OTIMIZAÇÃO DE CÓDIGOS MIPS EM PROCESSADORES COM PIPELINE

Luiz Felipe Faria Rodrigues, Gabryel Martins Assis

[1]Departamento de Ciência da Computação – Universidade Federal de Roraima (UFRR)
69310-970 – Boa Vista – RR – Brasil

Luizfelipefr243@gmail.com, gabryelmartins777@gmail.com

Abstract. *This paper presents a theoretical analysis of the impact of data hazards on the performance of processors based on the MIPS architecture. The study focuses specifically on the Load-Use Hazard and the software optimization technique known as Instruction Reordering. By comparing cycle-by-cycle timing diagrams, it is demonstrated how static instruction scheduling eliminates pipeline stalls, improving throughput and bringing the CPI (Cycles Per Instruction) closer to the theoretical ideal.*

Resumo. *Este artigo apresenta uma análise teórica sobre o impacto de conflitos (hazards) de dados no desempenho de processadores baseados na arquitetura MIPS (Microprocessor without Interlocked Pipeline Stages). O estudo foca especificamente no conflito de "Carga de Uso" (Load-Use Hazard) e na técnica de otimização via software conhecida como Reordenamento de Instruções. Através da comparação de diagramas de tempo ciclo a ciclo, demonstra-se como o escalonamento estático de instruções elimina bolhas (stalls) no pipeline, melhorando o throughput e aproximando o CPI (Ciclos por Instrução) do ideal teórico.*

1. Introdução

Processadores modernos baseados na arquitetura MIPS utilizam o pipeline para explorar o paralelismo em nível de instrução (ILP). Contudo, a eficiência desse modelo é frequentemente comprometida por dependências de dados e controle (hazards), que inserem pausas (stalls) indesejadas no fluxo de execução. Este trabalho analisa o desempenho em um pipeline clássico de cinco estágios, com o objetivo de identificar gargalos e demonstrar como técnicas de otimização de software, especificamente o reordenamento de instruções, podem mitigar latências e minimizar o número total de ciclos sem exigir alterações no hardware.

2. Estudo teórico

Pipeline MIPS de 5 estágios:

- IF – Instruction Fetch
- ID – Instruction Decode
- EX – Execute
- MEM – Memory Access

- WB – Write Back

Existem quatro tipos de dependências:

- RAW – Read After Write
- WAR – Write After Read
- WAW – Write After Write
- Dependência de controle

2.1 Metodologia e feraramente desenvolvida

Para automatizar a análise proposta, foi desenvolvido um simulador em linguagem Python (`MIPSPipelineSimulator`). O script processa o código Assembly MIPS e simula o comportamento de um pipeline de 5 estágios.

A ferramenta realiza:

- **Detecção de Dependências:** Identifica quando uma instrução precisa de um dado que ainda não foi gravado (RAW).
- **Inserção Automática de Bolhas:** Se houver um *Load-Use Hazard* ou latência de operação (ex: Divisão), o script insere NOPs automaticamente.
- **Contagem de Ciclos:** Calcula o total exato de ciclos para execução.

3. Análise e Otimização de Códigos MIPS

3.1.A)

ADD $t0, $t1, $t2 ------Escreve $t0

SUB $t3, $t0, $t4 -------Lê $t0

AND $t5, $t3, $t6 ------Lê $t3

OR $t7, $t5, $t8 ---------Lê $t5

Análise:

Dependência em cadeia (RAW): Este é o pior caso de RAW, cada instrução depende imediatamente do resultado da instrução anterior.

Ex.: A instrução 2 precisa de $t0, a instrução 3 precisa de $t3.

Sem forwarding haveria 2 bolhas entre cada instrução. Com forwarding as bolhas são eliminadas, mas não há paralelismo real aproveitado.

Otimização:

Não é possível reordenar este bloco isoladamente pois a lógica é sequencial.

3.2.B)

ADD R1, R2, R3 ------Escreve R1

SUB R4, R1, R5 ------Lê R1

AND R6, R1, R7 ------Lê R1

OR R8, R1, R9 --------Lê R1

XOR R10, R1, R11 ---Lê R1

Análise:

Dependência de difusão: A instrução ADD produz R1, e todas as instruções subsequentes precisam ler R1.

A instrução SUB tem um hazard RAW crítico, tenta ler R1 logo após a escrita.

A instrução AND pode ter conflito dependendo da implementação do forwarding.

As instruções OR e XOR provavelmente executarão sem problemas, pois já estará disponível no banco de registradores ou via forwarding quando chegarem à execução.

Otimização:

Inserir instruções independentes entre ADD e R1, se existirem.

A instrução SUB é a única que causaria uma bolha sem forwarding.

3.3.C)

Loop:

L.D F0, 0(R1) ---------Carrega F0 da memória

ADD.D F4, F0, F2-------Usa F0 imediatamente

S.D F4, 0(R1) ----------Usa F4

DADDUI R1, R1, #-8 —Atualiza o índice

BNE R1, R2, Loop

Análise:

Load-Use hazard: Entre L.D e ADD.D. O dado de F0 só está disponível no final do estágio MEM, mas ADD.D precisa dele no início do estágio EX. Mesmo com forwarding isso gera obrigatoriamente 1 bolha.

Otimização:

Podemos mover instruções independentes para preencher a bolha após L.D.

Ex.:Mover DADDUI R1, R1, #-8 para logo após L.D.

Ao mover, o valor de R1 muda antes do S.D, que precisaria ter seu offset ajustado.

3.4.D)

DIV.D F0, F2, F4 --------Divisão (Latência muito alta)

ADD.D F10, F0, F8 -----Depende de F0 da Divisão

SUB.D F12, F8, F14 ----Independente (Usa F8 e F14)

Análise:

A instrução DIV.D é extremamente lenta.

A instrução ADD.D ficará parada esperando o resultado de F0.

Otimização:

Mover SUB.D para antes do ADD.D para que ela seja executada enquanto o processador espera a divisão.

3.5.E)

DIV.D F0, F2, F4 -----------Escreve F0 (Lento)

ADD.D F6, F0, F8 ---------Lê F0, Escreve F6

S.D F6, 0(R1) -------------Lê F6

SUB.D F8, F10, F14 ------Escreve F8

MUL.D F6, F10, F8 --------Lê F8, Escreve F6

Análise:

RAW: ADD.D espera DIV.D. S.D espera ADD.D.

SUB.D é independente.

WAW: ADD.D e MUL.D escrevem em F6. Se o pipeline permitir execução fora de ordem, o MUL.D não pode terminar antes do ADD.D usar o valor antigo ou escrever o novo.

Otimização:

Mover SUB.D para logo após DIV.D. Isso esconde parte da latência da divisão.

3.6.F)

ADD R4, R5, R6 ---------Aritmética simples

BEQ R1, R2, EXIT ------Desvio condicional

OR R7, R8, R9 ----------Instrução seguinte

Análise:

Hazard de Controle: Após buscar o BEQ, o processador não sabe qual será a próxima instrução até que a condição R1 == R2 seja avaliada

Se o processador errar a predição, ele terá que descartar a instrução OR e quaisquer outras que entraram no pipeline, perdendo ciclos.

Otimização:

Em arquiteturas MIPS clássicas, a instrução logo após o Branch é sempre executada. O compilador deve colocar uma instrução útil que deve rodar independente do desvio, ou um NOP se não houver opção. Neste caso, se o OR for útil, ele está bem posicionado.

4. Comparação de Desempenho

Tabela 1. Comparação de desempenho antes e após otimização

Código	Nº de Instruções	Ciclos Antes	Ciclos Depois	Redução de ciclos
(A)	4	8	8	0
(B)	5	9	9	0
(C)	5	10	9	1
(D)	3	17	17	0
(E)	5	21	19	2
(F)	3	7	7	0

5. Conclusão

A implementação do simulador em Python permitiu visualizar concretamente o impacto das dependências de dados no desempenho do processador MIPS. Foi possível confirmar que: Hazards de Dados (Load-Use) são os mais comuns e podem ser resolvidos via software através do reordenamento de instruções independentes (como feito no Caso C). Instruções de longa latência (como Divisão) travam o pipeline por muitos ciclos, sendo candidatos ideais para técnicas avançadas como execução fora de ordem ou Loop Unrolling.

6. Referências

HENNESSY, J. L.; PATTERSON, D. A. Computer Architecture: A Quantitative Approach. 6. ed. San Francisco: Morgan Kaufmann, 2019.

PATTERSON, D. A.; HENNESSY, J. L. Computer Organization and Design MIPS Edition: The Hardware/Software Interface. 5. ed. Waltham: Morgan Kaufmann, 2014.

STALLINGS, W. Arquitetura e Organização de Computadores. 10. ed. São Paulo: Pearson Education do Brasil, 2017.

MIPS OPEN. MIPS32 Architecture for Programmers Volume II: The MIPS32 Instruction Set. Imagination Technologies, 2014. Disponível em: https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00086-2B-MIPS32BIS-AFP-6.06.pdf. Acesso em: 12 jan. 2026.