Especulação para “Resolver” Alias Dependency

Luciana Bulgarelli Carvalho
RA: 981561
Instituto de Computação
Universidade Estadual de Campinas
Av. Albert Einstein, 1251
Campinas/SP – Brazil
+55 (19) 3521-5838
luciana_bc@yahoo.com.br

RESUMO

Alias dependency indica que há dependências entre posições de memória acessas por instruções de load e de store. A identificação de alias dependency é díficil porque só são realmente conhecidas em tempo de execução [1, 2, 4, 6, 9]. A especulação tem sido utilizada na “resolução” de alias dependency para possibilitar um aumento na exploração de ILP. Há dois tipos de especulação: a implementada por hardware e a implementada por software.

Este trabalho descreve quatro técnicas de especulação implementada por hardware para “resolver” alias dependency – dependence prediction, address prediction, value prediction e memory renaming – e a comparação do desempenho destas técnicas [2, 3, 5]. Além disso, um exemplo de especulação implementada por software para “resolver” alias dependency é apresentado [4].

Categorias e Descrição de Assunto

Arquitetura de computadores.

Termos Gerais

Processadores, especulação.

Palavras Chaves

Especulação por hardware, especulação por software, alias dependency.

1. INTRODUÇÃO

A especulação é uma das técnicas mais utilizadas na exploração de paralelismo em nível de instrução – ILP (instruction-level parallelism) [6, 9]. Na especulação, algumas “propriedades” da instrução especulada (por exemplo: a instrução i) são “supostas” durante a sua execução. Assim, é possível aumentar a exploração de ILP permitindo que instruções que dependem da instrução i iniciem sua execução antes da sua conclusão.

Entretanto, as suposições feitas pela especulação podem estar incorretas. Por isso, a especulação também precisa verificar se as suposições feitas estão corretas, além de desfazer a execução das instruções executadas incorretamente e garantir que a execução correta das instruções seja realizada – mispeculation recovery [2, 6, 4, 9].

Além disso, a especulação é responsável por “divulgar” as exceções geradas por instruções executadas especulativamente. As exceções geradas por uma instrução especulada só são “divulgadas” quando se tem certeza que tal instrução deveria ter sido executada, ou seja, quando o resultado da especulação é conhecido.

Alias dependency são as dependências (dependências de dado e de saída e as anti-dependências) entre posições de memória acessadas por instruções de load e de store em um programa [1, 2, 4, 6, 9]. A técnica mais conservadora para “resolver” alias dependency é serializar as instruções de load e as de store. Entretanto, esta técnica pode resultar em muitos ciclos de espera desnecessários devido às falsas dependências. Para possibilitar uma maior exploração de ILP, a especulação é utilizada para “resolver” alias dependency.

Este trabalho apresenta algumas das técnicas de especulação por hardware utilizadas na “resolução” de alias dependency. Além de um exemplo de especulação por software que também permite “resolver” o problema de alias dependency.

2. ESPECULAÇÃO POR HARDWARE

A especulação por hardware é implementada através de uma extensão do algoritmo de Tomasulo [6, 9].

Algumas das técnicas de especulação por hardware que permitem “resolver” o problema de alias dependency são dependence prediction, address prediction, value prediction e memory renaming [2, 5]. A dependence prediction pode ser implementada através das técnicas blind, wait ou store sets. A address prediction e a value prediction podem ser implementadas pelas técnicas last address (value) prediction, stride, context e hybrid.

A especulação por hardware é encontrada nos processadores superscalar especulativo com dynamic scheduling, por exemplo: Pentium III/4, MIPS R10K, Alpha 21264, HP PA 8500 e IBM RS64III [6, 9].

2.1 Dependence Prediction

A dependency prediction especula se cada instrução de load é independente das instruções de store anteriores. Na verdade, esta técnica especula se há ou não dependência verdadeira entre o load e os stores anteriores [2, 3, 5, 8]. Se a especulação indicar que o load é independente dos stores anteriores, então o load é executado antes destes stores. Porém, se a

especulação indicar que o load depende de um ou mais stores anteriores, a execução do load precisa esperar a execução destes para que não haja violação de alias dependency.

Algumas das técnicas utilizadas na implementação de dependence prediction são blind, wait e store sets. A técnica mais simples é a blind prediction. Nesta técnica, os loads são sempre especulados como independentes dos stores anteriores, logo um load pode ser executado antes dos stores anteriores a ele. Mas, se durante a execução do programa é verificado que esta especulação é incorreta, então o procedimento de mispeculation recovery é executado.

Na técnica wait, adiciona-se um bit (o bit wait) a cada instrução do programa. Se a instrução é um load, então este bit auxilia na especulação da dependência deste load em relação aos stores anteriores a ele. No início da execução do programa, o bit wait indica que o load é independente dos stores anteriores (o bit wait é zero), portanto o load pode ser executado antes destes stores (da mesma forma que acontece na técnica blind). Entretanto, se a especulação for incorreta (o load depende de um store), o bit wait deste load é setado e o procedimento de mispeculation recovery é executado. Se, ao longo da execução deste programa, este load for executado novamente, então o seu bit wait indica que ele depende dos stores anteriores (o bit wait é um), portanto a sua execução deve esperar a execução dos stores anteriores.

A técnica store set é semelhante à técnica wait, porém a primeira técnica mantém um store set para cada load ao invés do bit wait. O store set de um load contém os stores que são anteriores a ele e que acessam a mesma posição de memória deste load. Os store sets estão vazios no início da execução do programa e, portanto, indicam que cada load é independente dos stores anteriores a ele. Neste caso, o load pode ser executado independentemente dos stores anteriores. Porém, se ocorre uma mispeculation (um load é dependente de um store), o store em questão é incluído no store set deste load e a mispeculation recovery é executada. Se este load for executado novamente, então a sua execução deve esperar a execução dos stores indicados pelo seu store set.

Dentre estas técnicas, a store sets apresentou o melhor desempenho para a implementação da dependence prediction [2].

2.2 Address Prediction

A técnica address prediction propõe a solução antecipada de alias dependency através da especulação do endereço da posição de memória a ser lida por um load [2, 5]. De posse do endereço especulado, a execução do load precisa esperar apenas a execução dos stores que são anteriores a ele e que escrevem no endereço especulado.

Enquanto o load é executado com o endereço especulado, o endereço real – endereço especificado na instrução – é calculado. Quando o cálculo do endereço real termina, o endereço especulado e o real são comparados. Se os dois endereços forem diferentes, então a especulação está incorreta, portanto a mispeculation recovery é executada e o load é executado com o endereço real.

As técnicas last address prediction, stride, context e hybrid são exemplos de técnicas de address prediction. A técnica last address prediction especula que o endereço de memória lido por um load é igual ao endereço utilizado na execução anterior deste load. Por isso, esta técnica precisa armazenar o endereço de memória lido em cada load.

Além de armazenar o endereço de memória lido em cada load, a técnica stride prediction armazena a diferença entre este endereço e o endereço lido na execução anterior do mesmo load – esta diferença é denominada stride. Nesta técnica, o valor do endereço especulado é o endereço lido na última execução deste load mais a diferença dos endereços lidos na última e na penúltima execução deste load.

A técnica context prediction utiliza os *n* últimos endereços de memória lidos na execução de um mesmo load para especular o próximo valor do endereço de memória a ser lido por deste load. Por exemplo, em [2] o valor de *n* é quatro.

Na técnica hybrid prediction, a especulação é feita por stride prediction ou context prediction, e a técnica adotada é selecionada através de confidence counters.

A stride address prediction apresentou o melhor desempenho quando os endereços de memória lidos por um mesmo load têm o mesmo stride (uma diferença constante). Por exemplo, na leitura de vetores. Por outro lado, a context address prediction apresentou o melhor desempenho quando os endereços de memória lidos não possuem o mesmo stride. Por exemplo, nas operações entre ponteiros de memória.

2.3 Value Prediction

A value prediction especula o valor lido de uma posição de memória através de uma instrução de load [2, 5]. Portanto, as instruções que dependem do resultado deste load podem ser executadas antes do valor lido da memória – valor real – estar disponível. Entretanto, a execução do load continua até que o valor real esteja disponível. Neste momento, o valor real é comparado com o valor especulado. Se os dois valores forem diferentes, então a especulação está incorreta, portanto a mispeculation recovery é executada utilizando o valor lido da memória.

Alguns exemplos de técnicas utilizadas para implementar value prediction são: last value prediction, stride, context e hybrid. Estas técnicas são equivalentes às respectivas técnicas de address prediction.

Os resultados em [2] mostram que o desempenho da técnica stride e da context quando empregadas separadas é inferior ao desempenho da técnica hybrid, que combina o uso destas duas técnicas.

2.4 Memory Renaming

A memory renaming é outra técnica de especulação implementada por hardware [2, 5]. Esta técnica identifica as dependências entre instruções de load e de store com a finalidade de "entregar" o valor armazenado na memória por um store às instruções de load que lêem esta mesma posição de memória bypassing a memória.

A implementação da técnica memory renaming adotada em [2] contém: a store cache que armazena os stores executados recentemente; a store/load cache que armazena as dependências entre loads e stores; o value file para viabilizar memory renaming ou value prediction; o mecanismo para definir quando utilizar value prediction.

O funcionamento do memory renaming é exemplificado de acordo com as posições e os valores da store cache, da store load/cache e do value file indicadas nas Figuras 1 e 2.

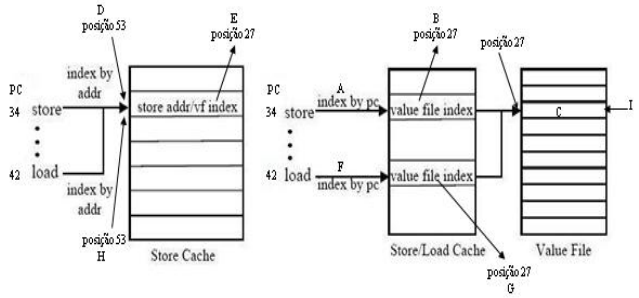

Figura 1. Memory renaming quando o load depende do store anterior.

Quando um store é decodificado, o valor do seu PC acessa uma posição da store/load cache (Figura 1 – etapa A) e o conteúdo desta posição indica uma posição do value file (Figura 1 – etapa B).

Se a posição 34 da store/load cache não indica uma posição do value file (houve um miss no value file para este store), então a posição de value file menos utilizada é alocada para este store. De acordo com a Figura 1, a posição 27 do value file é alocada para o store armazenado no endereço 34 da memória, portanto a posição 34 da store/load cache é atualizada com o valor 27.

Na próxima etapa (Figura 1 – etapa C), o valor a ser escrito na memória pelo store ou o endereço da instrução que calcula este valor é armazenado na posição 27 do value file.

Na etapa D da Figura 1, o endereço da posição de memória a ser escrita por este store torna-se disponível (endereço 53 da memória). Na seqüência, a posição 53 da store cache é atualizada para indicar a posição 27 do value file (Figura 1 – etapa E).

Por outro lado, quando um load é decodificado, o valor do seu PC acessa a store/load cache (Figura 1 – etapa F). Se esta posição da store/load cache indica uma posição do value file (Figura 1 – etapa G), então o conteúdo da posição 27 do value file pode ser utilizado na value prediction do valor a ser lido da memória.

Na etapa H da Figura 1, o endereço da posição de memória a ser lida por este load tornar-se disponível (endereço 53 da memória). Se a posição 53 da store cache referencia uma posição do value file (posição 27), então há uma dependência entre este load e o store anterior (Figura 1). Senão, este load é independente do store anterior a ele (Figura 2).

No caso em que o load é dependente do store anterior (Figura 1), a posição 42 da store/load cache é atualizada para indicar a posição 27 do value file (Figura 1 – etapa G), assim ambas as instruções de load e de store acessam a mesma posição 27 do value file a partir da store/load cache. Portanto, quando a posição 27 do value file é atualizada com o valor a ser escrito pelo store na posição 53 da memória (Figura 1 – etapa I), o valor a ser lido pelo load torna-se acessível na posição 27 do value file. Desta forma, o load tem acesso ao conteúdo da posição 53 da memória sem acessar a mesma (bypassing a memória).

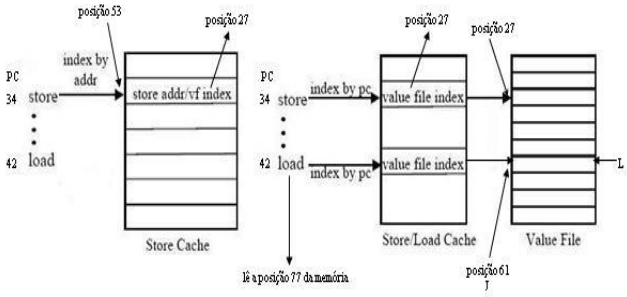

Figura 2. Memory renaming quando o load é independente do store anterior.

No caso em que o load é independente do store anterior (Figura 2), a posição 42 da store/cache indica a posição 61 do value file (Figura 2 – etapa J) e não a mesma posição do store (posição 27). Quando o valor na posição 77 da memória estiver disponível, a posição 61 do value file é atualizado com este valor (Figura 2 – etapa L). Assim, na próxima vez que este load for executado, o valor na posição 61 do value file pode ser utilizado em uma value prediction.

2.5 Resultados

A referência [2] comparou o desempenho de quatro técnicas de especulação implementadas em hardware (store set dependence prediction, hybrid address prediction, hybrid value prediction e memory renaming) e de todas as combinações possíveis entre estas técnicas. Os principais resultados foram:

- Individualmente, a hybrid value prediction apresentou o melhor desempenho.
- O desempenho da hybrid value prediction é bem próximo ao desempenho da store set dependence prediction, mas é significativamente superior ao desempenho da memory renaming.
- Quando combinadas duas a duas, o melhor desempenho foi alcançado pela store set dependence prediction combinada com a hybrid value prediction. A combinação da store set dependence predicton com a hybrid address prediction apresentou o segundo melhor desempenho.
- O desempenho da combinação das quatro técnicas (store set dependence prediction, hybrid address prediction, hybrid value prediction e memory renaming) não é significativamente melhor que o desempenho da combinação da store set dependence prediction com a hybrid value prediction. Entretanto, a implementação da combinação das quatro técnicas requere duas vezes mais espaço de armazenamento que a implementação da combinação destas duas técnicas.
- As técnicas de especulação implementadas por hardware mais promissoras são store set dependence prediction, hybrid address prediction e hybrid value prediction. A técnica store set dependence prediction

tem o maior destaque devido ao desempenho alcançado e à pequena quantidade de hardware requerida em sua implementação.

3. ESPECULAÇÃO POR SOFTWARE

A especulação por software é implementada pelo compilador e, diferente da especulação por hardware, na especulação por software, o compilador modifica o código original, mas sem alterar a sua funcionalidade [1, 4, 6, 9]. Estas modificações incluem desde a adição e a remoção de instruções até a movimentação de instruções no código fonte, além de possíveis alterações nos operandos especificados nas instruções.

De uma forma geral, na especulação por software, o compilador adiciona ao código original o speculation check (verificação da especulação) e a mispeculation recovery. No speculation check, as instruções acrescentadas verificam se a suposição adotada pela especulação é correta ou não. Se a suposição adotada pela especulação é incorreta, as instruções da mispeculation recovery "desfazem" a execução das instruções que foram executadas incorretamente e executam as instruções corretas com o resultado da especulação.

A especulação por software é encontrada nos processadores VLIW (very long instruction word) – por exemplo: Trimedia e i860 – e nos EPIC (explicity parallel instruction computers) – por exemplo: Itanium e Itanium 2 [1, 4, 6, 7, 9].

3.1 General Compiler Framework [4]

O compiler framework proposto em [4] é um exemplo de como a especulação implementada por software "resolve" alias dependency e, então, possibilita um aumento na exploração de ILP.

Neste exemplo, é importante destacar que as dependências entre posições de memória são identificadas através de uma regra heurística simples. De acordo com esta regra, se duas posições de memória apresentam o mesmo access path e as variáveis que determinam o access path não são explicitamente modificadas entre os acessos a estas duas posições de memória – Figura 3 (a), então estas duas posições de memória são consideradas dependentes. Caso contrário – Figura 3 (b), elas são consideradas independentes.

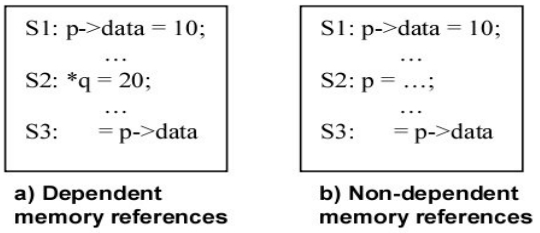

Figura 3. Exemplos da heurística utilizada para determinar se dois acessos à memória são dependentes (a) ou independentes (b) [4].

A referência [4] também exemplifica como a especulação por software permite a movimentação de instruções mesmo quando há dependência entre duas posições de memória. A Figura 4 mostra três casos onde a movimentação de instruções pode ser realizada devido ao uso da especulação para "resolver" alias dependency.

Case 1:	Case 4:	Case 6:
Before DSCM: st [r1] = r2 ld r3 =[r4]	Before DSCM: st [r1] = r2 ld r3 = [r4]	Before DSCM: st [r1] = r2 st [r3] = r4
After DSCM: S1: flag =1 S2: ld r3 = [r4] S3: if overlap(r1, r4) flag =0 S4: st [r1]= r2 S5: if (flag == 0) S6: ld r3=[r4]	After DSCM: S1: flag =1 S2: if overlap(r1,r4) flag =0 S3: if (flag==0) S4: st[r1] = r2 S5: ld r3 = [r4] S6: if (flag == 1) S7: st [r1] = r2	After DSCM: S1: flag =1 S2: if overlap(r1, r3) flag =0 S3: st [r3] = r4 S4: if (flag == 1) S5: st [r1] = r2

Figura 4. Exemplos onde a especulação possibilita a movimentação de instruções [4].

No caso 1 da Figura 4, há uma dependência verdadeira entre as instruções store e load, pois r1 e r4 podem indicar a mesma posição de memória durante a execução deste código. A especulação garante que não há violação de dependência quando o load for movido para antes do store. No código resultante (after DSCM), o flag é setado (S1) para indicar que o load pode ser movido para antes do store e, então, o load é executado (S2). Em S3, a suposição da especulação (o store e o load não acessam a mesma posição de memória, ou seja, r1 é diferente de r4) é verificada. Se a suposição for incorreta (o store e o load acessam a mesma posição de memória), então o load não pode ser movido para antes do store, por isso o flag é zerado. O store é executado (S4). As instruções S5 e S6 correspondem à mispeculation recovery. Se a suposição for incorreta – o flag é zero (S5), então o load (S6) é executado conforme o código original, ou seja, após o store (before DSCM).

O caso 2 da Figura 4 apresenta a mesma dependência verdadeira do caso 1. Entretanto, no caso 2 a especulação garante que não há violação de dependência quando o store for movido para depois do load. No código resultante (after DSCM), o flag é setado (S1) para indicar que o store pode ser movido para depois do load. Em S2, a suposição da especulação (o store e o load não acessam a mesma posição de memória, ou seja, r1 e r4 são diferentes) é verificada. Se a suposição for incorreta (o store e o load acessam a mesma posição de memória), então o store não pode ser movido para depois do load, portanto o flag é zerado. As instruções S4 e S5 implementam a mispeculation recovery. Se a suposição for incorreta – o flag é zero (S3), então o store (S4) deve ser executado antes do load como é feito no código original (before DSCM). O load é executado (S5). Mas, se o store pode ser executado depois do load (S6) – a suposição está correta, então o store é executado (S7).

No caso 3 da Figura 4, há uma dependência de saída entre as instruções de store, pois r1 e r3 podem indicar a mesma posição de memória durante a execução deste código. A especulação garante que não há violação de dependência

quando o primeiro store (st [r1] = r2) for movido para depois do segundo (st [r3] = r4). No código resultante (after DSCM), o flag é setado (S1) para indicar que o primeiro store pode ser movido para depois do segundo. Em S2, a suposição da especulação (os dois stores não escrevem na mesma posição de memória, ou seja, r1 e r3 são diferentes) é verificada. Se a suposição for incorreta (os dois stores escrevem na mesma posição de memória), então o primeiro store não pode ser movido para depois do segundo, por isso o flag é zerado. O segundo store é executado (S3). Se a suposição da especulação for correta – o flag é um (S4), então o primeiro store é executado após o segundo.

Neste último caso, não há mispeculation recovery, porque o primeiro store não pode ser movido para depois do segundo, quando os dois escrevem na mesma posição de memória. Portanto, o primeiro store não precisa ser executado, pois o valor escrito pelo segundo store sobrepõe o valor escrito pelo primeiro.

Além disso, é importante destacar que a arquitetura IA64 – implementada por exemplo no processador Itanium 2 [7] – oferece duas grandes vantagens para a implementação de especulação por software. A primeira vantagem são instruções que dão suporte à implementação da especulação [4], por exemplo: ld.a (advanced load) e chk.a (advanced load check).

A segunda vantagem é a disponibilização de predication [1, 4]. As instruções predicadas permitem ao compilador restruturar o código do programa de uma forma mais eficiente do que quando o programa contém apenas branches convencionais [1, 4, 9].

4. CONCLUSÃO

A "resolução" de alias dependency é um dos principais desafios para aumentar a exploração de ILP nos processadores. Felizmente, a especulação tem sido utilizada nesta tarefa com sucesso.

Este trabalho descreveu algumas das técnicas de especulação implementadas por hardware e uma análise de desempenho entre elas. Além disso, um exemplo de especulação por software implementado no processador Itanium 2 também foi apresentado.

Uma sugestão de tema para uma pesquisa futura é a integração de técnicas de especulações por hardware e por software na "resolução" de alias dependency em busca de um melhor desempenho.

5. REFERÊNCIAS

[1] August, D., Connors, D., Mahlke, S., Sias, J., Crozier, K., Cheng, B., Eaton, P., Olaniran, Q., Hwu, W. 1998. *Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture. Proceedings of the 25th Annual International Symposium on Computer Architecture* (ISCA'98), 227-237.

[2] Calder, B., Reinman, G. 2000. *A Comparative Survey of Load Speculation Architectures. Journal of Instruction-Level Parallelism*, 1-39. University of California, San Diego.

[3] Chrysos, G., Emer, J. 1998. *Memory Dependence Prediction using Store Sets. 25th Annual International Symposium on Computer Architecture*, 142-153.

[4] Dai, X., Zhai A., Hsu W., Yew P. 2005. *A General Compiler Framework for Speculative Optimizations Using Data Speculative Code Motion. International Symposium on Code Generation and Optimization* (CGO), 280-290.

[5] Embry, A., Towles, B., Erez M., 2000. *Memory disambiguation and speculation. Processor Architecture – Advanced Computer Organization. Lecture* #9, 1-4. Stanford University.

[6] Hennessy, J., Patterson, D. 2003. *Computer Architecture: A Quantitative Approach*. 3ª edição, capítulo 3 – *Instruction-Level Parallelism and its Dynamic Exploitation*, capítulo 4 – *Exploiting Instruction Level Parallelism with Software Approaches*.

[7] Itanium from Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Itanium

Data do último acesso: 24/05/2010.

[8] Önder, Soner. *Cost Effective Memory Dependence Prediction using Speculation Levels and Color Sets*. 2002. *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[9] Patterson, D. e Hennessy, J. 2007. *Computer Organization and Design – The Hardware/Software Interface*. 3ª edição, capítulo 6 – *Enhancing Performance with Pipelining*, 434-435.