Counting Dependence Predictors

Franziska Roesner, Doug Burger e Stephen W. Keckler. Counting Dependence Predictors. ISCA 2008, pp 215-226, 2008.

Ricardo Dutra da Silva (RA 089088)

Previsores de dependências para despachar *loads* são necessários em microprocessadores de alta performance. Alguns previsores existentes usam técnicas como tabela *Load-Wait*, *Tabela de História de Colisão* (CHT), *Store Sets* e *Exclusive Collision Predictor* (ECP). Esses previsores, em geral, dependem de um ponto central de despacho ou de informações de execução centralizadas, o que torna difícil sua utilização em novas microarquiteturas distribuídas. O artigo propõe um previsor para esse tipo de microarquiteturas chamado *Counting Dependence Predictor* (CDP), no qual toda informação está disponível localmente ou é feita disponível usando menos mensagens adicionais quanto possível.

A predição usa uma tabela indexada pelo PC que indica o número de *stores* que um *load* deve esperar através de três estados: agressivo (zero, *load* executa logo que seu endereço esteja disponível), N-store (N *stores* chegados antes ou depois do *load*), conservativo (todos os *stores* na ordem do programa). O artigo mostra que é possível predizer esse número com testes sobre o SPEC2000. Ao contrário de outros previsores, o CDP prediz *loads* dinâmicos para um número arbitrário de *stores*. Quando uma violação ocorre o pipeline é esvaziado.

A implementação usa uma tabela hash que contém dois bits para representar um dos três estados. Cada entrada é iniciada em estado agressivo. Quando em estado agressivo o *load* deveria ter esperado por um *store*, uma violação é disparada e a entrada da tabela é atualizada para conservativo. Quando em estado conservativo, o número de *stores* que executam e conflitam com o *load* são contados e a entrada é atualizada se menos de dois são contados. Quando em N-store, o número de *stores* conflitantes também é contado e quando atinge N o *load* é liberado. Caso a contagem não some N, muda-se N pois o compostamento está conservativo.

Quando o número de *stores* dependentes varia em estâncias dinâmicas de um *load*, a atualização da tabela flutua muito. Gravando alguns bits do PC do *store* quando há violação do *load*, na próxima execução do *load* para o estado, verifica-se se existe uma estância do *store* que gerou a violação. Isso permite que *loads* não esperem todos *stores* quando não é dependente.

O CDP foi simulado e avaliado na microarquitetura TFlex mas pode ser usado em outras arquiteturas. TFlex é uma arquitetura *fully distributed tiled* de 32 cores com bancos de *load-store* múltiplos e distribuídos. O tamanho de despacho é de até 64 e janela de execução de 2096 instruções com até 512 *loads* e *stores*. Decisões de controle e despacho de instruções e predição de dependência podem ocorrer em *tiles* diferentes. Para isso, um protocolo distribuído para lidar com predição eficiente é necessário. Os protocolos de controle para despacho, término de instruções e commit operam em blocos de 128 instruções. As estruturas não são centralizadas, sendo particionadas por endereço. Cada bloco é assinalado para um core baseando-se em seu endereço inicial e as instruções são divididas entre cores participantes baseando-se em IDs e na fila de *load-store* (LSQ). Caches de dados são particionadas conforme os endereços de *load*/*store*.

O artigo descreve o protocolo de mensagens e mostra como a predição de dependências distribuída pode ser eficientemente executada com poucas perdas em performance (1 – 2%) em relação a um CDP ideal, centralizado e sem latência de roteamento. Para confirmar a correção de especulações e término de *stores* relacionados com determinado *load*, é preciso um conjunto de mensagens simples e com pouca latência. Quatro tipos de mensagens são usadas: para informar dependências, informar quando *stores* completam, liberar *loads* para execução e esvaziar pipelines. Nos experimentos, cada *load* requisita 0.28 mensagens em média. Para a melhor configuração do CDP, sem latências de mensagens, a performance melhora apenas 1%.

O protocolo implementa a previasão de dependência no lado da memória, após um *load* ser despachado e enviado para o banco de cache de um core. Uma alternativa seria a predição ao lado da execução, antes do despacho. No entanto o protocolo de mensagens fica mais complexo. Sem considerar latências de mensagens o ganho de tal implementação chega a apenas 2%.

Os resultados experimentais foram rodados usando um subconjunto do SPEC2000 (9 Integer and 8 FP), simulados com *single SimPoints* de 100 milhões de instruções, em um simulador da microarquitetura TFlex. Em geral, foram usados 16 cores, com janela de execução de até 2048 instruções com até 512 instruções de memória. A penalidade de previsões erradas é de 5 a 13 ciclos para identificação do erro, esvaziamento do pipeline e reinicio do despacho.

Foram avaliadas as estratégias de previsão Conservativa, Agressiva, Load-Wait, CDP, Store Sets e ECP. Na média, o CDP errou na previsão de menos *loads* independentes do que os outros esquemas. No entanto, quando o CDP fica muito conservativo, ele pode ser mais custoso que Store Sets ou ECP. No CDP, quando um *load* é predito dependente corretamente e deve esperar um *store*, ele não é atrasado desnecessariamente por mais do que este *store*, diferentemente de ECP e Store Sets.

Na comparação de performance a melhor configuração do CDP alcança 92% de acerto, enquanto o previsor Agressivo alcança 76% e o Load-Wait 81%. A execução conservativa é a pior, com 45%. Store Sets chegam a 97% e ECP a 94%, mas o modelo implementado assume acesso centralizado para despacho e execução, sem latência adicional de mensagens, o que seria preciso para distribuir os previsores no modo do CDP. A maioria dos *loads* espera zero ou um *store*, e fazê-los esperar todos os *stores* anteriores faz perder paralelismo. O CDP é melhor em vários benchmarks, quando erra menos na predição de *loads* independentes. Mas o erro de predição de *loads* dependentes e subsequentes esvaziamentos de pipeline degradam a performance geral. Com otimizações propostas no artigo, o CDP melhora a performance de 4% a 4.9%.