalmente e parcialmente redundantes bem como, executar a eliminação de carga redundante (LLVM, 2007).

A GVN tenta substituir um conjunto de instruções que calculam o mesmo valor com uma única instrução. Esses conjuntos são encontrados procurando por identidades algébricas ou instruções que tenham a mesma operação e entradas idênticas. Este passo depende muito do programa estar no formato (*SSA*), pois o requisito para entradas idênticas é um requisito para valores idênticos, não nomes de variáveis (CLICK, 1995).

4 PROPAGAÇÃO DE CONSTANTE (*-IPSCCP*)

O passo *-ipsccp* faz a propagação de constante inter-procedural utilizando a representação esparsa(SSA), propagando os valores por meio de programa. Está propagação faz com que o fluxo de controle do programa seja alterado, podendo diminuir a sua quantidade de blocos básicos e suas ligações (LLVM, 2007).

Segundo WEGMAN; ZADECK (1991), a propagação de constante vem do problema de análise de fluxo global, com o objetivo de descobrir valores que são constantes em todas as execuções possíveis do programa e propagar esses valores o mais à frente. Expressões que tenham todos os operadores constantes podem ser avaliados em tempo de compilação e podem ser propagados ainda mais à frente.

A utilização desta técnica serve a vários propósitos na otimização de compiladores, WEGMAN; ZADECK (1991) cita alguns:

- Impressões avaliadas em tempo de compilação não precisam ser avaliadas no momento da execução, se tais expressões estiverem dentro de loops, uma única avaliação em tempo de compilação pode salvar muitas avaliações em tempo de execução;

- Códigos que não são executados em nenhuma execução possível, podem ser excluídos. Eles são identificados quando são descobertos ramificações condicionais que sempre ocupam um dos possíveis caminhos da ramificação;

- Detectar esses caminhos nunca tomados, ajudam na simplificação do fluxo de controle do programa;

- Como muitos dos parâmetros para os procedimentos são constantes, o uso da propagação constante com a integração de procedimentos pode evitar a expansão do código que geralmente resulta de implementações ingênuas da integração de procedimentos.

- A propagação consistente pode ser feita em vários domínios, por exemplo, nos campos de tipos de valores.

A Figura 4.1 apresenta um exemplo de propagação de constante. (A) Antes da propagação de constante. (B) Momento em que há a transformação para o modelo *SSA*. No passo (B) para o (C) é feita a propagação de constantes. Já nos passos seguintes é feita a simplificação da

CFG, fazendo a eliminação de código morto(C para D, D para E e E para F) além de transformar desvios condicionais em incondicionais(C para D).

```
a = 5                1
b = 3                1

if b > 2:            1
   a = b             2
   b = a * 2         2
else:
   b = a * 2         3


return b             4
          (A)
```

```
a1 = 5               1
b1 = 3               1

if b1 > 2:           1
   a2 = b1           2
   b2 = a2 * 2       2
else:
   b3 = a1 * 2       3
b4 = phi(b2,b3)      4
a3 = phi(a2,a1)      4
return b4            4
          (B)
```

```
a1 = 5               1
b1 = 3               1

if True :
   a2 = 3            2
   b2 = 3 * 2        2
else:
   b3 = 5 * 2        3
b4 = phi(b2,b3)      4
a3 = phi(a2,a1)      4
return b4            4
          (C)
```

```
a1 = 5               1
b1 = 3               1


a2 = 3               1
b2 = 3 * 2           1


b4 = phi(b2,b3)      1
a3 = phi(a2,a1)      1
return b4            1
          (D)
```

```
b2 = 3 * 2           1


return b2            1
          (E)
```

```
return 6             1
          (F)
```

Figura 4.1: Exemplo de propagação de constante e simplificação do código

As Figuras 4.2 e 4.3 apresentam as CFGs dos códigos da figura 4.1, mostrando que a quantidade de blocos básicos e suas ligações, não alteram conforme é realizado a propagação de constante, somente depois nas etapas (D), (E) e (F), quando acontecem a eliminação de código morto e as transformações dos desvios, sendo este papel do passo *-simplifyCFG*.

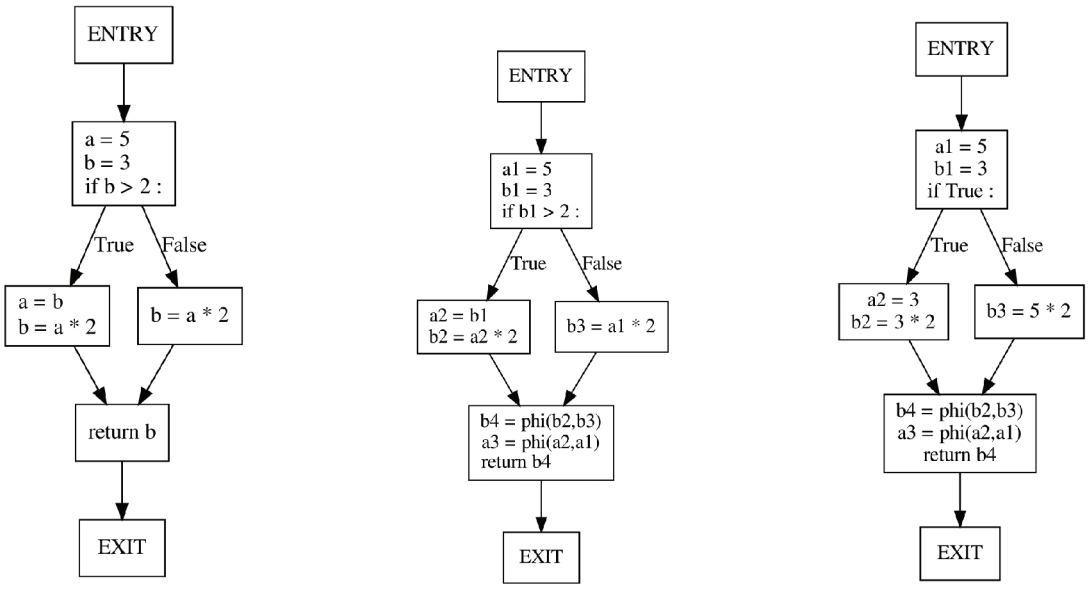

Figura 4.2: CFGs dos códigos A, B e C da Figura 4.1

Contudo podemos verificar que os dados simplificados posterior ao passo de propagação de constante só foi possível devido a suas otimizações, sendo ele o real causador dessas mudanças do código.

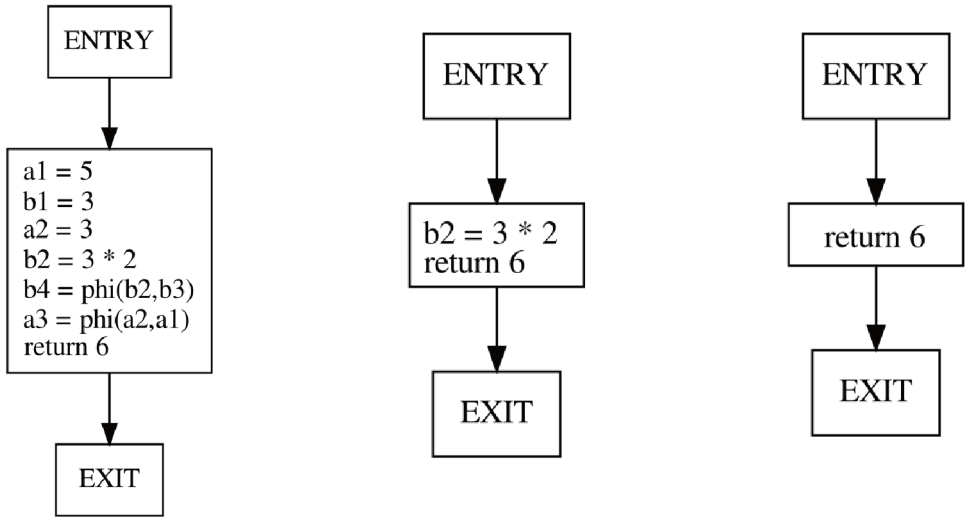

Figura 4.3: CFGs dos códigos D, E e F da Figura 4.1

5 CONCLUSÃO

Os compiladores têm o papel de realizar traduções de códigos em uma linguagem para um código equivalente em outra linguagem, durante esse processo ele realiza passos que fazem uma série de otimizações e análises no programa. Nós trabalhamos em cima desses passos, buscando encontrar quais desses alterariam mais as CFGs do programa.

Pode-se dizer que provavelmente apenas uma métrica não é o suficiente para saber se a CFG muda ou não, sendo melhor enumerar as otimizações que causaram mudanças mais drásticas e compreender o seu funcionamento.

Este trabalho apresentou uma análise das CFGs entre os passos independentes de arquitetura. Identificou-se que após o passo *-simplifyCFG* ocorre a maior mudança em número de Blocos Básicos e Ligações (diminuição no número de ambas as métricas). Entretanto concluímos que o passo que causa estas mudanças é o passo de propagação de constantes(*-ipsccp*). Isto acontece pois todo o resultado da otimização do passo *-ipsccp* fica pendente e é efetuado durante o passo *-simplifyCFG*.

Neste trabalho focamos nossos estudos no efeito do passo *-ipsccp*. Trabalhos futuros poderiam investigar a influência nas mudanças sobre as otimizações realizadas em *loops*, como o *-loop-rotate*.

Além disso nos restringimos nosso estudo nos passos independentes de arquitetura. Outra alternativa seria repetir nossos esforços nos passos dependentes de arquitetura.

REFERÊNCIAS

APIWATTANAPONG, T.; ORSO, A.; HARROLD, M. J. A differencing algorithm for object-oriented programs. In: AUTOMATED SOFTWARE ENGINEERING, 2004. PROCEEDINGS. 19TH INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2004. p.2–13.

BALL, T.; LARUS, J. R. Optimally profiling and tracing programs. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, [S.l.], v.16, n.4, p.1319–1360, 1994.

BINKERT, N. et al. The gem5 simulator. **ACM SIGARCH Computer Architecture News**, [S.l.], v.39, n.2, p.1–7, 2011.

CHAN, P. P.; COLLBERG, C. A method to evaluate CFG comparison algorithms. In: QUALITY SOFTWARE (QSIC), 2014 14TH INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2014. p.95–104.

CLICK, C. Global code motion/global value numbering. In: ACM SIGPLAN NOTICES. **Anais...** [S.l.: s.n.], 1995. v.30, n.6, p.246–257.

CYTRON, R. et al. An efficient method of computing static single assignment form. In: ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 16. **Proceedings...** [S.l.: s.n.], 1989. p.25–35.

CYTRON, R. et al. Efficiently computing static single assignment form and the control dependence graph. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, [S.l.], v.13, n.4, p.451–490, 1991.

KINABLE, J.; KOSTAKIS, O. Malware classification based on call graph clustering. **Journal in computer virology**, [S.l.], v.7, n.4, p.233–245, 2011.

KINDER, J.; ZULEGER, F.; VEITH, H. An abstract interpretation-based framework for control flow reconstruction from binaries. In: INTERNATIONAL WORKSHOP ON VERIFICATION, MODEL CHECKING, AND ABSTRACT INTERPRETATION. **Anais...** [S.l.: s.n.], 2009. p.214–228.

LLVM. LLVM's Analysis and Transform Passes. **https://llvm.org/docs/Passes.html**, [S.l.], nov 2007.

ROSEN, B. K.; WEGMAN, M. N.; ZADECK, F. K. Global value numbers and redundant computations. In: ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 15. **Proceedings...** [S.l.: s.n.], 1988. p.12–27.

SETHI, R.; ULLMAN, J. D.; S. LAM monica. **Compiladores**: princípios, técnicas e ferramentas. [S.l.]: Pearson Addison Wesley, 2008.

VUJOŠEVIĆ-JANIČIĆ, M. et al. Software verification and graph similarity for automated evaluation of students' assignments. **Information and Software Technology**, [S.l.], v.55, n.6, p.1004–1016, 2013.

WEGMAN, M. N.; ZADECK, F. K. Constant propagation with conditional branches. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, [S.l.], v.13, n.2, p.181–210, 1991.

WICHT, B. et al. Hardware Counted Profile-Guided Optimization. **arXiv preprint arXiv:1411.6361**, [S.l.], 2014.

WOLFE, M. Loop rotation. **CSETech**, [S.l.], 1989.

XU, L.; SUN, F.; SU, Z. Constructing precise control flow graphs from binaries. **University of California, Davis, Tech. Rep**, [S.l.], 2009.