Analisador de Dependências

Assis T. Fabiani, Marcelo L. Onhate, Mateus L. Nava
URI – Universidade Regional Integrada do Alto Uruguai e Das Missões
Erechim – RS – Brasil
Departamento de Ciências da Computação
assis_fabiani@hotmail.com
onhate.marcelo@yahoo.com
mate42@gmail.com

Abstract. *A code in the language of low level MIPS can generate several dependencies between the lines of command when executed on a pipeline of depth 5 (five). These dependencies can be classified as dates (RAW, WAR and WAW) and control (caused by determinate or indeterminate ties). Basically, 2 (two) techniques constitute the data dependencies, one for control (Stalls) and other for elimination (Forwarding). Then, for try to supply this possible idle ties in the pipeline, this article is propose an algorithm in JAVA language, which the user through the graphical interface, for more interaction, enter your code and afterward, get access to the dependencies, total of execution cycles and a reorganization proposal.*

Resumo. *Um código na linguagem de baixo nível MIPS pode gerar várias dependências entre as linhas de comando quando executadas em um pipeline de profundidade 5 (cinco). Estas dependências podem ser classificadas como dados (RAW, WAR e WAW) e controle (ocasionadas por laços determinados ou indeterminados). Basicamente, 2 (duas) técnicas constituem as dependências de dados, uma para controle (Stalls) e outra para eliminação (Forwarding). Então, para tentar suprir esta possível ociosidade no pipeline, neste artigo é disponibilizado um algoritmo na linguagem JAVA, da qual, o usuário através da interface gráfica, para maior interação, digita seu código e, por consequência, obtém o acesso as dependências, total de ciclos de execução e a reordenação proposta.*

1. Introdução

Com base nos conhecimentos adquiridos ao longo da disciplina de Arquitetura de Computadores II, neste artigo será desenvolvido um código na linguagem Java com o intuito de analisar instruções MIPS dispostas em um pipeline de profundidade cinco, e posteriormente dispor suas dependências, das quais empregam: dependências de fluxo, dependências de saída e antidependências.

Após a análise ser finalizada, com seus stalls (bolhas) e forwading (repasses) necessários, será sugerida uma forma de reordenação do código com a finalidade de reduzir o número de ciclos de execução e, posteriormente calcular as dependências e os ciclos necessários novamente.

Para interagir mais fácil com o usuário será feita uma interface gráfica, da qual, por meio desta é digitado o código MIPS respectivo, e por conseqüência, é mostrado na tela às dependências, ciclos de execução e uma possível reordenação de código. As

seções ao longo do artigo estão dispostas para mostrar à implementação do analisador, que consiste na explicação do algoritmo.

2. Implementação

A parte de implementação dos algoritmos de Análise de Dependência com Bolha e Análise de Dependência por Reordenação, baseiam-se basicamente em verificar a dependência de uma instrução em uma lista e tomar as devidas providências com os resultados obtidos desta análise. Posteriormente isto será abordado mais amplamente, mas antes precisamos algumas definições.

O que é uma instrução para os algoritmos: um comando ADD, SUB ou LW, podendo possuir de zero a três registradores, um de saída e dois para entrada como no caso do ADD saída, entrada1, entrada2, ou um de saída e um de entrada como no LW saída, entrada1, ou nenhum registrador como no J 200. Existem ainda casos onde a ordem dos registradores é "inversa", ao contrário do comum, como é o caso do BEQ e do BNE onde é BEQ entrada1, entrada2, saída.

Essa instruções são introduzidas em um pipeline, que nada mais é, uma fila de instruções, com tamanho fixo a serem executadas, ao inserir um item na fila do pipeline remove-se o primeiro, ou seja, entra uma instrução no fim do pipeline e sai uma do começo do mesmo.

A análise de dependência de uma instrução em uma lista envolve ter conhecimento de que:

- ➔ Em que ciclo de execução de uma instrução os valores já estão disponíveis para serem repassados à instrução subseqüente.
- ➔ Em que ciclo é dado à entrada dos valores dos registradores para que a instrução seja executada.

Com estes dois valores é possível saber quantos ciclos a segunda instrução depende da primeira, por exemplo:

Dadas as instruções C1 = ADD $t0, $t1, $t2 e C2 = SUB $t2, $t0, $t3.

C2 tem as seguintes dependências contra C1: Dependência de Fluxo e Antidependência;

Sabendo que dependência de Antidependência não gera ciclos ociosos porque, o ciclo em que a leitura é feita sempre será menor que o ciclo em que a escrita será feita, só nos resta calcular quantos ciclos C2 depende de C1 pela Dependência de Fluxo então:

ADD ciclo de repasse 3 (três), SUB ciclo de entrada 2 (dois), fazendo *"3 – 2 - 1 = 0"* não temos ciclos de dependência ente C2 e C1, isso não que dizer que não haja dependência, só que ela não afeta na execução. Mas porque o **-1** (um)? Este -1 (um) significa quantos ciclos já foram executados desde a entrada de C1 até C2 entrar no pipeline, como este ciclo já foi executado é descontado da quantidade de ciclos para repasse.

Já se tivermos C1 = LW $t0, 8($t2) e C2 = SUB $t2, $t0, $t3.

Onde LW ciclo de repasse 4 (quatro) e SUB ciclo de entrada 2 (dois), temos *"4 – 2 – 1 = 1"*, existe dependência de 1(um) ciclo entre C1 e C2 então para C2 executar

após C1 ela deve esperar um ciclo, ai que entra a particularidade de cada algoritmo, o Bolha requer a inserção de 1(uma) instrução vazia para executar, já a Reordenação busca outra instrução não dependente de C1 e nem de C2 para ocupar este ciclo ocioso.

2.1 Algoritmo

2.1.1 O algoritmo de Análise de Dependência com Bolha

Este algoritmo tem a lógica de verificar quantos ciclos uma instrução depende de uma lista de outras instruções e gerar bolhas (“Stalls”) com este valor.

Ao iniciar temos uma lista completa de instruções (instruções originais), que são carregadas. Estas instruções são o código assembly a ser analisado, da qual, pega-se a primeira instrução da lista e verifica se existe dependência de algum dos seus registradores com os do pipeline.

O retorno da análise desta primeira instrução contra a lista de instruções do pipeline é a quantidade de ciclos que a instrução comparada depende do pipeline, com este resultado gera-se um mesmo número de bolhas ou instruções nulas que são inseridas no pipeline, assim esta instrução comparada não depende mais do pipeline e pode ser posta para execução no mesmo. Ao ser inserida no pipeline esta instrução é removida da lista de instruções originais. Então, segue-se o ciclo de análise pegando próxima instrução da lista original e comparando com o pipeline.

2.1.2 O algoritmo de Analise de Dependência por Reordenação

Do mesmo jeito que na anterior, temos uma lista original que é comparada com o pipeline, só que se houver alguma quantidade de ciclos dependente não são inseridas instruções vazias para “esperar o tempo passar”, mas sim é buscada uma próxima instrução que é comparada com o pipeline, e após com a lista de espera, que contém todas as instruções que tiveram dependência do pipeline e estão esperando para execução.

Diferente da comparação do pipeline que retorna quantos ciclos se depende do mesmo, na comparação com lista de espera basta somente saber se depende ou não de outra instrução, porque se depender não é possível reordenar, mesmo não dependendo em quantidade de ciclos, porque como é possível executar uma instrução que depende de outra que ainda nem foi executada? Não é possível.

Neste algoritmo ainda existem outras particularidades, como instruções que não podem ser reordenadas, ou seja, só podem ser executadas depois que todas as outras instruções anteriores a ela já foram executadas, um exemplo são J, JR e o JAL.

3. Conclusão

A análise de dependência de dados na linguagem assembly (arquitetura MIPS) possui um papel fundamental para o desempenho de uma lista de instruções. Com ela podemos suprir algumas ociosidades no pipeline através da reordenação de código e, posteriormente, obter um melhor tempo de execução.

Com isso, foi percebido em alguns testes uma redução substancial no total de ciclos de execução comparados antes e após a reordenação, levando é claro, sempre em consideração as restrições apresentadas nas instruções requeridas no artigo.

O código foi constituído totalmente na linguagem JAVA, visando apresentar algumas das principais instruções e registradores da linguagem MIPS e seu respectivo comportamento em um pipeline de profundidade 5 (cinco).

Então, com base neste artigo, desenvolver um código na linguagem MIPS requer um cuidado extra para quem deseja obter um melhor desempenho do código, e por conseqüência, uma melhor execução.

4. Referencias

PATTERSON, A. DAVID e HENNESSY, L. JOHN **Organização e Projetos de Computadores, A Interface Hardware/Software, 3º Edição.**

DE ROSE, F. A. CÉSAR e NAVAUX, A. O. PHILIPPE **Arquiteturas paralela, Nº 15.** Instituto de Informática da UFRGS.