Software Pipeline

Sergio Ricardo Souza Leal de Queiroz

IC-Unicamp - RA: 107070

RESUMO

A partir do momento em que se constatou que a evolução dos computadores não estaria mais na velocidade do processador, e sim em projetar máquinas com mais de um processador, o paralelismo se tornou o fundamento principal da construção dos computadores apresentados ao mercado. A partir desse momento, os projetistas de *software* e *hardware* concentraram seus esforços na otimização dos programas em relação a sua execução em paralelo, onde duas ou mais instruções´são executadas ao mesmo tempo, resultando em uma máquina mais rápida. As otimizações dos programas utilizam o conceito de *software pipeline,* que será apresentado a seguir.

1. INTRODUÇÃO

Com o aumento significativo da relação entre a eficiência dos processadores e a eficiência dos tipos de memórias [1], tem sido buscada melhorias no tempo de execução dos programas através do paralelismo, visando apresentar uma máquina mais rápida ao mercado.

O paralelismo consiste em usar mais de um processador na mesma arquitetura, para executar diferentes instruções de um programa no mesmo instante de tempo, ou seja, idealmente distribuir o conjunto de instruções de um programa em processadores diferentes para obter ganho de tempo de execução, implicando diretamente em um melhor desempenho da máquina.
Para isso, o uso da técnica do ILP (paralelismo em nível de instrução, em inglês) ganha importância no caminho da melhoria de desempenho.

O objetivo do ILP é expor o paralelismo possível no programa. Existem duas abordagens para o ILP: a dinâmica - que tem o foco no *hardware,* e a estática - que foca o *software*.
A aplicação do ILP estático é feita através dos compiladores de programas de alto nível.

Na maioria das vezes, o tempo para executar *loops* é o grande responsável pelo tempo de execução dos programas, por isso são o principal foco do ILP.

Software pipeline utiliza a ideia de que o corpo do *loop* do programa pode ser modificado de modo que uma iteração do *loop* possa iniciar antes da iteração anterior ter terminado, obtendo assim um maior paralelismo [2].

Os conceitos de *software pipeline* e *software pipelining* são apresentados no capítulo 2. Também no capítulo 2, o *loop unrolling* é comentado juntamente com suas limitações, dando destaque à variável que quantifica a sua eficiência, denominada Intervalo de Inicialização (II).
Em seguida, no capítulo 3, são mostrados os algoritmos propostos do *software pipelining* de forma resumitiva.
Finalmente, no capítulo 4, as conclusões e considerações pertinentes ao assunto *Software Pipeline* são realizadas.

2. *SOFTWARE PIPELINE*, CONCEITOS

Textualmente descrito em [3], um *pipeline* é como uma linha de montagem. Em uma linha de montagem de automóveis, existem muitas etapas, cada qual contribuindo com algo para a construção do carro. Cada etapa opera em paralelo com as outras etapas, embora para um carro diferente.
Em um *pipeline* de computador cada etapa na pipeline completa uma parte de uma instrução. Assim como a linha de montagem, diferentes etapas estão completando diferentes partes de instruções em paralelo. (...) instruções entram em uma ponta, prosseguem pelo *pipeline* e saem na outra ponta.

Utilizando o conceito de *pipeline* para obter ganhos de eficiência na execução de um programa, a técnica desenvolvida para este propósito é denominada de *software Pipelining*.

2.1 *Software Pipelining*, Conceitos

A técnica de *Software Pipelining*, descrita em [4], consiste em escolher e intercalar instruções de iterações distintas de um *loop*. O escalonador deve de alguma maneira desdobrar o *loop* até encontrar uma linha reta de execução, e então escalona as instruções nessa linha reta sobre as iterações do laço afim de construir um novo laço que inicie sua execução usando o máximo de unidades funcionais possíveis.

A técnica realiza uma reorganização de instruções, reestruturando *loops* de modo que várias iterações sejam sobrepostas (executadas ao mesmo tempo em processadores paralelos) sem modificar a semântica do programa de alto nível. *Software Pipeline* comporta-se de forma similar a um *Loop Unrolling* ilimitado, isto é, infinito [5].

O *Loop Unrolling* realiza o desdobramento do *loop* para que as instruções presentes no mesmo em diferentes iterações sejam executadas de forma sequencial.

2.2 Desdobramento dos loops (*Loop Unrolling*), Conflitos e Restrições

No desdobramento de um *loop*, certamente existem operações conflitantes e restrições, e por isso não poderão ser executadas simultâneamente.

A restrição principal é não modificar a semântica do programa de alto nível.

Um conflito existente é o de dependência de dados.
As dependências são classificadas em três tipos: dependência de dados verdadeira, anti-dependência e dependência de saída.
A dependência de dados verdadeira ocorre quando a instrução dependente precisa ler um registrador a ser escrito por uma instrução anterior.
A anti-dependência ocorre quando uma instrução precisa escrever, mas deve esperar uma anterior ler o registrador em questão
A dependência de saída ocorre quando as instruções escrevem no mesmo registrador.

Uma outra restrição é o uso de mais registradores. O desdobramento do *loop* precisa de registradores diferentes para evitar conflitos gerados pelo uso do mesmo registrador em computações diferentes.

O tamanho do código também restringe o desdobramento, pois pode não caber na janela do hardware, gerando um gasto de tempo no carregamento das instruções do programa, denominado de *overhead.*

O tempo constante para iniciar cada nova iteração, respeitando as dependências e restrições, é denominado Intervalo de Inicialização (II) [6].
Quanto maior for o valor de II, maior o tempo gasto para todas as iterações do *loop*.

2.3 Intervalo de Inicialização (II)

O valor de II é limitado pelas restrições de recursos, isto é, pelos recursos disponíveis de acordo com a necessidade de cada instrução, como exemplo um conflito entre as unidades funcionais e de dados.

O pior valor de II corresponde ao valor da maior restrição, sendo ela de recursos ou de dados.
No trabalho de [5], são citados métodos para calcular o melhor valor de II.

A minimização do valor de II, ou o melhor valor de II, tem relação direta com o número de registradores necessários para o escalonamento, conforme pesquisas de [7].

O II é o elemento principal para a eficiência do *software pipeline*.

3. ALGORITMOS PARA *SOFTWARE PIPELINING*

Aproveitando os recursos de paralelismo, *Software Pipelining* é feito no corpo do *loop*, denominado *Kernel. A*s instruções de *loop* que iniciam o processo de paralelismo antes da formação do *Kernel* são denominadas *Prólogo*, e as que sucedem o *Kernel* são chamadas de *Epílogo*.
Estes algoritmos podem ser abordados em dois grupos: o escalonamento de módulo e a identificação do *Kernel*.

3.1 Escalonamento de Módulo

A partir da identificação do *Kernel*, são realizadas adequações (escalonamento) para melhorá-lo, gerando um novo *Kernel*. Nesse novo *Kernel*, deve-se garantir que não haja violações das restrições de recursos ou da dependência de dados no processo de sobreposição de iterações.
O início de cada iteração de sobreposição deve seguir o valor do intervalo de inicialização (II).
Segundo [5], a dificuldade desse algoritmo é garantir que as instruções sejam posicionadas de tal forma, que todas as iterações sejam identicamente escalonadas. A vantagem em relação a outros algoritmos é de não precisar que o *loop* seja desenrolado, para que se inicie o escalonamento. Assim a principal questão do algoritmo está em descobrir o valor de II para iniciar o escalonamento.
O Escalonamento de Módulo possui algumas variações, descritas a seguir.

3.1.1 O Escalonamento de Módulo Via Redução Hierárquica

O Escalonamento de Módulo Via Redução Hierárquica, que consiste em escalonar separadamente os componentes fortemente conectados [5].

3.1.2 O Escalonamento Caminho Algébrico

O Escalonamento Caminho Algébrico aborda o problema sob o ponto de vista matemático, gerando uma solução elegante, mas apenas teórica [5].

3.1.3 O Escalonamento de Módulo Predicado

O Escalonamento de Módulo Predicado, diferentemente do algoritmo de escalonamento hierárquico, não precisa de reduzir os componentes fortemente conectados. O escalonamento do *loop* gera caminhos diferentes apenas para as instruções dependentes do desvio condicional. Esse método evita a expansão desnecessária de código através da expansão de variável para renomeamento de registradores. Isso reduz a quantidade de recursos necessários para o reuso do código, mas precisa de suporte de *hardware* [5].

3.1.4 *Enhanced* Módulo

Este método utiliza *if-conversion* para posicionar linearmente as instruções do código predicado. Após o escalonamento, renomeia os registradores para as diferentes iterações, e depois restaura-se a estrutura do desvio, *reverse if-conversion*, inserindo instruções de desvio condicional no código. A simplicidade do método pode ser uma vantagem, mas eleva a complexidade do algoritmo, explodindo o código.

3.1.5 *Single-dimention software pipeline* (SSP)

Este algoritmo foi proposto em 2004 por [6], que transforma um loop multi-dimensional em um código unidimensional. É considerado uma extensão do método de escalonamento de módulo tradicional.

3.2 Identificação do *Kernel*

Os algoritmos com essa classificação identificam o *Kernel* através do desdobramento do *loop*, não precisando repetir o escalonamento para diferentes intervalos de inicialização (II) como no escalonamento de módulo. A desvantagem é a elevação do custo do algoritmo. Basicamente, é feita em três etapas, a seguir.

Primeiro faz o desenrolamento do *loop* e verificação das dependências. Depois, realiza o escalonamento das instruções de acordo com o permitido pela dependência de dados. Por último faz a identificação de blocos idênticos, definindo o novo corpo do *loop*, que em seguida forma um novo *loop*.

3.2.1 Perfect Pipeline

Utilizado com arquiteturas com modelos mais geral, resolvendo o problema de mudança de arquitetura através da mudança de parâmetros do problema. Combina a movimentação do código com escalonamento [1].

3.2.2 Modelo de Redes de Petri

O algoritmo de redes de Petri é fundamentado na teoria de grafos para identificar o Kernel, utilizando a renomeação dos registradores.

3.2.3 Técnica de Vegdahl's

É um modelo de exaustão, explorando todas as soluções possíveis, e seleciona a melhor. É impraticável em códigos reais.

3.3 Explorando o Paralelismo em *Multithread*

Multthread é o modelo de programação concorrente mais utilizado. Neste modelo , *threads* concorrentes comunicam-se através da escrita e leitura em variáveis concorrentes. Este modelo responde a questão de como realizar a comunicação entre as diversas tarefas permitindo o compartilhamento de recursos entre elas.
Segundo [8], *threads* podem progredir em paralelo de duas maneiras distintas: (1) sendo realmente executada em paralelo, cada um de posse de um processador; (2) um único processador se revezando entre as *threads*, de forma que cada *thread* possa fazer uso de um procedimento em um período de tempo.

4. CONCLUSÃO

A paralelização é o novo paradigma da arquitetura de computadores. A técnica de *software pipelining* é um desafio para programadores e arquitetos de computadores. Com ela, o espaço entre a velocidade do processador e o acesso a memória pode diminuir significamente, irrelevando a importância da lei de *Moore*.

Para os programadores de alto nível, essa questão se torna transparente, pois entre eles e o código binário existe os compiladores, produzidos por programadores de baixo nível. Os programadores de baixo nível devem trabalhar colaborativamente com os arquitetos para evoluir a questão do paralelismo.

Certamente há muitos desafios para serem descobertos, especialmente para os compiladores. Muitos *loops* exigem uma transformação significativa antes que possam ter *pipelining* de

softwares, as escolhas em termos de *overhead* contra a eficiência do *loop* com *pipelining* de *software* são complexas, a questão do gerenciamento de registradores e ninhos de *loop* criam complexidades adicionais, determinando o tamanho do desafio.

5. REFERÊNCIAS

[1] G. MOORE, "Cramming More Components onto Integrated Circuits", Electronics,
p114-117, April 1965. (Re-impresso em: Proc. of the IEEE, vol 86, no. 1)

[2] JONES, R. B. AND ALLAN, V. H. 1990. Software plpelining: A comparison and improvement. In Proceedings of the 23rd Znternatzonal Symposwm and Workshop on Mlcroprogrammmg and Mzcroarchztecture (MICRO-23) (Orlando, FL,Nov. 27-29). IEEE Computer Society Press, 46-56. KUCK, D. J.

[3] Hennessy, JL and Patterson, DA, (2008), "Arquitetura de computadores: uma abordagem quantitativa", Rio de Janeiro, Elsevier

[4] M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proc. of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, 1988.

[5] Allan, V.H., Jones, R.B., Lee, R.M., and Allan, S.J. Software pipelining. ACM Computing Surveys, vol. 27,No. 3, pp. 367-432, September 1995.

[6] RAU, B. R., LEE. M , TIRUMIiLM, P P , AND SCHLANSKER, M S 1992 Register allocation for modukz scheduled loops Strate@es, algorithms, and heuristics. In Proceedings of the ACM SIGPLAN '92 Conference on Programmrng Language Design and Implementchon (San Francisco, CA, June), ACM, New York, 283-299

[7] Touati S. 2007. *On the Periodic Registrer Need in Sofware Pipelining IEE Transactions on Computer*, 1493-1504

[8] Silva, Dilma M. (2009). "Introdução à Programação Concorrente para a Internet". http://www.inf.puc-rio.br/~francis/ Intro_Prog_Concor.pdf, Acessado em 16 de Junho de 2010.