Desvios atrasados e predição dinâmica

Lucas

17 de outubro de 2015

pg 380 Assuma a seguinte forma de predição (forma básica): "desvios não são tomados". Nesse caso predizemos que *branches* (ble, beq, bne, begz, etc) não são tomados, e quando nós estamos errados nessa hipótese? então descarregamos o pipeline quando estivermos errados a respeito dessa forma básica.[a] Em um pipeline agressivo um preditor estático vai gastar muita performance (degradar performance). O custo da penalidade por *branches* mau preditos, *branches* preditos erroneamente é em instruções perdidas, no pipeline isso degrada a vazão de instruções.

Uma abordagem é olhar para o endereço da instrução do desvio que já foi processado pelo datapath e verificar se ele **foi tomado**, e se foi mesmo, começa a buscar novas instruções do mesmo lugar da última vez. Essa técnica é chamada predição dinâmica de desvios.

Uma implementação dessa abordagem é criar uma fila de predições dos desvios que já foram processados pelo datapath, também é chamado de tabela histórico dos desvios.[b].

Nessa predição 1-bit (0-não tomado, 1-tomado): mesmo se o desvio é sempre tomado, nós podemos predizer incorretamente 2 vezes, ao invés de apenas uma vez, quando um desvio não for tomado ("e foi predito que era para ser tomado").

Considere o exemplo de um loop que garantidamente é tomado 9 vezes. Porém o preditor está com seu histórico vazio. Assim, ele começa predizendo 'não-tomado' e verifica que foi tomado (pois no histórico estava como foi tomado), o preditor segue 8x predizendo corretamente, porém na décima predição ele errada, pois estava no histórico que antigamente aquele esse branch fora tomado, errando assim duas vezes.

t	t	t	t	t	t	t	t	t	n
n	t	t	t	t	t	t	t	t	t

(a) Primeiro o preditor considera não tomado, mas foi tomado.

(b) O preditor dinâmico, então, registra no histórico que foi tomado

(c) Aí então ele considera tomado as próximas iterações.

[a] Para o pipeline de 5 estágios predições em tempo de compilação são suficientes, porém pipelines mais profundos requerem um preditor mais avançado.

[b] É um jeito simples de ordenara a filha (buffer), a máquina não tem certeza se a predição está correta na tabela histórico de desvios.

(d) E então na última iteração, que era para ser não tomado, o preditor considera tomado por causa do histórico. Errou duas vezes, ao invés de uma vez.

pg 381 Idealmente, a acurácia do preditor iria combinar com a frequência de desvios, considerando eles regulares, isto é não variam. Para remediar essa fraqueza, projetistas de hardware usam comumente: 2-bit prediction scheme. Veja que agora a predição precisa errar duas vezes antes de mudar a tabela histórico dos desvios.

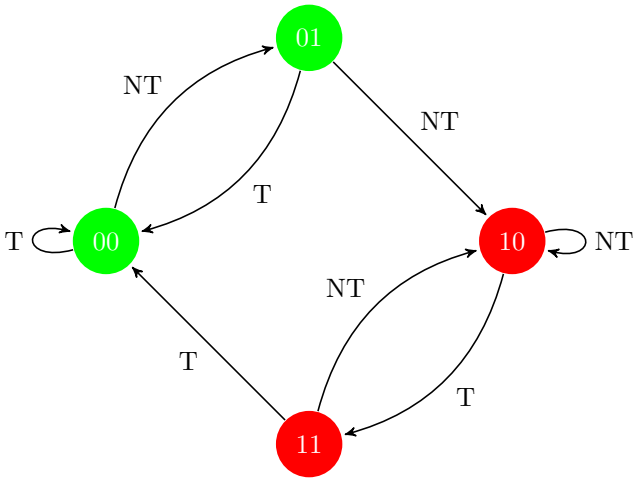

Figura 1: Esquemático da predição dinâmica usando 2 bits. Os estados em verde "gravam"predição de desvio tomado na tabela histórico de desvios, e os estados em vermelho "atualizam" predição de desvio não tomado, na mesma tabela (exemplo: começando no estado '10'—histórico vazio, e então o desvio foi tomado ele pula para o estado '11'—atualiza a tabela (NT), na próxima iteração foi tomado novamente—atualiza a tabela (T), pula para o estado '10' e acerta todas até o final do loop, predizendo corretamente $\frac{9}{10}$ do tempo.)

A implementação, vista a grosso modo, se da pelas alternativas (1): tabela histórico de desvios é feita em cache dedicada, indexada pelos bits menos significativos do endereço de desvio, e o acesso acontece no estágio de Busca da Instrução (IF); alternativa (2) a tabela histórico de desvios é embutido na cache de instruções e exitem um campo para armazenas par de bits extras para o controle. O esquema de dois bits é uma instancia generalizada de um contador-preditor, ele incremente quando uma predição é bem sucedida e decrementa quando uma predição é mau sucedida.[c]

pg 382 Um branch com atraso vai sempre executar a próxima instrução, mas a segunda instrução seguinte do branch vai ser afetada pelo branch. Compiladores e montadores tentam colocar uma instrução que sempre executa após o branch no campo de atraso do *branch*[d]. As limitações em agendar branch atrasado vem de (1) a restrição sobre a instrução que foi acomodada dentro do campo de atraso e (2) nossa habilidade de predizer em tempo de compilação quando um branch é

[c]Implementando um preditor de 2 bits, quando temos um branch que fortemente é tomado ou é fortemente não tomado, estatisticamente os *branches* são assim. Ou eles são muito tomados, ou são muito não tomados, esse preditor de 2-bits se da melhor em preencher a tabela histórico.

[d]Campo de atraso do *branch*: é o campo diretamente depois da instrução *branch* com atraso, a qual na arquitetura MIPS é preenchido por uma instrução que não afeta o branch.

provavelmente tomado ou não. *branch* atrasados funcionam bem em pipelines de 5 estágios, mas quanto mais estágios maior fica o campo de atraso do branch, e pipelines que emitem múltiplas instruções aumentam ainda mais o campo de atraso do branch, e acaba que um único campo de atraso, nesses pipelines mais modernos, é insuficiente. Branch atrasado perdeu a popularidade, comparado com abordagens caras porém dinamicamente flexíveis[e].

Elaboração: Um preditor de desvios nos diz quando um branch é tomado e quando não é. mas ainda requer cálculos do alvo para onde se deslocar. Em um pipeline de 5 estágios, esse cálculo **leva 1 ciclo**, significando que *branches* tomados irão ter **1 ciclo** de penalidade associado. Os 3 tipos de *delayed branches* são uma maneira de eliminar essa penalidade de 1 ciclo, veja:

a. Preenchimento com uma instrução independente e anterior ao branch, essa estratégia é a melhor de todas.

```
+--------------------+
| add $s1, $s2, $s3  |
| if $s2 = 0 then    |
|   [delay slot]     |
+--------------------+

+--------------------+
| if $s2 = 0 then    |
|   add $s1, $s2, $s3 |
+--------------------+
```

b. Instrução vindo a partir do endereço alvo do branch, **b.** é usado com **a.** não é possível de se fazer. Existe uma dependência de dados para com $s1—por isso eu não poderia mover o `add` para dentro do *delay slot*. Essa estratégia é preferível quando existem branch de alta probabilidade de acontecer como em *branch-loop*.

```
        +--------------------+
target: | sub $t4, $t5, $t6  |
        |     ...            |
        | add $s1, $s2, $s3  |
        | if $s1 = 0 then    |
        |   [delay slot]     |
        +--------------------+

        +--------------------+
target: |     ...            |
        | add $s1, $s2, $s3  |
        | if $s1 = 0 then    |
        |   sub $t4, $t5, $t6 |
        +--------------------+
```

[e] Paralelamente, o crescimento de maior números de transistores por chip tem feito possível baratear o custo de predição dinâmica.

c. Vindo de dentro do branch, **c.** é usado com **a.** não é possível de se fazer. Veja que eu não posso mover `add` para o campo de atraso, pois $s1 é modificado/atualizado logo antes do branch, que por sua vez utiliza o $s1 para verificar se é igual a zero, existe uma dependência de dados. Só é possível mover instruções de dentro do branch para o *delay slot* se obrigatoriamente elas não afetam de nenhum jeito a maneira como o branch irá desvios o fluxo de execução.

```
+---------------------+
| add $s1, $s2, $s3   |
| if $s1 = 0 then     |
|   [delay slot]      |
|   ...               |
|   sub $t4, $t5, $t6 |
+---------------------+

+---------------------+
| add $s1, $s2, $s3   |
| if $s2 = 0 then     |
|   sub $t4, $t5, $t6 |
+---------------------+
```

pg 383 Outra abordagem que iremos discutir é a utilização de cache para salvar o destino do PC ou salvar a instrução destino, usando enfileiramento de desvios alvo (branch target buffer)[f]. O esquema de predição com 2-bits dinâmico usa apenas informação sobre um branch em particular. Pesquisadores notaram que usando informações sobre ambos o *branch* local, e o comportamento global dos *branches* ajudam a aumentar a acurácia da predição, usando o mesmo número de bits para no preditor. Esses preditores são chamados preditores correlatos. Um preditor correlato[g] poderá ter 2-bit para cada branch. E mais bits extra para índice do comportamento global dos *branches*. Uma descoberta mais recente é o uso de torneio do preditor, que usa múltiplos preditores, traçando, para cada branch, qual preditor levanta o melhor resultado. Um torneio[h] típico pode conter duas predições para cada índice de branch: m é baseado na informações de comportamento local e o outro na global. Um seletor poderia escolher qual preditor usar, para qualquer predição dada. O seletor pode operar similarmente para 1-bit ou 2-bit preditor, alternando entre esse dois preditores tem mostrado ser mais acurado. Alguns microprocessadores essa elaboração.

Elaboração: Um caminho para reduzir o número de desvios condicionais é adicionar instruções condicionais de `moção`. Ao invés de mudar o PC com um desvio condicional, a instrução condicionalmente muda o registrador destino do `move`. Se a condição falhar, o move atua como um nop. Por exemplo, uma versão do MIPS tem duas novas instruções chamadas `movn` (move if not zero) e `movz` (move if zero). Então, `movn $8, $11, $4` copia o conteúdo do registrador 11 para o registrador 8, "sabendo que o valor do registrador 4 não é negativo"; do contrário, não acontece nada.

O ARM tem um campo condicional na maioria das suas instruções. Isso foi feito sabendo que os programas ARM poderiam ter menos desvios condicionais que programas MIPS.

[f] Estrutura que salva na cache o PC destino, ou instrução destino para um branch qualquer. É mais custoso que um simples preditor enfileirador.
[g] É um preditor de branches que combina comportamento de desvios locais e globais (recentemente executados)
[h] Um conjunto de preditores, cada um fazendo predições para cada branch e há uma seleção do preditor que se sai melhor predizendo branches.

Questão Mova 1 instrução para o *delay slot* se possível para evitar desperdiçar 1 ciclo de cálculo do endereço do branch, e exponha o método que você utilizou.

a.

```
        ...
        ADD R1,R2,R3
        SW R1,20(R2)
        BEQ R3,R0,Label
        [delay slot]
        OR R2,R1,R0
        ADD R1,R1,R3
        ...
LABEL:
```

Nesse código acima é possível que eu faça a estratégia (1), mover uma instrução anterior para dentro do delay slot. Sabendo que o valor de R1 (atualizado) pelo primeiro ADD, vai ser colocado em 20(R2). E sabendo que o branch é tomado. Caso o branch não for tomado o valor de R1 tem que ser salvo em 20(R2).

b.

```
       LA R1,pyrc
       LW R1,0(R2)
       BEQ R1,R0,Label
       [delay slot]
       ORI $v0,R3,0x4
LABEL: ADD R5,R10,R3
```

Nesse caso podemos fazer a estratégia número (2), visto que a estratégia (1) não pode ser utilizada nesse ensaio. A estratégia (2) visa transferir uma instrução do alvo do branch para o delay slot. No caso a instrução a ser movida é o ADD R5,R10,R3. Nesse ensaio LABEL é o alvo do branch.