Software Pipelining

Carla Geovana do Nascimento Macário
RA: 895063

RESUMO

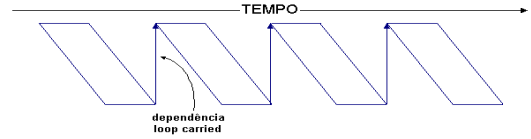
Software pipelining constitui uma das principais técnicas utilizadas pelos compiladores para aumentar o paralelismo na excução de programas. Isto deve-se ao fato de que, em geral, o tempo de execução de loops domina o tempo de execução de um programa. Portanto, atenção especial é direcionada para melhorar o seu desempenho. Este trabalho apresenta um visão geral da técnica de software pipelining, os principais conceitos e desafios envolvidos, trazendotambém uma breve descrição dos principais algoritmos existentes.

Palavras-Chave
paralelismo, software pipelining, modulo scheduling, identificação do kernel.

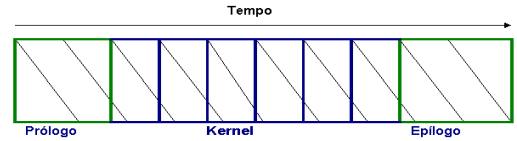
1.INTRODUÇÃO

Cada vez mais busca-se por melhorar o desempenho na execução dos programas e a exploração estática de paralelismo em nível de instrução (ILP) é uma das chaves para isso. Neste caso, o compilador, mediante uma análise prévia, promove um conjunto de transformações capazes de expor mais paralelismo ao código em questão, de maneira a usar diferentes unidades de processamento disponíveis.

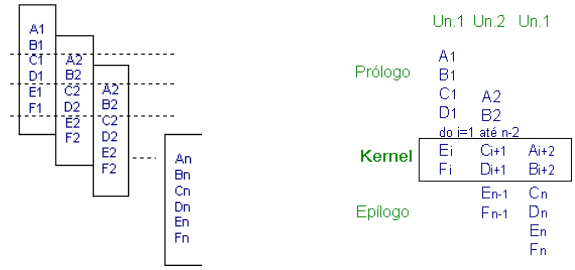
Diversas são as transformações promovidas e dentre elas merecem destaque as que buscam paralelizar os loops, pois em geral o tempo de execução de loops domina o tempo de execução de um programa. Técnicas para isso são o *Loop Unrolling* e o *Software Pipeline*. Na primeira é feito um desenrolamento (desdobramento) do código que compõe o loop, repetindo-o, com os devidos ajustes, tantas vezes quantas forem suas iterações, tornando-o um programa seqüencial. Como limitações tem-se o aumento do tamanho do código e possibilidade de falha em registradores, já que é alta a probabilidade de não ser possível alocar registradores para todas as iteraçõres desdobradas. A segunda técnica, o software pipelining reorganiza as operações do loop de forma a permitir que um conjunto destas operações seja executado em paralelo, promovendo um desenrolamento simbólico e infinito. Com isso, cria-se um novo corpo para o loop, denominado k*ernel* ou *pattern,* que é a parte a ser paralelizada. Considera, então, que embora as instruções ABC que compõem uma iteração do loop possam ser paralelizadas entre si, mais paralelismo pode ser obtido se todas as iterações do loop também puderem ser paralelizadas, numa execução sobreposta, com uma nova iteração tendo sua execução iniciada em intervalos regulares. Para isso leva-se em conta que um loop $\{ABC\}^n$, onde n é o número de iterações a serem executadas, pode ser escrito na forma $A\{BCA\}^{n-1}BC$. Embora as operações não mudem, e nem o número de vezes que serão executadas, o corpo do loop passa a conter operações de provenientes de diferentes iterações. O novo loop passa a ser composto de um prólogo, operações iniciais para preencher o pipeline , kernel e epílogo, instruções para esvaziar o pipeline, respectivamente A, BCA e BC no exemplo dado. A figura 1 ilustra o efeito de uso da técnica software pipelining.



a) loop sem software pipeling



b) loop com software pipeling



c) escalonamento do software pipeling

Figura 1 – Software Pipeling

Este trabalho contém uma breve descrição apresenta uma visão geral desta técnica, seus desafios e os principais algoritmos desenvolvidos para sua implementação. A maior parte do texto foi baseada no survey de Allan [Allan et al 1995] que contém uma descrição detalhada da técnica, conceitos envolvidos e algoritmos existentes

2.SOFTWARE PIPELING: VISÃO GERAL

Software pipelining é um excelente método para promover paralelismo em loops, tendo sido motivado pelo trabalho de Patel para escalonamento de hardware pipelining [Patel & Davidson 1976]. Sua viabilidade deu-se com a tecnologia de processadores paralelos, sendo desenvolvido por Rau e Aiken o primeiro compilador que realmente tirava proveito das características da arquitetura paralela para implementar paralelismo em loops [Lam 2004].

A idéia básica da técnica de software pipeling é promover um novo escalonamento do loop, ou seja reformá-lo, de tal forma que cada iteração possa ser iniciada antes que outra termine, potencializando, assim, o paralelismo. Percebe-se então que a identificação deste novo loop, ou *kernel*, é chave para o sucesso da técnica. Vários itens devem ser levados em conta neste processo e um deles é conhecer as dependências entre operações, já que operações dependentes não podem ser executadas em paralelo.
Um grafo de dependência de dados (DDG) é a ferramenta comumente usada para representar as dependências existentes entre operações de uma iteração, onde os nós representam as operações e os arcos que representam uma ligação do tipo "deve seguir". Um arco *loop carried* indica dependências "deve seguir" entre iterações diferentes e um *loop independent* dependência numa mesma iteração. No escalonamento das operações, as que são dependentes não podem ser executadas em paralelo (seja num mesmo processador ou em vários deles). Associado a cada nó há duas informações (diff ,min). *Min* indica o atraso que deve haver para a próxima instrução ser iniciada, sendo usado para especificar operações multi-ciclos (p.ex. de ponto flutuante). *Diff* indica o número de iterações relativas a uma dependência entre iterações. A figura 2 ilustra o código de um loop, o DDG correspondente que contém um loop carried e três independent e o escalonamento correspondente. Note que no loop carried corresponde a um ciclo e seu diff tem valor 1, pois a operação 1 (a[i+1]=a[i]+2) tem dependência com a próxima.

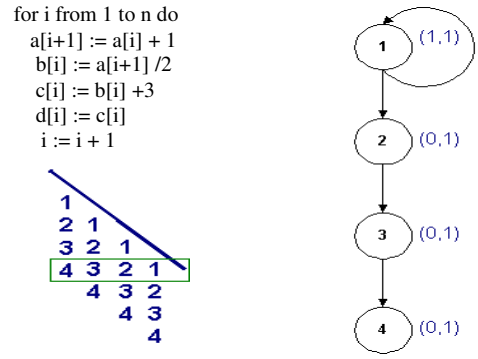


Figura 2 – Grafo de Dependências de Dados e escalonamento baseado nas dependências

Na geração do escalonamento, estas informações são levadas em conta e as dependências *loop carried* são preservadas. Portanto, na definição do novo escalonamento e do novo loop é importante definir o intervalo de tempo necessário para iniciar cada iteração. Este intervalo denomina-se Intervalo de Iniciação (*II*) e quanto menor ele for, maior *troughput*. O *II* mínimo (*MII*) é o menor *II* possível para um escalonamento válido para o software pipelining. Geralmente não é fácil identificar o *II* apenas analisando-se o DDG. Na verdade, este é um problema NP-Completo. Mas se *MII* é encontrado e ele é igual ao *II* do escalonamento obtido, então este é um escalonamento ótimo do ponto de vista do *throughput* [Rau 1994].

Alguns fatores afetam a determinação do *II*. Um deles é a dependência de dados representada pelo DDG. O *II* deve considerar as dependências existentes respeitando os devidos atrasos. O outro fator é o uso de recursos (unidades funcionais, bus, etc). A utilização de recursos é calculada totalizando os requerimentos impostos por cada operação de uma iteração. A figura 3 ilustra tabelas de reservas utilizadas para a identificação de recursos necessários para operações de adição e de multiplicação. Numa breve análise deste exemplo, verifica-se que estas operações não poderiam ser excutadas em paralelo, pois ambas usam os dois Source Bus disponíveis no primeiro ciclo da execução. No processo de escalonamento é usada uma estrutura de dados para

representar esta informação de maneira a tornar possível o cálculo do *II* acomodando adequadamente o uso dos recursos por ciclo. Portanto, o *MII* deve ser igual ou maior que as restrição de recursos (RecMII) e de dependência (ResMII) existentes. Então, MII = max(ResMII, RecMII).

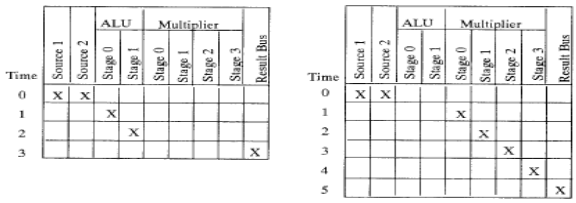


Figura 3 – Tabelas de Reservas de Recursos

Diversos métodos são usados para calcular o *II.* Dente eles temos*:* enumeração simples de ciclos; algoritmo de menor caminho, que usa fecho transitivo; menor caminho iterativo e programaçao linear.

3.CLASSIFICAÇÃO DOS ALGORITMOS

Alguns problemas podem existir no software pipeling, como por exemplo não se conseguir chegar a um *kernel* viável. Portanto a escolha do algoritmo de escalonamento a ser utilizado e a técnica adotada é importante. Os algoritmos de software pipelining geralmente dividem-se em 2 grupos segundo a abordagem utilizada para promover o escalonamento, Escalonamento Módulo e Identificação do Kernel, os quais serão discutidos a seguir.

3.1. Escalonamento Módulo

O Escalonamento Módulo foi proposto por Rau [Rau & Glaeser 1981] e consiste num framework que especifica um conjunto de restrições a serem seguidas para gerar um escalonamento válido. Esta abordagem prevê um escalonamento inicial do *kernel*, que depois, usando técnicas de movimentação, é melhorado. Como vantagem tem-se que não há muita expansão de código, já que não exige desenrolamento de código para efetuar o escalonamento.

No escalonamento obtido, espera-se que na repetição da iteração em intervalos regulares (o *II*), não haja conflito de recursos e que as restrições de dependência existentes sejam respeitadas. Em outras palavras, durante o escalonamento para geração do novo *kernel*, quando uma operação é colocada em uma dada localização, seja garantido que nenhuma dependência de dados ou conflito de recursos será violado durante a sobreposiçao de iterações. A dificuldade da técnica está em garantir que a colocação das operações é legal de tal forma que sucessivas iterações sejam escalonadas de maneira idêntica. Uma vez determinado o escalonamento é efetuada a movimentaçao do código e sua estrutura completa pode, então, ser definida.

Para promover o escalonamento inicial, é necessário o conhecimento do *II*, que é estimado em função das restrições existentes, não sendo necessariamente o melhor, ou seja aquele que promove o melhor *throughput*. O algoritmo é repetido testando valores menores para o *II*, até que uma solução seja encontrada, satisfazendo as restrições existentes.

No caso da existência de caminhos múltiplos decorrentes de desvios condicionais, antes de proceder o escalonamento, o código deve ser transformado em um único caminho usando algum algoritmo específico para isso, pois o seu uso em loops com desvios torna-se muito complicado. Diversas instâncias de algoritmos foram criadas sob este framework, sendo que elas diferenciam-se exatamente no tratamento dos caminhos múltiplos. Os princi e serão aboradados a seguir.

Redução Hierárquica

Este algoritmo proposto por [Lam 1988] é uma extensão do escalonamento módulo, com algumas melhorias, bastante difundido, sendo uma referência para a maioria dos trabalhos desenvolvidos desde então. Sua principal motivação foi tornar o software pipelining aplicável a todos loops, inclusive àqueles contendo instruções condicionais. Esta abordagem escalona o loop hierarquicamente, começando com as construções de loops mais internos. À medida que cada construção é escalonada ela é reduzida a um nó simples do DDG, com este nó representando todas as restrições existentes de seus componentes em relação aos demais nós. Com isso reduz toda a construção condicional para um único caminho que representa a união do uso de recursos. Com isso, o nó pode então

ser escalonado considerando as "novas" restrições existentes. O processo continua até que todo o programa esteja representado por um único nó. A figura 4 ilustra este processo no DDG.

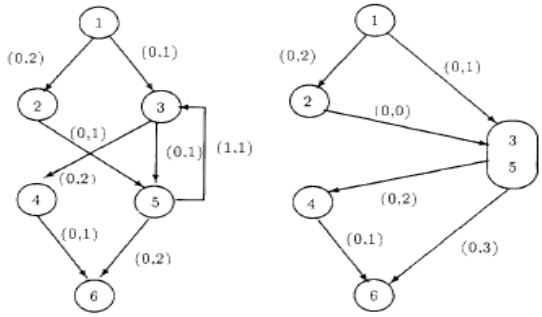


Figura 4 – DDG na Redução Hierárquica

Uma de suas principais contribuições é a expansão de variável, uma otimizaçao feita quando um mesmo registrador é usado em todas as iteraçoes. Neste caso, para permitir o paralelismo, é necessária alguma modificação. Inicialmente o algoritmo identifica os registradores que são usados em todas as iterações, fazendo o escalonamento normal do loop, como se não houvesse a dependência. O escalonamento resultante é usado para identificar onde é necessária a troca de registradores, considerando a sua duração (primeira e última vez que é usado), efetuando-a quando necessária para permitir o paralelismo. Por exigir o desenrolamento inicial para identificação dos conflitos de registradores, esta técnica pode levar a muita expansão de código. A figura 5 exemplifica um trecho de código onde esta operação foi efetuada.

```
Def(R1)          Def(R1)
op               op       Def(R2)
Use(R1)       L:Use(R1)   op       Def(R1)
                          Use(R2)  op       Def(R2) CJump L
                                   Use(R1)  op
                                            Use(R2)
```

a) código da iteração

b) iteração após a expansão de variável

Figura 5 – Expansão de Variável

Escalonamento Predicado

Esta é outra abordagem para promover o tratamento de desvios condicionais em loops. Neste caso, cada operação tem uma entrada predicada de 1-bit. Se o bit é falso, a operação não deve ser executada e o hardware a trata como um no-op. Caso contrario, ela é executada de forma usual. A execução predicada geralmente elimina a necessidade de desvios condicionais e é uma extensão da técnica *if-conversion* que transforma um código não estruturado em código livre de desvios condicionais. Uma alternativa ao uso de bit predicado é o uso de instruções predicadas específicas (invalidade, etc).

A execução predicada usa uma alternativa à expansão de variável, que evita duplicação de código. É usado um arquivo de registros rotativo (*rotating register file*) que funciona como uma lista circular, tornando disponível a cada iteração apenas uma janela de registradores que implementam de maneira lógica os registradores usados na iteração.

Esta técnica é mais flexivel que a anterior, pois na redução hierárquica o código condicional é escalonado antes das demais operações. Neste caso, uma decisão arbitrária mal feita pode impactar as demais que sequer foram ainda consideradas. Na execução predicada todas operações são carregadas e apenas aquelas que devem ser executadas (bit true) são efetivadas e portanto, o escalonamento leva todas em conta. Entretanto, tem como desvantagem a execução de todas as operações

3.2. Identificação do Kernel

Diferente do escalonamento módulo, os algoritmos desta classe promovem o desenrolamento do loop para que consigam identificar um padrão. Assim é possível que em alguns casos o seu custo se torne impraticável, dada a expansão de código existente.

Compreende os seguintes passos: desenrolamento do loop, anotando as dependências existentes; escalonamento das operações respeitando as dependências; tentativa de identificação de um conjunto de operações que se repete para formar o corpo do novo loop; reescrita do loop, considerado o novo kernel. A questão em relação a esta abordagem é: e se não surgir o novo bloco? Normalmente usam-se técnicas de movimentação para isso. A seguir as principais técnicas desenvolvidas seguindo esta abordagem.

Perfect pipelining

Esta é uma das técnicas mais difundidas, e assim com a redução hierárquica, deu origem a vários outros trabalhos. Aiken e Nicolau [Aiken & Nicolau 1988]

introduziram o perfect pipeline como um algoritmo ou framework que combina movimentação de código com escalonamento para melhorar o paralelismo. Usa alguns testes para recolocar o problema segundo novos parâmetros, respondendo à pergunta: "Como o software pipelining seria afetado se a arquitetura fosse modificada para suportá-o"?

Os autores colocaram em seu trabalho que as técnicas disponíveis até então não exploravam toda a capacidade de paralelismo disponível, mesmo quando os recursos existentes eram ótimos. Com isso, propuseram o perfect pipelining e provaram que esta técnica levava a um escalonamento ótimo, ou seja, considerando as depedências de dados existentes, nenhum paralelismo melhor podia ser obtido em melhor tempo, fazendo uso efetivo dos processadores e dos recursos diponíveis. O problema é encontrar uma utilização ótima dos rescursos, mesmo quando todos os recursos necessários nào estão disponíveis.

A abordadem é simples: considera o histórico de execução de um loop e baseado nas i primeiras iterações, escalona estas iterações o mais breve possível, considerando a dependência entre as iterações como o fator para definir o nível de paralelismo entre elas. Apesar do pressuposto de que para revelar um padrão para o kernel deve ser escalonado um conjunto grande de iterações, os autores mostraram que a quantidade de iteraçoes a serem executadas para revelar este corpo é mínimo.

Para a identificaçao do corpo do loop, são escalonadas as operações que não estão no caminho crítico, buscando-se com isso, um *kernel* compacto. Nesta atividade são usadas técnicas de movimentaçao do código. Inicialmente o escalonamento é feito como se não houvesse restrições de recursos, ou seja, eles fossem infinitos. Após a identificaçao do prólogo, epílogo e *kernel*, é feita uma análise da quantidade de processadores disponíveis, veriicando se eles são suficientes para sua paralelização. Caso não sejam, um conjunto de heurísticas simples é usado para reduzir o largura do *kernel*, sem contanto, aumentar o seu tamanho, tornando-o adequado aos recursos disponíveis.

Da mesma forma que no escalonamento módulo, o corpo do loop não pode conter desvios além daqueles de teste de saída. Portanto, técnicas de *if-conversion* devem ser aplicadas antes da utilização do algoritmo.

Modelo de Redes de Petri

Consiste numa variação do algoritmo anterior, usando redes de petri para resolver o problema de reconhecimento do kernel [Rajagopalan & Allan, 1993]. Poderoso como o perfect pipelining, substitui as técnicas *adhoc* utilizadas por uma fundamentação matemática. No algoritmo proposto a rede de Petri é usada para mapear as depedências cíclicas do loop. Apesar do DDG mostrar as dependências "deve seguir" entre operações, ele não é capaz de identificar as operações que estão prontas para serem executadas. Neste sentido a rede de Petri é utilizada, funcionando como um DDG com o estado do escalonamento corrente embutido, o que pode ajudar bastante no processo de identificação do *kernel*. Cada operaçao é representada por uma transição e os arcos representam as dependências entre elas. Apesar das vantagens apresentadas, não é uma técnica muito difundida.

Técnica de Vergdahl

Representa um método exaustivo de identificação do *kernel* no qual todas as soluções possíveis de escalonamento do loop são representadas e dentre elas a melhor é escolhida. O método usa programação dinâmica para chegar a um grafo de decisões a partir do desenrolamento de loop. Neste grafo os nós representam as operações escalonadas e os arcos as possíveis decisões de escalonamento. Como o escalonamento em si já é um problema NP-completo, esta técnica é inviável na prática. Entretanto, é capaz de fornecer heuríticas bastante úteis para os demais algortimos existentes.

Enhanced Pipeline

É uma extensão do Perfect Pipelining, desenvolvido para tirar proveito de arquiteturas que suportam a execução multi-predicada. Para isso, integra transformação de código com escalonamento conseguindo obter bons resultados. O algoritmo usa movimentaçao de código, por meio de renomeação e substituição de foward, mas retém o corpo do loop, deixando de ser necessária a identificação do kernel. Apesar de ser um algoritmo de difícil entendimento, traz benefícios que nenhum outro conseguiu.

4. ESTADO ATUAL

Embora os algoritmos apresentados sejam relativamente antigos, nenhum novo objeto de impacto foi proposto. O que tem sido apresentado ao longo destes anos são trabalhos desenvolvidos buscando melhorar os algoritmos existentes em pontos específicos. Neste sentido, podemos citar os trabalhos desenvolvidos por [Chabin et al. 2005] e [Rong et al. 2005] que tratam da alocação de registros entre iterações, o de [Zhuge et al. 2003] e o de [Kim et al. 2003] que apresentam abordagens para reduzir o tamanho do código. Em especial tem-se o trabalho de retrospectiva de Lam [Lam 2004] que mostra que software pipelining é efetivo em máquinas VLIW sem exigir suporte complicado de hardware .

5. CONCLUSÃO

Quando se pensa em melhorar desempenho de um programa via paralelismo, software pipelining aparece como uma das técnicas mais importantes existentes, dado o domínio do tempo de execução de um loop sobre o tempo de execução do programa. Esta técnica tem como principais objetivos (e desafios) identificar o corpo do loop a ser paralelizado e diminuir ao máximo o intervalo de execução (*II*) entre suas iterações. Esta não é uma atividade simples, consistindo em um problema NP-Completo. Vários algoritmos surgiram para realizar esta atividade, buscando obter este escalonamento que execute em um tempo polinomial. Dentre eles temos o Escalonamento Módulo, Escalonamento por Redução Hierárquica e o Perfect Pipelining, que ao longo dos anos vêm sofrendo algumas modificações e ajustes, mas continuam dominando o cenário .

Software pipeling é uma técnica tão importante que chega a dirigir o projeto de arquiteturas de computadores. Entretanto, Monica Lam mostra em seu trabalho que ele é efetivo, mesmo em máquinas que não foram projetadas especificamente para estes fim [Lam 2004].

Este trabalho apresentou um visão geral da técnica de software pipelining, os principais conceitos e desafios envolvidos. Além disso, trouxe uma breve descrição dos principais algoritmos existentes, provendo ao leitor uma boa idéia da importância desta técnica.

REFERÊNCIAS BIBLIOGRÁFICAS

Aiken A, Nicolau A. Perfect Pipelining: A New Loop Parallelization Techinique, *Proceedings of the 2nd European Symposium in Programming,* p. 221-235, March 21-24, 1988

Allan V.H, Jones R.B, Lee R. M, Allan S. J. Software pipelining *ACM Comput. Surv.* 27 3 1995 367-432.

Chabini N., Aboulhamid E.M., Chabini I., Savaria Y., Scheduling and optimal register placement for synchronous circuits derived using software pipelining techniques, *ACM Transactions on Design Automation of Electronic Systems* (TODAES), v.10 n.2, p.187-204, April 2005

Kim S., Moon S.M., Park J., Ebcioglu K., Unroll-Based Copy Elimination for Enhanced Pipeline Scheduling, *IEEE Transactions on Computers*, v.51 n.9, p.977-994, September 2002

Lam Software pipelining: an effective scheduling technique for VLIW machines *SIGPLAN Not.* V 23 7 1988 p 318—328.

Lam M. Software pipelining: an effective scheduling technique for VLIW machines *SIGPLAN Not*. V 39 4 2004 p 244-256.

Patel J. and Davidson E.S. Improving the throughput of a pipeline by insertion of delays ISCA '76: *Proceedings of the 3rd annual symposium on Computer architecture* 1976 159—164 ACM Press New York, NY, USA

Rajagopalan M. , Allan V. H. Efficient scheduling of fine grain parallelism in loops MICRO 26: *Proceedings of the 26th annual international symposium on Microarchitecture* 1993 p.2--11Austin, Texas, United States.

Rau B. R., Glaeser C. D. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing MICRO 14: *Proceedings of the 14th annual workshop on Microprogramming* 1981 p183-198 Chatham, Massachusetts, United States.

Rau B. R. Iterative modulo scheduling: an algorithm for software pipelining loops MICRO 27: *Proceedings of the 27th annual international symposium on Microarchitecture* 1994 63—74 San Jose, California, United States

Rong H., Douillet A., Gao G. R., Register allocation for software pipelined multi-dimensional loops, *ACM SIGPLAN Notices*, v.40 n.6, June 2005

Zhuge Q., Xiao B., Sha E.H.M., Code Size Reduction Techinique and Implementation for Software-Pipelined DSP Applications, *ACM Transactions on Embedded Coputing Systems*, vol.2, no. 4, November 2003, p. 590-613.