Trabalho Prático 1 - Compiladores - 2024-2

Henrique Mendonça Castelar Campos

1. Introdução

No desenvolvimento de software, algoritmos são codificados em linguagens de programação de alto nível, como Java, C, C++, C#, Rust, Go, Kotlin, dentre outras, e depois essa linguagem de alto nível é traduzida para uma forma que possa ser executada pelo computador, seja durante a "tradução" (linguagens interpretadas), seja algum momento depois (linguagens compiladas). Para que esse processo possa ocorrer de forma automatizada, foram desenvolvidos os compiladores.

Um compilador é composto das seguintes partes: *Front* e *Back*. O *Front* de um compilador é dividido nos seguintes componentes: Analisador léxico, analisador sintático e analisador semântico. E o *Back* é composto dos componentes: Gerador de código intermediário, otimizador de código dependente de máquina, gerador de código de máquina e otimizador de código independente de máquina.

O entendimento do funcionamento de compiladores permite o desenvolvimento de novos métodos de otimização de programas, tornando o código final gerado mais eficiente, o desenvolvimento de novas linguagens de programação, permitindo que programadores tenham novas formas de expressarem seus programas, além de possibilitar o aprimoramento das ferramentas existentes. Dessa forma, o estudo de compiladores pode, em última instância, contribuir para o aprimoramento da Computação como um todo.

Neste trabalho, será analisado o funcionamento da parte de otimização de código, analisando os diferentes tipos de otimização realizados no código intermediário. Para isso, foi desenvolvido um *script* (programa de linguagem interpretada), que faz o uso do arcabouço de compilação LLVM (*Low Level Virtual Machine*), para a geração de grafos de fluxo de controle (CFG) nos formatos *.digraph* e *.png*, permitindo o entendimento do processo de otimização de código intermediário, desenvolvido dentro do compilador.

2. Desenvolvimento

O *script* desenvolvido foi feito na linguagem de programação Python. A escolha dessa linguagem se deve pelo fato do Python possuir recursos avançados e de fácil utilização que permitem a manipulação de strings e a manipulação de caminho de arquivos (*path*).

O programa desenvolvido é organizado da seguinte forma: ele possui uma única função *generate_unique_file_path*, e a parte principal. A função *generate_unique_file_path* é responsável por gerar caminhos de arquivo únicos, garantindo que não exista dois arquivos com o mesmo nome. E a parte principal, (*__main__*) é responsável por realizar

```
python ./gen_cfg.py ./exemplo.c
```

Figura 1. Exemplo de comando utilizado para a execução do *script* desenvolvido

processo de chamada do *Front* do Clang, do otimizador OPT, e do gerador de imagem DOT.

Para o desenvolvimento deste programa, foi feito o uso dos seguintes módulos da biblioteca padrão do Python: *pathlib*, *string*, *sys*, *os*, *shutil*, *subprocess*, *tempfile* e *random*. Os módulos *pathlib* e *os* foram utilizados na resolução dos caminhos dos arquivos. Já o módulo *string* foi utilizado na geração de caracteres ASCII aleatórios. O módulo *sys* foi utilizado para a obtenção dos argumentos de linha de comando. Já o módulo *shutil*, foi utilizado para a obtenção do caminho absoluto dos programas de linha de comando utilizados (*clang*, *opt*, *dot* e *llc*).

A utilização do programa é feita através da chamada do interpretador python, seguido do caminho do *script*, e seguido do programa em C a ser compilado. Uma vez executado este *script*, automaticamente serão gerados os arquivos *.digraph* e *.png* para cada função do programa passado por parâmetro, para os diferentes níveis de otimização: *O0*, *O1*, *O2* e *O3*. Cada arquivo gerado tem em seu nome a função analisada e o tipo de otimização aplicada.

Conforme solicitado no enunciado, foram gerados os grafos de fluxo de controle dos programas de exemplo solicitados, tanto no formato *graphviz*, como no formato de imagem PNG. Esses arquivos estão localizados nas pastas *arquivos_.digraph* e *imagens_cfg*, respectivamente. Essas pastas estão organizadas de acordo com a estrutura hierárquica dos arquivos de exemplo.

Durante o processo de geração dos grafos de fluxo de controle, foi encontrado um erro na geração da representação intermediária do Clang para o programa *ReedSolomon* (em *src/singlesource/ReedSolomon.c*). Isso se deve pelo fato de, na linha 60 (‘*static inited = 0;*’), não conter uma declaração de tipo válida para a variável *inited*. A mensagem de erro de compilação foi armazenada no arquivo de texto *nao_compila.txt* em *src/singlesource/ReedSolomon.c/nao_compila.txt*.

3. Análise

Para a análise das técnicas de otimização, foram escolhidos os seguintes módulos: Fibonacci (em *src/singlesource/fib2.c*), Private (em *src/singlesource/private.c*) e Recursive (em *src/singlesource/recursive.c*).

3.1. Fibonacci

Na análise das técnicas de otimização de Fibonacci, foi escolhida a função ‘*fib*’. Isso se fundamenta no fato dela ser a parte mais importante do programa.

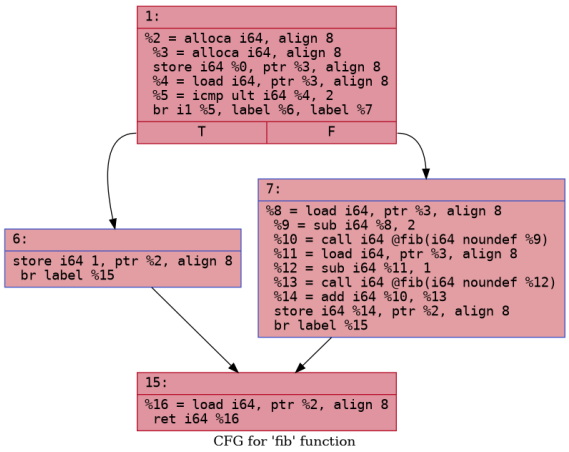

Figura 2. Grafo de fluxo de controle para a função '*fib*' com nível 0 de otimização

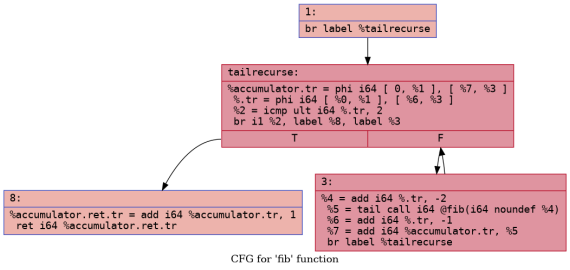

Figura 3. Grafo de fluxo de controle para a função '*fib*' com nível 1 de otimização

No nível 1 de otimização, em comparação com a versão sem nenhuma otimização, é possível perceber que uma das recursões envolvendo a função '*fib*', foi substituída por uma iteração, como é possível perceber, uma vez que a chamada de função, feita pela instrução '*call*', foi substituída pela iteração, feita pela instrução '*br*'.

Em relação ao nível 2 de otimização, se comparado com o nível 1, é possível perceber que foi adicionado uma comparação inicial do valor passado à função '*fib*', por meio da instrução '*icmp*'. Essa comparação garante que se na primeira vez que a função '*fib*' for chamada, caso seja passado um valor menor ou igual a 2, a função não irá executar instruções da recursão, aumentando a eficiência do programa.

E em relação ao nível 3 de otimização, se comparado com o nível 2, não ocorreu nenhuma mudança no grafo de fluxo de controle.

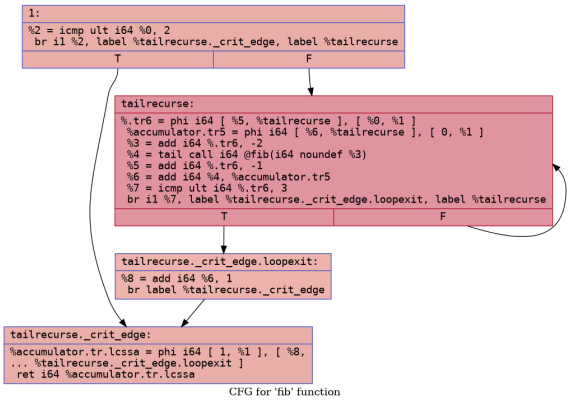

Figura 4. Grafo de fluxo de controle para a função '*fib*' com nível 2 de otimização

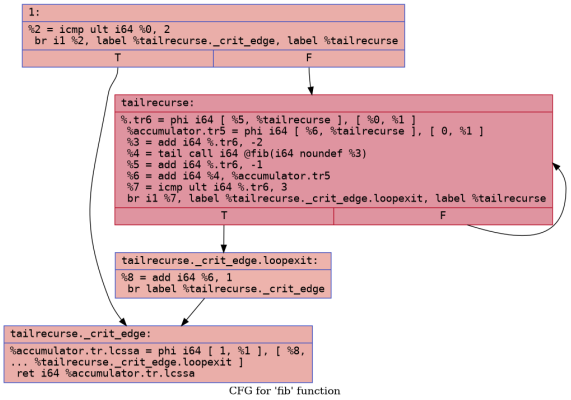

Figura 5. Grafo de fluxo de controle para a função '*fib*' com nível 3 de otimização

3.2. *Private*

Para a análise das técnicas de otimização de *Private*, foi escolhida a função *main*. Isso surge em virtude desta função apresentar um tipo específico de otimização pertinente ao programa *Private*.

Como é possível perceber, no nível 1 de otimização, se comparado com a versão sem otimização, a função '*main*' deixou de executar as instruções de inicialização de uma função. Isso pode ser atribuído ao fato de que na '*main*', nada é executado, portanto essas instruções são desnecessárias.

E em relação aos níveis 2 e 3 de otimização, não houve mudanças em relação ao nível 1.

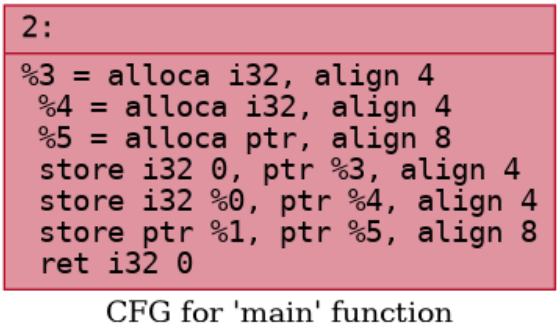

Figura 6. Grafo de fluxo de controle para a função '*main*' com nível 0 de otimização.

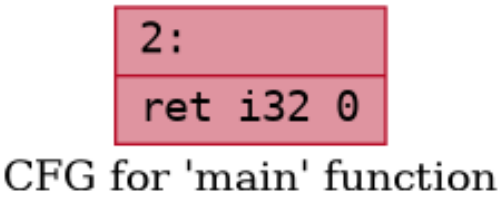

Figura 7. Grafo de fluxo de controle para a função '*main*' com nível 1 de otimização.

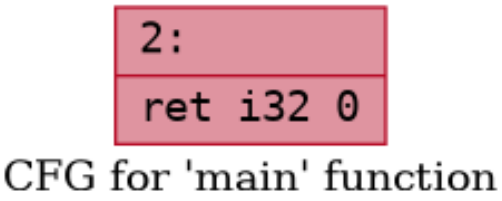

Figura 8. Grafo de fluxo de controle para a função '*main*' com nível 2 de otimização.

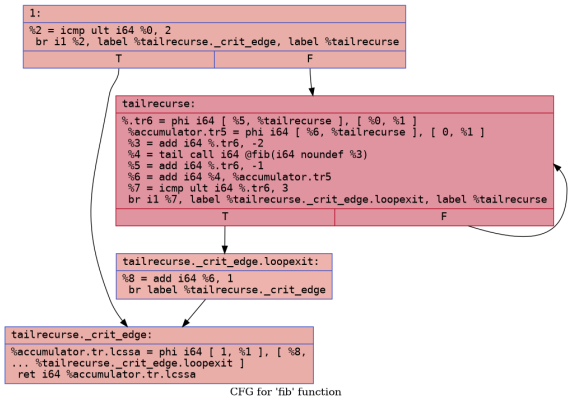

Figura 9. Grafo de fluxo de controle para a função '*main*' com nível 3 de otimização.

3.3. *Recursive*

Em relação ao programa *Recursive*, para a análise das técnicas de otimização, foi escolhida a função '*ack*'. A escolha dessa função deriva do fato dela ser uma das funções nas quais ocorrerão mais recursões.

Uma das otimizações que pode ser percebida, no nível 1, é a redução do número de instruções do tipo '*load*' e '*store*', na *label* filha à esquerda da primeira ramificação. Além disso, é possível notar que a *label* filha à esquerda, na versão de nível 1 de otimização, está realizando diretamente o retorno da função, sem a necessidade de pular para a *label* final. Isso é consequência de que na versão não otimizada, o valor do registrado da *label* mais à esquerda da primeira ramificação, estava sendo armazenado na memória para depois ser carregado pela *label* final, para realizar o retorno. Dessa forma essa otimização remove os *loads* e *stores* desnecessários, tornando o programa mais eficiente, por utilizar menos a memória e mais os registradores do processador.

E em relação à otimização realizada no nível 2, em relação ao nível 1, é possível perceber, que de maneira semelhante com a otimização de nível 2 da função '*fib*' do programa Fibonacci, visto anteriormente, foi adicionado uma comparação inicial do valor passado por parâmetro para a função '*ack*'. Dessa forma, se o valor passado for igual a 0, o fluxo de execução do programa será direcionado para a *label* final, dispensando a necessidade de atribuição estática de valor único, que não seria utilizada caso a função começasse com o valor de x igual à 0.

E em relação ao nível 3 de otimização, se comparado com o nível 2, não ocorreu nenhuma mudança no grafo de fluxo de controle.

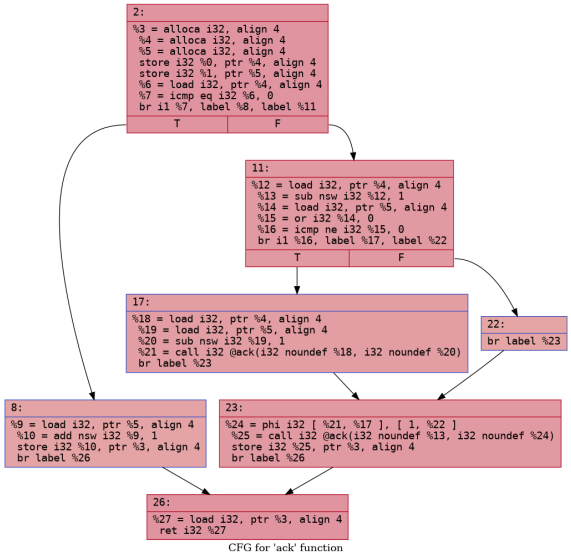

Figura 10. Grafo de fluxo de controle para a função '*ack*' com nível 0 de otimização.

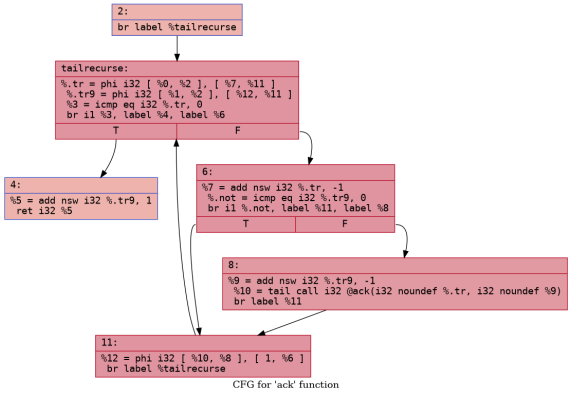

Figura 11. Grafo de fluxo de controle para a função '*ack*' com nível 1 de otimização.

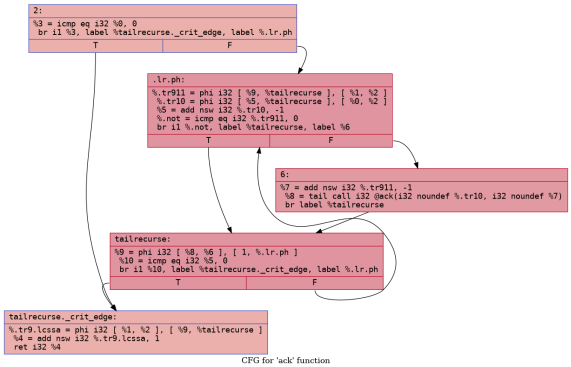

Figura 12. Grafo de fluxo de controle para a função '*ack*' com nível 2 de otimização.

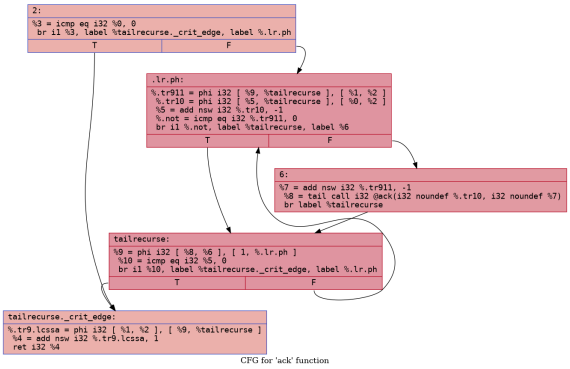

Figura 13. Grafo de fluxo de controle para a função '*ack*' com nível 3 de otimização.

4. Conclusão

Através deste trabalho foi possível compreender o funcionamento básico de um compilador de linguagens de programação, além dos diferentes níveis de otimização que o código intermediário sofre, permitindo, em última instância, a geração de um objeto executável otimizado para a máquina alvo.

Além disso, foi possível compreender, em parte, o funcionamento de uma das ferramentas mais utilizadas na compilação de código fonte, que é o LLVM, o seu funcionamento, e como é possível realizar o desenvolvimento de novos métodos de otimização para o código intermediário.