Cross Compiling e Análise de Código Assembly Gerado na Compilação Com o GNU Compiler Collection – Arquitetura de Computadores

Emanuel Santana Santos

Curso de Engenharia de computação – Universidade Estadual de Feira de Santana – Campus Feira de Santana
Emanuel.santana1514@gmail.com

1. Introdução

Este relatório tem como objetivo demonstrar os resultados da cross compiling (compilação cruzada), que permite gerar código assembly para outras máquinas e ou dispositivos que utilizam processadores de arquiteturas diferentes. E tudo isso através de um único dispositivo. Com estes códigos de diferentes arquiteturas é possível analisar as diferenças entre os códigos assembly gerados para ARM e Intel, uma vez que os códigos gerados para o ARM utilizam mais instruções, porém instruções mais simples baseadas em Reduced Instruction Set Computer (RISC). Enquanto que Intel baseia-se em Complex Instruction Set Computer (CISC).

2. Sistemas Utilizados

Para a confecção do projeto foi solicitado que o sistema operacional utilizado fosse baseado em Linux, então foi utilizado o Linux Ubuntu versão 20.04.1. Este sistema Linux foi utilizado por meio do Virtual Box versão 6.1.12 que possibilitou a utilização sem necessariamente criar um dual boot ou instalação do sistema como principal na máquina utilizada.

Além do sistema operacional também foi solicitado que se usasse o compilador Gnu Compiler Collection (GCC), e este foi utilizado na versão GCC 9.3.0, que é a versão default ao fazer a instalação pelo apt install do sistema Linux. Com este compilador já é possível gerar código assembly para Intel e Amd, mas para gerar código para arquitetura ARM é necessário instalar um cross compiling também do GCC para AARCH64 também da versão 9.3.

Além dos sistemas utilizados acima foram utilizadas opções do compilador que ajudam a otimizar o código gerado, abaixo uma melhor descrição de cada nível de otimização e o que cada uma faz no código.

• -O0: Não realiza nenhuma otimização e cada comando do código-fonte é convertido diretamente na instrução correspondente no arquivo executável.

• -O1: O compilador visa reduzir tamanho do código e o tempo de execução. Permitindo reordenação de instruções, otimizações nos laços de repetição, e quando faz isso elimina código morto ou desnecessário.

• -O2: Aprimora ainda mais que -O1, incluindo otimizações para agendamento de instruções, otimização de chamadas recursivas.

• -O3: Otimização que visa melhorar o tempo de execução da aplicação. Move condições invariantes do laço de repetição, função embutida. Código maior - Mais custoso.

Essas otimizações são feitas através de flags que são ativadas quando cada comando é utilizado na compilação, no nível 2 são utilizadas todas as flags utilizadas no -O0 e -O1 e adiciona outras flags que permitem otimizar ainda mais o código, isso aumenta o tempo de compilação e uso de memória para melhorar ainda mais o código final gerado. Isso também pode ser observado no nível -O3, que também reutiliza as flags dos níveis anteriores e adiciona algumas outras a fim de otimizar ainda mais o código.

3. Experimentos

Diferença do código intel para o arm, otimização e diferença entre os códigos.

Código 1: O primeiro código é um código muito simples onde é criada apenas uma variável Int em C e esta variável é incrementada utilizando ++;

- ARM – O código ARMv8 sem otimização (nível -O0) gerado pelo compilador continha 9 linhas onde todos os passos normais para criação do código foram feitos, mas como essa variável não foi utilizada em nada, foram instruções quase desperdiçadas.

  Com o nível de otimização -O1 a opção ficou mais direita e foi utilizado apenas um Mov para fazer a atribuição mais rápida de valor, reduzindo a duas instruções o código.

  No nível de otimização -O2 e -O3 não houveram mais mudanças que influenciassem o código.

- Intel – O código Intel sem otimização (nível -O0) gerou 9 linhas de instrução assim como o ARM, gerou instruções para um código que não tem utilização.

  Com o nível de otimização -O1 a opção ficou mais direita assim como o ARM e foi utilizado apenas um Mov para fazer a atribuição direta de valor.

  No nível -O2 e -O3 a instrução Mov foi substituída por uma XOR, continuando com duas linhas de instrução, mas substituindo o Mov.

Código 2: O código 2 como exemplo do primeiro código é apenas criação de variável, mas dessa vez float, e também uma incrementação de valor utilizando ++.

- ARM – O código ARMv8 sem otimização (nível -O0) gerado pelo compilador continha 9 linhas onde todos os passos do código foram feitos.

  Com o nível de otimização -O1 a opção ficou mais direita e foi utilizado apenas um Mov para fazer a atribuição mais rápida de valor, reduzindo a duas instruções o código.

  No nível de otimização -O2 e -O3 não houveram mais mudanças que influenciassem o código.

- Intel - O código Intel sem otimização (nível -O0) gerou 13 linhas de instrução, e gerou instruções para um código que não tem utilização.

Com o nível de otimização -O1 a opção ficou mais direita assim como o ARM e foi utilizado apenas um Mov para fazer a atribuição direta de valor.

No nível -O2 e -O3 a instrução Mov foi substituída por uma XOR, continuando com duas linhas de instrução, mas substituindo o Mov.

Código 3: O código 3 como exemplo dos códigos anteriores é apenas criação de variável, mas dessa vez double, e também uma incrementação de valor utilizando ++.

- ARM - O código ARMv8 sem otimização (nível -O0) gerado pelo compilador continha 11 linhas onde todos os passos do código foram feitos.

  Com o nível de otimização -O1 a opção ficou mais direita e foi utilizado apenas um Mov para fazer a atribuição mais rápida de valor, reduzindo a duas instruções o código.

  No nível de otimização -O2 e -O3 não houveram mais mudanças que influenciassem o código.

- Intel - O código Intel sem otimização (nível -O0) gerou 13 linhas de instrução, e gerou instruções para um código que não tem utilização.

  Com o nível de otimização -O1 a opção ficou mais direita assim como o ARM e foi utilizado apenas um Mov para fazer a atribuição direta de valor.

  No nível -O2 e -O3 a instrução Mov foi substituída por uma XOR, continuando com duas linhas de instrução, mas substituindo o Mov.

Código 4: O código 4 como exemplo dos códigos anteriores é apenas criação de variável, mas dessa vez são duas variáveis inteiras, e também uma atribuição de valor utilizando +=.

- ARM - O código ARMv8 sem otimização (nível -O0) gerado pelo compilador continha 14 linhas onde todos os passos do código foram feitos.

  Com o nível de otimização -O1 a opção ficou mais direita e foi utilizado apenas um Mov para fazer a atribuição mais rápida de valor, reduzindo a duas instruções o código.

  No nível de otimização -O2 e -O3 não houveram mais mudanças que influenciassem o código.

- Intel - O código Intel sem otimização (nível -O0) gerou 11 linhas de instrução, e gerou instruções para um código que não tem utilização.

  Com o nível de otimização -O1 a opção ficou mais direita assim como o ARM e foi utilizado apenas um Mov para fazer a atribuição direta de valor.

  No nível -O2 e -O3 a instrução Mov foi substituída por uma XOR, continuando com duas linhas de instrução, mas substituindo o Mov.

Código 5: O código 5 como exemplo dos códigos anteriores é apenas criação de variável, mas dessa vez são duas variáveis float, e também uma atribuição de valor utilizando +=.

- ARM - O código ARMv8 sem otimização (nível -O0) gerado pelo compilador continha 14 linhas onde todos os passos do código foram feitos.

  Com o nível de otimização -O1 a opção ficou mais direita e foi utilizado apenas um Mov para fazer a atribuição mais rápida de valor, reduzindo a duas instruções o código.

  No nível de otimização -O2 e -O3 não houveram mais mudanças que influenciassem o código.

- Intel - O código Intel sem otimização (nível -O0) gerou 14 linhas de instrução, e gerou instruções para um código que não tem utilização.

  Com o nível de otimização -O1 a opção ficou mais direita assim como o ARM e foi utilizado apenas um Mov para fazer a atribuição direta de valor.

  No nível -O2 e -O3 a instrução Mov foi substituída por uma XOR, continuando com duas linhas de instrução, mas substituindo o Mov.

Código 6: O código 6 como exemplo dos códigos anteriores é apenas criação de variável, mas dessa vez são duas variáveis double, e também uma atribuição de valor utilizando +=.

- ARM - O código ARMv8 sem otimização (nível -O0) gerado pelo compilador continha 14 linhas onde todos os passos do código foram feitos.

  Com o nível de otimização -O1 a opção ficou mais direita e foi utilizado apenas um Mov para fazer a atribuição mais rápida de valor, reduzindo a duas instruções o código.

  No nível de otimização -O2 e -O3 não houveram mais mudanças que influenciassem o código.

- Intel - O código Intel sem otimização (nível -O0) gerou 14 linhas de instrução, e gerou instruções para um código que não tem utilização.

  Com o nível de otimização -O1 a opção ficou mais direita assim como o ARM e foi utilizado apenas um Mov para fazer a atribuição direta de valor.

  No nível -O2 e -O3 a instrução Mov foi substituída por uma XOR, continuando com duas linhas de instrução, mas substituindo o Mov.

4. Resultados

Após concluir a análise percebe-se que a “lógica” como o compilador gera os códigos assembly para cada arquitetura é parecida, mas há mudanças em nomes de registradores e comandos entre as arquiteturas, é possível perceber também que o compilador utiliza instruções específicas para fazer as operações com double, int e float. Pelo menos isso é utilizado no nível -O0 de otimização, após isso quando há outras otimizações o

compilador reduz drasticamente o código fazendo apenas o que é realmente necessário no código.

5. Referências

GCC -Documentação sobre opções de otimização – Disponível em: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html