Trace Scheduling

Vitor Monte Afonso
046959
vitor@las.ic.unicamp.br

ABSTRACT
Existem diversas técnicas usadas por compiladores para otimizar código, tornando-o mais rápido ou até diminuindo (compactando) o código, como é o caso da técnica *trace scheduling*. Neste artigo esse método de otimização é descrito, além disso, são apresentados os algoritmos existentes.

Termos Gerais
Otimização global de código, compactação de microcódigo, paralelização de instruções

Palavras-chave
Trace scheduling, compactação global de microcódigo

1. INTRODUÇÃO
Para um bom aproveitamento dos recursos computacionais diversas técnicas de otimização de código são empregadas. Uma forma de fazer isso é com o desenvolvedor indicando que conjuntos de instruções podem ser executados em paralelo, mas esse método é muito difícil e custoso. Por isso é necessário que esses processos de otimização sejam empregados pelo compilador.

Uma forma de alcançar isso é utilizando a compactação de microcódigo. Esta consiste de transformar código seqüencial (vertical), em código horizontal, através da paralelização das microoperações.

Essa compactação pode ocorrer de duas maneiras, localmente ou globalmente. O método local consiste da otimização dentro dos blocos básicos. Blocos básicos são seqüências de instruções que não possuem saltos, excetuando-se a última instrução. Entretanto os blocos básicos não costumam possuir muitas instruções, fazendo com que essa otimização seja bastante limitada.

Isso mostra a importância do método global, no qual vários blocos básicos são levados em consideração de uma única vez, ao invés de trabalhar um a um. Pesquisas mostram que dessa forma é possível obter mais paralelismo [2], [3].

Estudos passados mostram que esse é um problema NP-completo [1], então achar sua solução ótima é computacionalmente inviável. Para isso foi criada a técnica de *trace scheduling* que apesar de não chegar na solução ótima, se aproxima consideravelmente dela.

O objetivo deste trabalho é apresentar essa técnica e mostrar os algoritmos utilizados nela. Ele está organizado da seguinte forma, na seção 2 são apresentadas algumas definições, na seção 3 são apresentados os algoritmos e na seção 4 as conclusões.

2. CONCEITOS NECESSSÁRIOS
Durante a compactação de microcódigo trabalhamos com microoperações (MOPs) e grupos de MOPs. MOPs são a operação mais básica com a qual o computador trabalha.

Uma ou mais MOPs podem formar micriinstruções (MIs), que é a menor unidade com a qual o algoritmo de *trace scheduling* trabalha. A união das Mis forma P, o programa que estiver sendo procesado.

Existe também a função *compacted* : P $\rightarrow$ {*true, false*}. No início da compactação todas as Mis são inicializadas com *false*. Se *compacted*(m) é *false*, m é chamado de MOP.

As funções *readregs* e *writeregs* : P $\rightarrow$ conjunto de registradores, definem respectivamente os registradores lidos e escritos pelas microinstruções.

Há ainda a função *resource compatible* : P $\rightarrow$ {*true,false*}, que define se um determinado conjunto de microinstruções pode ser executado em paralelo, ou seja, o processador possui recursos suficientes para todas.]

Durante o processo de compactação algumas MOPs podem mudar de posição. Para que essa mudança não acarrete em alterações na semântica do programa algumas regras devem ser seguidas.

Definição 1: Dada uma certa seqüência de MIs (m1, m2, m3, ..., mt), é definida a ordem parcial (<<). Se mi << mj mi tem precedência de dados sobre mj. Ou seja, se mi escreve em um registrador e mj lê esse valor, mi << mj e mj não pode ler esse valor enquanto mi não escrevê-lo. Além disso, se mj lê um registrador e mk escreve nele posteriormente, mj << mk e mk não pode escrever no registrador enquanto ele não for lido por mj.

Definição 2: A ordem parcial gera um grafo direcionado acíclico (DAG) que é chamado de grafo de precedência de dados, cujos nós são Mis e existe uma aresta de mi para mj se mi << mj.

Definição 3: Dado um DAG gerado a partir de P, a função sucessores: P $\rightarrow$ conjuntos de P define os sucessores de uma MI da seguinte forma. Se mi,mj pertencem a P e i<j, mj pertence ao sucessores(mi) se existe uma aresta de mi para mj.

Definição 4: Tendo um P com um grafo de precedência de dados, definimos a compactação ou scheduling como um particionamento de P em conjuntos disjuntos S = (S1, S2, ..., Su) seguindo certas propriedades. Para cada k, $1 \leq k \leq u$, *ressource_compatible*(Sk) = true. Além disso, se mi << mj, mi está em Sk e mj está em Sh, com k<h.

3. TRACE SCHEDULING

3.1 Método Menu
Muitos programadores realizam a compactação de microcódigo manualmente, quando acham que é possível. A tabela 1 mostra um menu com regras para movimentação de código que poderia

ser usado implicitamente por um programador que faria essa compactação.

A tabela é feita utilizando conceitos de grafos de fluxo. Definições de grafos de fluxos podem ser vistas em [4] e [5].

Regra	DE	PARA	Condições
1	B2	B1 e B4	MOP livre no início de B2
2	B1 e B4	B2	Cópias da MOP estão livres no fim de B1 e B4
3	B2	B3 e B5	MOP livre no fim de B2
4	B3 e B5	B2	Cópias da MOP estão livres no início de B3 e B5
5	B2	B3 (ou B5)	MOP livre no fim de B2 e os registradores modificados pela MOP estão mortos em B5 (ou B3)
6	B3 (ou B5)	B2	MOP livre no início de B3 (ou B5) e os registradores modificados pela MOP estão mortos em B5 (ou B3)

Tabela 1. Menu com regras para movimentação de código

Esse método foi automatizado por técnicas anteriores de compactação de microcódigo [6], [7]. Isso é feito basicamente da seguinte forma:

1. Apenas código sem laços é tratado;
2. Os blocos básicos são tratados separadamente;
3. É feita ordenação nos blocos básicos;
4. Movimentações de blocos para blocos tratados anteriormente são consideradas e são feitas se forem salvar ciclos.

Esse método automatizado possui alguns problemas.

- Quando uma MOP é movida, podem ser criadas mais possibilidades de movimentação. Isso causa uma grande quantidade de buscas na árvore, deixando o processo bastante pesado;
- Operações em movimentações são pesadas e são repetidas muitas vezes;
- Para localizar movimentações que resultam em grandes melhorias é necessário passar antes por movimentações que não ajudam ou até pioram o código.

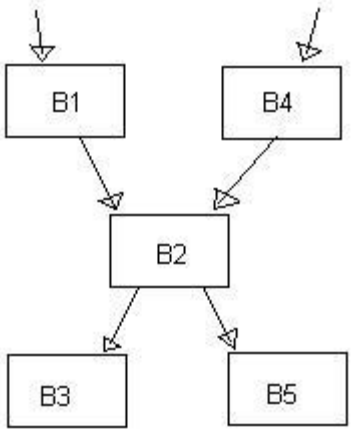

Figura 1. Grafo de fluxo

3.2 Trace Scheduling

Os problemas encontrados pelo método menu são resolvidos na técnica de trace scheduling. Esta, ao invés de operar sobre blocos básicos, opera sobre *traces.* Traces são seqüências de instruções sem ciclos que, para determinado conjunto de dados, são executadas continuamente. Assim, a técnica avalia vários blocos básicos por vez, podendo localizar mais possibilidades de movimentação de código.

A definição que segue será usada pelo algoritmo.

Definição 5: Dada a função *followers*: P $\rightarrow$ subconjuntos de P e uma microinstrução m, o conjunto *followers*(m) é composto pelas microinstruções que podem ser executadas após a execução de m. Se mi está no *followers*(mj), então dizemos que mj é líder de mi. Se o *followers*(m) possui mais de uma micoinstrução, então m é chamada de *jump* condicional.

Um *trace* é portanto uma seqüência de microinstruções distintas (m1, m2, m3, ..., mt) tal que para cada j, $1 \leq j \leq t\text{-}1$, mj+1 está no *followers*(mj). Este *trace* forma um DAG da seguinte forma.

Definição 6: Dado um *trace* T = (m1, m2, m3, ..., mt), usamos a função de sucessores para gerar um DAG chamado grafo de precedência de dados de um trace. Isso é feito de forma análoga aos blocos básicos usando *readregs*(m) e *writeregs*(m), com excessão das arestas de *jumps* condicionais. Se m é um *jump* condicional, os registros em *condreadregs*(m) são tratados como sendo de *readregs*(m).

Então, o *trace scheduling* para códigos sem ciclos, é feito da seguinte maneira. Dado P, selecionamos o caminho cuja execução é mais provável dentre as MOPs que ainda não foram compactadas, a partir de uma aproximação de quantas vezes cada MOP é executado para um conjunto de dados, depois construímos o DAG e fazemos a compactação.

Após esse processo, algumas inconsistências podem ter sido inseridas, por isso é necessário executar o algoritmo de *bookkeep* descrito em [8].

3.3 Código com Loops

Códigos com loops devem ser tratados com cuidado porque em geral estas são as partes que executam com maior freqüência nos programas.

Definição 7: Loops são conjuntos de microinstruções que ligam um bloco a um bloco anterior no grafo de fluxo. Mais detalhes podem ser vistos em [5].

Assumindo que o grafo em questão é redutível e os loops estão ordenados. A figura 2 mostra um exemplo de grafo redutível.

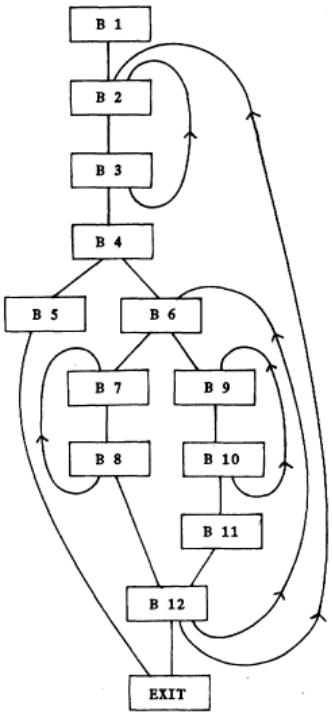

Figura 2. Grafo redutível

Existem duas formas de modificar o algoritmo de *trace scheduling* de forma que ele trate códigos com loops.

No primeiro caso, basta compactar os loops na ordem do mais interno para o mais externo, As arestas de retorno são tratadas como *jumps* para *exit*. O último loop a ser tratado é o P.

O método acima não se aproveita de certos casos em que a compactação pode ser consideravelmente mais efetiva, que são apresentados a seguir.

- Deve-se considerar trocas entre MOPs que estão antes e que estão depois de loops;
- Em certos casos de loops que executam poucas vezes, uma boa estratégia é mover certos MOPs para dentro do loop quando podem ser feitos sem ciclos adicionais e são invariantes de loop.

A técnica mais eficiente trata cada loop compactado como um MOP do loop de nível superior. O escalonador não tem ciência dessa representação, então quando um desses loops aparece no *trace*, as operações que estão antes e depois dele poderão ser alteradas sem problemas.

A definição a seguir é importante para podermos mover MOPs para dentro de loops também.

Definição 8: Dado um loop L, sua representação como MOP lr e um conjunto de operações N, o conjunto [lr] U N é *resource compatible* se N não possui representações de loop, cada operação de N é invariante com relação a R e a adição das operações de N no escalonador de L não o tornam mais pesado.

Nesse caso os loops também começam a ser compactados a partir do mais interno, e os loops vão sendo cada um na sua vez substituídos por MOPs representativos. Transferências de controle de e para o loop são tratadas como *jumps* de e para a instrução que representa o loop.Após isso o algoritmo se comporta como no caso em que não são tratados loops. Quando um representante de loop aparece no *trace*, ele é incluído no DAG.

Assim que o DAG está terminado, o processo de *scheduling* continua até que algum representante de loop esteja pronto. Ele é então considerado para inclusão em cada novo ciclo C. A inclusão é feita se a microinstrução representativa for *resource compatible* com as operações que já estão em C.

Após o fim do escalonamento os representantes são substituíddos pelos loops completos incluindo as possíveis modificações que tenham sido feitas. O algoritmo de *bookkeeping* é então executado normalmente. Esse método possibilita a movimentação de MOPs para antes, depois e até para dentro de loops.

3.4 Melhorias Para o Algoritmo de Trace Scheduling

3.4.1 Reduzindo Espaço

O algoritmo apresentado consome uma grande quantidade de memória devido ao algoritmo de *bookkeeping*. Essa perda de espaço se deve mais especificamente ao espaço necessário para gerar escalonamentos mais curtos e espaço usado porque o escalonador toma decisões arbitrárias, que as vezes acabam consumindo mais espaço em memória do que é necessário.

Medidas para redução do consumo de espaço podem ser tomadas juntamente com a compactação de código, a partir da adição de arestas no DAG para reduzir a duplicação, ao custo de flexibilidade.

Se a probabilidade de um bloco ser executado está abaixo de um certo limite e um escalonamento curto não é crítico, o processo de adição destas arestas segue da seguinte forma:

- No caso de um bloco terminado em *jump* condicional, adicionamos uma aresta de cada MOP que está acima do *jump* e escreve em um registrador que está vivo. Esta aresta irá ajudar a evitar que sejam feitas cópias em blocos que não são muito utilizados;
- Se o início do bloco é um ponto de reunião do *trace*, adicionamos uma aresta para cada MOP livre no início do bloco, de cada MOP que está em um bloco anterior no *trace* e não possui sucessores em blocos anteriores. Isso deixa o ponto de reunião limpo e sem cópias;
- Arestas de todos os MOPs que estão acima de pontos de reunião para a representação do ciclo previnem cópias inapropriadas..

3.4.2 Task Lifting

Antes de compactar um *trace* que sai de um outro *trace* já compactado, pode ser possível mover MOPs que estão no início do *trace* novo para buracos no escalonamento do outro. Essa técnica é explicada com mais detalhes em [9].

4. CONCLUSÕES

No artigo foi apresentada a técnica de *trace scheduling* [8] proposta por Fisher em 1981 que é utilizada para realizar compactação de microcódigo. A técnica possui alguns problemas mas sua utilização é bastante útil. Foram apresentadas também maneiras de aprimorar o algoritmo.

5. REFERÊNCIAS

[1] S. S. Yau, A. C. Schowe and M. Tsuchiya, “On Storage Optimization of Horizontal Microprograms,” Proceedings of 7th Annual Microprogramming Workshop, Sept. 30-Oct. 2, 1974, pp. 98-106

[2] C. C. Foster and E. M. Riseman, "Percolation of Code to Enhance Parallel Dispatching and Execution," IEEE Trans. Comput., vol. C-21, pp.1411-1415, Dec. 1972

[3] E. M. Riseman and C. C. Foster, “The Inhibition of Potential Parallelism by Conditional Jumps”, IEEE Trans. Comput., vol. C-21, pp. 1405-1411, Dec. 1972

[4] A. V. Aho and J. D. Ullman, Principles of Compiler Design. Reading,MA: Addison-Wesley, 1974.

[5] M. S. Hecht, Flow Analysis of Computer Programs. New York: El-sevier, 1977.

[6] S. Dasgupta, "The organization of microprogram stores," ACM Comput. Surveys, vol. 11, pp. 39-65, Mar. 1979.

[7] M. Tokoro, T. Takizuka, E. Tamura, and 1. Yamaura, "Towards an efficient machine-independent language for microprogramming," in Proc. II th Annu. Microprogramming Workshop, SIGMICRO, 1978, pp. 41-50.

[8] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. IEEE Transactions on Computers, C-30(7): 478-490, July 1981

[9] J. A. Fisher, "The optimization of horizontal microcode within and beyond basic blocks: An application of processor scheduling with re-sources," Courant Math. Comput. Lab., New York Univ., U.S. Dep.of Energy Rep. COO-3077-161, Oct. 1979