Paralelismo em Nível de Instrução: Abordagem por Software

Se você estiver lendo este texto e encontrar algum erro, por favor envie por email para ra008849@ic.unicamp.br

Grupo:

André Lauar Sampaio Meirelles	RA: 008072
Carlos Eduardo Fernandes	RA: 008275
Gustavo Sverzut Barbieri	RA: 008849
Ivens Prates Telles Alves	RA: 008908

no trabalho final, retirar o modo ''draft'' do \docummentclass

Para fazer um comentário no código, utilize: \comment{comentário}

Termos em inglês devem estar em itálico: \textit{english term}

Referências bibliográficas devem ser colocadas em trabalho.bib (abra no emacs que ele ajuda a preencher!) e citadas no texto com \cite{referencia}, depois deve-se rodar (em ordem!): latex trabalho; bibtex trabalho; latex trabalho

Sumário

Começo: Parte do André

1 Introdução

Paralelismo em nível de instruções é uma técnica que vem sendo desenvolvida já desde a década de 40. Basicamente existem duas maneiras de se aumentar a performance de um processador: uma tecnologia de semicondutores mais rápida ou um processamento paralelo eficiente. Neste trabalho será descrita uma forma de processamento paralelo em um único processador. Os processadores em questão são os da arquitetura VLIW(*Very Long Instruction Word*), e as técnicas de paralelismo estudadas serão duas, ambas tendo uma abordagem por *software*: *software pipelining* e *trace scheduling*.

2 Histórico Evolutivo da Arquitetura

A tecnologia VLIW é nasceu em meados dos anos 70, como um desenvolvimento natural do microcódigo horizontal. Esse microcódigo horizontal era usado para gerar os sinais de controle nas primeiras CPU[1]s, que possuiam instruções complexas de serem executadas. Eles eram guardados em uma memória *read-only* de alta velocidade.

Em contraste a essa complexidade de microcódigo horizontal surgiu o conceito de microcódigo vertical, que usa pequenas instruções ou séries de microinstruções para gerar os sinais de controle.

Em 1979, Joseph Fisher, um dos pioneiros do VLIW, estava trabalhando em microcódigo horizontal e se viu com sérias dificuldades para manter e usar essa tecnologia para uma arquitetura de microcódigo de 64 bits e resolveu procurar meios de gerar um microcódigo longo a partir de vários pequenos. Isso deu origem ao *trace scheduling*. Fisher percebeu que esse método podia ser usado por um compilador para gerar código para os processadores VLIW a partir de um código de programa sequencial. Essa descoberta levou ao desenvolvimento do processador ELI-512 e do compilador Bulldog.

No início as máquinas equipadas com processadores VLIW eram destinadas ao meio científico e da engenharia, os quais possuiam aplicações com padrões bem conhecidos.

Em 1984 foram fundadas duas companhias que iriam produzir os processadores VLIW, a Multiflow, de Fisher e associados e a Cydrome, de Bob Rau, outro pioneiro da tecnologia. Em 1987 a Cydrome lançou seu primeiro produto, o Cidra 5, processador de 256 bits, que possuia suporte a *software pipelining*. A Multiflow lançou no mesmo ano o Trace-200, seguido pelo Trace-300 no ano seguinte e o Trace-500 em 1990. Infelizmente ambas empresas foram comercialmente infelizes. Apesar disso a contribuição tecnológica de ambas ainda está presente hoje. O Intel[2] Itanium[3], moderno processador de 64 bits usa muitos conceitos do Cidra 5. Outros notáveis processadores do presente que implementam essas tecnologias são Transmeta[4] Crusoe[5], Philips[6] TriMedia[7] e Texas Instruments[8] TMS320C62x[9].

3 Princípio Geral de Funcionamento

Na arquitetura do VLIW, o compilador é o responsável em passar para o hardware a informação sobre as instruções que podem executar em paralelo. Na verdade, o processador precisa de 3 informações, e passar essas 3 informações são as funções que devem ser cumpridas:

- As dependências entre as operações devem ser determinadas.
- As operações que são independentes de outras operações que ainda não foram executadas devem ser determinadas.

[1] CPU: *Central Processing Unit* ou Unidade Central de Processamento

[2] `http://www.intel.com`

[3] `http://www.itel.com/itanium/`

[4] `http://www.transmeta.com`

[5] `http://www.transmeta.com/crusoe/`

[6] `http://www.philips.com`

[7] `http://www.semiconductors.philips.com/products/nexperia/media_processing/`

[8] `http://www.texasinstruments.com`

[9] `http://dspvillage.ti.com/docs/catalog/generation/overview.jhtml?templateId=5154&path=templatedata/cm/dspovw/data/c62_ovw&familyId=326`

- Essas operações independentes devem ser agendadas para execução em algum momento em particular, em alguma unidade funcional em específico e deve ser definido um registrador no qual o resultado será colocado.

No caso dos processadores VLIW, existem dois conceitos importantes que devem ser entendidos bem, que são o de operação e instrução. Uma operação é uma unidade de processamento, como uma soma, um *load* ou um *store*. Uma instrução é o conjunto dessas operações que devem ser executadas simultaneamente. O compilador deve escolher quais são as operações que formam cada instrução. Esse processo é o escalonamento das operações. A ordem das operações dentro de cada instrução vai definir cada unidade funcional na qual cada operação vai executar.

O paralelismo no VLIW então advém de dois fatos: cada instrução que é passada para o processador e que ele executa, na verdade são formadas de um número definido máximo de operações, que para os processadores sequenciais são idênticas às instruções que esses executam. Além das instruções multioperacionais (MultiOps) os processadores VLIW ainda implementam *pipelining*, geralmente em 4 estágios (busca, decodificação, execução e escrita).

O escalonamento das operações é feito em um processo chamado de especulativo. O compilador determina as operações de cada instrução muitas vezes sem saber se elas realmente deveriam ser executadas,como no caso de um desvio. O processador não tem como identificar se a operação deveria ser executada, mas ele provê meios para que os efeitos da mesma sejam desfeitos caso o programa chegue em um estado que indique que a operação deva ser anulada. Basicamente são necessários alguns registradores extras, ler algumas instruções a mais e reprimir a geração de algumas exceções de erro que porventura tenham ocorrido.

Independente do tipo de arquitetura envolvida, algumas operações em comum devem ser feitas, seja em tempo de execução (superescalar) ou em tempo de compilação (VLIW). O programa deve ser analisado para que as dependências sejam identificadas, e o ponto no tempo em que as operações são independentes das outras que nesse momento ainda não foram executadas. Devem ser feitos o escalonamento das operações e a alocação de registradores. Para que seja feito o escalonamento de maneira especulativa é necessário que se faça uma previsão de desvio, que vai indicar qual o possível caminho que será percorrido entre as opções existentes.

Aumentar o grau de paralelismo de um processador envolve muitas variáveis, e geralmente é complicado achar um equilíbrio entre elas de modo a gerar uma performance melhor. Um meio de baixo custo é acrescentando mais portas para *pipelining* nas unidades funcionais. Além de possuir um baixo custo essa solução pode duplicar ou triplicar o grau de paralelismo possível, mas em contrapartida existem pontos negativos sérios a serem considerados. Em geral o *clock* máximo sofrerá perdas e o gerenciamento da memória se torna mais complicado. Outro fator é que aumentar o grau de *pipelining* acrescenta atrasos(*delays*) na execução de operações individuais. Outras abordagens para o aumento de performance e paralelismo existem e junto com elas existem prós e contras que devem ser considerados.

`# Fim: Parte do André`

`# Começo: Parte do Charlão`

4 Mecanismos de Previsão de Desvios

5 Emissão de instruções

5.1 Organização da arquitetura VLIW

A arquitetura compartilha um grande banco de registradores entre várias unidades funcionais, responsáveis pelo processamento das instruções.

As instruções, localizadas em longas palavras de centenas de bits na memória de instruções, contém diversas operações que serão executadas paralelamente em uma unidade funcional específica, por este motivo a arquitetura VLIW recebe a denominação de SIMOMD (*Single Instruction, Multiple Operation, Multiple Data*) ou simplificadamente MultiOp.

Figura 1: Processador VLIW.

Figura 2: representaçãio esquemática de um *pipeline*.

Esta arquitetura apesar de apresentar um hardware relativamente simples, exige um compilador extremamente eficiente para escalonar as operações nas unidades funcionais corretas, mantendo a semântica correta, e evitando conflitos estruturais, conflitos de dados ou de controle.

5.2 Múltiplas Instruções

Esta parte é redundante:

A arquitetura VLIW utiliza palavras de centenas de bits, nestas palavras são armazenadas instruções com diversas operações que serão executadas paralelamente nas unidades funcionais. Deste modo, a arquitetura VLIW utiliza a execução múltipla de operações.

CONFUSO PRA KARALHO:

Além de executar diversas operações através do uso de grandes palavras de instruções, a arquitetura VLIW também possui emissão múltipla de instruções através do uso de *pipeline*. Deste modo, o paralelismo da arquitetura VLIW se dá não somente através do uso de *pipeline*, mas o paralelismo se dá através do trabalho simultâneo de várias unidades funcionais e do *pipeline* com a emissão de múltiplas instruções.

Utilizando um *pipeline* de 4 estágios (busca, decodificação, execução e escrita) temos que a utilização das unidades funcionais se dá apenas no estágio de execução, como mostra a Figura 5.2.

Estágios do *pipeline*:

1. **Busca:** A busca de instruções na arquitetura VLIW é simples, uma palavra de centenas de bits é lida como se fosse uma palavra de

Como assim?

32 bits.

2. **Decodificação:** A decodificação das operações também é simples pois as operações dentro da instrução possuem posições pré-definidas. Sendo assim, lê se uma operação, identifica-se seu opcode e envia-o a unidade funcional específica.

3. **Execução:** Neste estágio, temos a concretização das diversas operações contidas em cada instrução. Nesta etapa ocorre o grande diferencial da arquitetura VLIW, pois ao invés de ter apenas uma operação sendo executada no estágio, teremos várias, dependendo do número de unidades funcionais disponíveis e da compatibilidade das operações.

4. **Escrita:** A escrita na arquitetura também é simples, os resultados vindos do estágio de execução são encaminhados à memória ou ao banco de registradores dependendo da operação realizada.

O sucesso da arquitetura VLIW está atrelado a um bom compilador que consiga gerenciar todo processo estaticamente, ou seja, durante o processo de compilação. Isto porque na arquitetura VLIW é função do compilador organizar todo o processo de execução de instruções desde a busca por instruções, passando pela decodificação,*pipeline*, funcionamento das diversas unidades funcionais e escrita de dados.

Esta responsabilidade sobre o compilador, é a causa da simplicidade do hardware de uma arquitetura VLIW, representada na Figura 5.1.

6 Recursos nos Compiladores para expor e explorar ILP

Para um bom desempenho utilizando paralelismo em nível de instrução é necessário, ou desejável, que as unidades funcionais permaneçam ocupadas a maior parte do tempo.

`# O que você quis dizer:`

Para isso, várias técnicas têm sido desenvolvidas de técnicas tanto em *hardware* (*bypassing*) como em *software* (Escalonamento estático). A arquitetura VLIW utiliza escalonamento estático realizado pelo compilador, ou seja, utiliza técnicas de *software* para melhorar o seu desempenho . A seguir apresentamos duas técnicas utilizadas pela arquitetura VLIW para melhor utilizar seus recursos de *hardware*.

6.1 Trace Scheduling

Ao se utilizar paralelismo a nível de instrução é necessário preocupar-se com dependências de dados e de controle afim de que os resultados apresentem os resultados desejados. As dependências dentro de um programa é o principal entrave para o paralelismo a nível de instruição. Pensando nisso, Joseph Fisher desenvolveu um método chamado *Trace Scheduling* que tenta solucionar o problema das dependências e viabilizar o paralelismo.

Segundo Fisher, um programa original pode ser organizado em trechos de instruções independentes que serão escalonadas juntas. Para determinar como serão criados e escalonados os trechos, devemos primeiro realizar um primeiro teste com o código não otimizado, afim de colher estatísticas sobre a execução. Com isso calculamos a propabilidade de cada *branch* condicional.

Das informações colhidas com o bloco inicial, criamos um grafo de precedência de dados (DAG ou *Directed Acyclic Graph*), selecionamos dele uma seqüência linear de trechos de dados (*traces*) livres de laços (*loopings*).

`# Melhor elaborar mais essa parte.  Não entendi nada`

Com o *trace* pronto, consideramos-no como um bloco básico e construímos outro DAG, considerando operações de *branch* como operações comuns. Se ao utilizar um salto condicional algum valor fora do *trace* puder ser perdido, então adiciona-se uma aresta que torna este valor dependente do *branch*. Caso necessário adicione arestas que preservem a ordem do *branch*.

Um novo DAG é obtido, devemos então considerá-lo um bloco básico e escaloná-lo (compactar), selecionando registradores alocados e unidades funcionais conforme são escalonados.

Ao se considerar o DAG como um bloco básico, comete-se erros e para isso é necessário a geração de código compensatório, visando eliminar os efeitos dos erros. Exemplos de compensações:

- Se uma operação que geralmente precede o *branch* condicional for movida para depois do *branch*, devemos adicionar uma cópia desta operação precedendo a parte fora do *trace* do salto condicional.
- Se uma operação que está fora do *trace* for movida para dentro do *trace*, então uma cópia desta operação deve ser adicionada ao caminho que está fora do trace.

Após as compensações, devemos ligar o novo *trace* ao antigo DAG.[Mat81]

Em resumo, o *trace scheduling* consiste em tomar o *trace* com maior probabilidade de execução dentre os *traces* não compactados, construir um DAG *"loop free"* e compactá-lo. Durante a compactação, operações podem ser movidas ou inseridas em outros *traces* não compactados, como forma de manter a semântica do programa ou como código de compensação.

`# Isso não tá repetindo o texto anterior:`

A compactação consiste basicamente em tomar o trace e tentar escalonar (agrupando e movendo) suas operações no menor número de palavras longas possível [Ito98].

6.2 Software Pipelining

`# Fim: Parte do Charlão`

Referências

[Ito98] Sérgio Akira Ito. *Arquiteturas VLIW: Uma Alternativa para Exploração de Paralelismo a Nível de Instrução*. 1998.

[Mat81] Binu K Mathew. *VLIW Processors and Trace Scheduling*. 1981.