Trace Scheduling

Márcio Paixão Dantas
Universidade Estadual de Campinas
Instituto de Computação
Campinas, Brasil
ra049174@students.ic.unicamp.br

RESUMO

Trace Scheduling é uma técnica de otimização (ou compactação) global de microcódigo criada no início da década de 80. Este trabalho tem como objetivos: explicar o que é *Trace Scheduling*, mostrar algumas variações presentes na literatura e fazer uma comparação entre elas.

Termos Gerais

Dependência de dados, otimização global de microcódigo, compactação de microcódigo, escalonamento de instruções paralelas, processamento paralelo, conflito de recursos.

Palavras-Chave

Otimização global de microcódigo, *trace scheduling*.

1. INTRODUÇÃO

Técnicas de otimização (ou compactação) de microcódigo são importantes porque permitem o desenvolvimento de compiladores eficientes para linguagens de microprogramação de alto nível. Outro fator que impulsiona o estudo de técnicas deste tipo é que à medida em que os microprogramas aumentam de tamanho, torna-se humanamente impossível identificar todas as oportunidades de otimização no código [4]. O problema consiste basicamente em converter microcódigo sequencial (ou vertical) em eficiente microcódigo paralelo (ou horizontal). Historicamente, há duas abordagens de otimização para este problema:

1. local - compactação de blocos básicos[1], separadamente;
2. global - compactação do microprograma, como um todo.

Ao se estudar o problema de otimização local, descobriu-se que o problema é NP-Completo e, então, várias heurísticas foram propostas para resolvê-lo. Desde a década de 80 considera-se o problema de otimização local bem resolvido. [1] e [2] recomendam alguns *surveys* sobre otimização local.

[1]Sequência de instruções sem desvios, exceto no seu começo e fim.

Inicialmente, as técnicas de compactação global consistiam no uso de otimização local nos blocos básicos, separadamente, e, em sequência, na busca por movimentações de operações entre os blocos. [1] mostrou através da técnica de *Trace Scheduling* que compactar blocos separadamente, sem levar em conta as necessidades e capacidades dos blocos vizinhos, leva a escolhas arbitrárias que prejudicam o processo de otimização.

Este trabalho tem como objetivo apresentar os conceitos básicos de *Trace Scheduling* e algumas importantes variações desta técnica. A Seção 2 explora o algoritmo original de *Trace Scheduling*. Nas Seções 3, 4, 5 são apresentadas, respectivamente, as seguintes técnicas: *Singly Rooted Directed Acyclic Graph* (SRDAG), *Tree Compaction* e *Improved Trace Scheduling Compaction* (ITSC). Depois, na Seção 6, é feita uma comparação dos algoritmos apresentados baseando-se no trabalho de [4]. Finalmente, o trabalho é concluído na Seção 7.

2. TRACE SCHEDULING

Ao invés de trabalhar sobre blocos básicos, *Trace Scheduling* trabalha sobre *traces*. Um *trace* ou caminho de execução é uma sequência de instruções livre de ciclos que pode ser executada contiguamente para alguma escolha de dados (entrada). Dado um *trace*, $T = (m_1, m_2, \cdots, m_t)$, um grafo direcionado acíclico (DAG) é construído a partir da seguinte ordenação parcial em T:

- se m_i escreve em um registrador e m_{i+n}, $n > 1$, lê este valor, então $m_i \ll m_{i+n}$ de forma que m_{i+n} não tentará ler o dado até que esteja lá;
- se m_i lê um registrador e m_k é o próximo a escrever neste registrador, então $m_i \ll m_k$. Assim, m_k não sobrescreverá o registrador até que este seja totalmente lido.

O DAG formado pela ordenação parcial $\ll$ é denominado DAG de precedência de dados. Os nós são as instruções de T e arestas de m_i para m_j são estabelecidas se $m_i \ll m_j$.

Dado T, com um DAG de precedência de dados associado, uma compactação ou escalonamento é definido como qualquer particionamento de T em uma sequência exaustiva de subconjuntos disjuntos, $S = (S_1, S_2, \cdots, S_h)$, que satisfaça as seguintes propriedades:

- em cada elemento de S, não pode haver instruções com conflitos estruturais (*hardware*);
- se $mi \ll m_j$, com m_i em S_k e m_j em S_h, então $k < h$. Isto é, o escalonamento preserva a dependência de dados.

O DAG construído para um *trace* abstrai todas as restrições necessárias sobre a movimentação de instruções entre blocos. Para isto, é preciso fazer algumas modificações no DAG de dependência de dados, tendo como base algumas regras básicas de movimentação de microoperações (MOPS's) entre blocos, explicitadas na Tabela 1.

Regra	**De**	**Para**	**Condições**
R1	B2	B1 e B4	a MOP está livre no topo de B2
R2	B1 e B4	B2	cópias idênticas da MOP estão livres no final de ambos B1 e B4
R3	B2	B3 e B5	a MOP está livre no final de B2
R4	B3 e B5	B2	cópias idênticas da MOP estão livres nos topos de B3 e B5
R5	B2	B3 (ou B5)	a MOP está livre no final de B2 e todos os registradores escritos pela MOP estão mortos em B5 (ou B3)
R6	B3 (ou B5)	B2	a MPO está livre no topo de B3 (ou B5) e todos os registradores escritos pela MOP estão mortos em B5 (ou B3).

Table 1: Regras de movimentação de MOP entre blocos básicos da Figura 1.

Uma MOP é *livre no topo* de seu bloco se ela tem nenhum predecessor no DAG de precedência de dados do bloco e, *livre no fim*, se não possui sucessores.

Um *registrador está* vivo em algum ponto do grafo de fluxo se o valor armazenado nele pode ser referenciado em algum bloco depois daquele ponto, mas antes que ele seja sobrescrito. Se o registrador não está vivo em um ponto, então está *morto*.

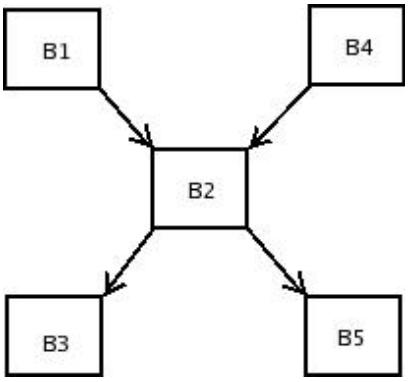

Figure 1: Grafo de fluxo entre blocos básicos.

O escalonador compacta o *trace* produzindo o menor escalonamento possível. O algoritmo usado para fazer isto é o *List Scheduling*, o qual é explicado resumidamente no trabalho de [1].

Uma vez que o escalonador trabalha sobre um *trace* e não sobre o código-fonte do microprograma, é preciso depois do escalonamento de T duplicar algumas MOP's em locais fora do *trace*. Isto mantém o programa correto ao percorrer outros caminhos de execução. Esta etapa chama-se *bookkeeping* e é bastante complexa. Uma explicação conceitual é dada em [1].

Seja, P, uma sequência de instruções (código vertical) que representa um microprograma. De forma resumida, o algoritmo de *trace scheduling* funciona da seguinte forma:

- Para escalonar P, repetidamente é escolhido a partir das MOP's ainda não compactadas o *trace*, T, mais provável de se concretizar. Um DAG com base em T é construído e compactado. O escalonamento é integrado a P fazendo-se as devidas correções (*bookkeeping*). O processo se repete até que todas as MOP's estejam compactadas.

A escolha de qual caminho de execução é o mais provável geralmente baseia-se em simulações ou experimentos reais com o microprograma P.

3. SINGLY ROOTED DIRECTED ACYCLIC GRAPH (SRDAG)

Trace Scheduling compacta caminhos ou *traces* do programa essencialmente como se fosse blocos básicos. As interações entre blocos do caminho são tratadas na construção do DAG de caminhos do programa. Entretanto, interações com blocos fora do caminho não são levadas em consideração. A idéia que suporta esta estratégia é que apenas os caminhos com maior probabilidade de execução devem ser compactados. Porém, segundo [2], compactar blocos de um caminho pode alterar a probabilidade de execução do mesmo. Assim, após compactar um bloco do caminho, deveríamos selecionar novamente o caminho mais provável.

Para aumentar as possibilidades de compactação e escolher corretamente os blocos com maior probabilidade de serem executados foi proposto o *Singly Rooted Directed Acyclic Graph* (SRDAG) [2] ou DAG singularmente enraizado. A idéia básica deste algoritmo é usar um SRDAG para compactar cada bloco do microprograma ao invés de usar e compactar caminhos inteiros. O bloco compactado em cada passo é a raiz do SRDAG. A raiz escolhida em cada iteração é o bloco não-compactado o qual possui a maior probabilidade de execução e, ao mesmo tempo, não possui predecessores não-compactados. Especificada a raiz, o SRDAG selecionado é simplesmente o SRDAG maximal de todos os blocos não-compactados.

A seguir, apresentamos um esquema simples do algoritmo:

- Enquanto houver bloco não-compactado: seleciona SRDAG, compacta raiz de SRDAG e atualiza probabilidades e DAG original.

No SRDAG, é priorizada a movimentação de operações que estão livres no topo de diversos blocos diretamente ligados ao bloco raiz. Assim, diversos caminhos de execução se beneficiam com a compactação. Muitas vezes, a melhor compactação é obtida com a criação de novos blocos.

Outra característica positiva do SRDAG é que a fase de *bookkeeping*, realizado no *Trace Scheduling*, não é necessária nesta abordagem (por construção). Isso facilita bastante a sua implementação.

SRDAG é uma generalização de *Trace Scheduling* que visa ao aumento de desempenho através do uso de um contexto maior de informação. Os resultados obtidos mostram que é possível melhorar o desempenho obtido com a técnica anterior em até 100%. Quando o SRDAG escolhido é um caminho e a compactação de blocos em ordem não afeta as probabilidades de execução, SRDAG se degenera em *Trace Scheduling*. Se um programa compactado com base no *trace* ocupa $O(n)$ de espaço, o mesmo compactado com SRDAG pode chegar a ocupar $O(sqrt(n))$.

4. TREE COMPACTION

Um dos problemas de *Trace Scheduling* é que o crescimento do consumo de memória devido a cópia extensiva de blocos durante o processo de *bookkeeping* pode ser enorme. Em casos específicos pode exponencial [3]. *Tree Compaction* ou Compactação de Árvore foi uma técnica proposta para aliviar este problema por [3]. A idéia básica desta técnica é realizar todas as compactações realizadas por *Trace Scheduling*, exceto àquelas que implicam cópia de blocos. Assim, pretende-se atingir quase o mesmo nível de compactação, porém com uma redução drástica no tamanho de micromemória necessária.

[3] introduz as seguintes definições de árvores:

- *top tree* - subgrafo conexo (do grafo de programa) o qual é em si uma árvore com um único nó com grau de **entrada** zero, a raiz da árvore.
- *bottom tree* - subgrafo conexo (do grafo de programa) o qual é em si uma árvore com um único nó com grau de **saída** zero, a raiz da árvore.

Segundo esta definição, um caminho é simultaneamente uma árvore *top* e *bottom*. O mesmo vale para um nó isolado.

Uma descrição básica do algoritmo:

1. blocos são particionados em subconjuntos de *top trees*;
2. *Trace Scheduling* é aplicado a cada árvore separadamente. Isto é, em cada árvore, um caminho é repetidamente selecionado e compactado usando *List Scheduling* com cópia de MOP's quando necessário;
3. blocos são particionados em *bottom trees*;
4. fazer 2 até que todos os blocos estejam compactados.

No pior caso, *Tree Compaction* apresenta crescimento de memória na ordem de n^2, onde n é o número de desvios condicionais. Lembrando que no pior caso de *Trace Scheduling* o crescimento é exponencial, este é um excelente resultado.

O desempenho em tempo de execução é quase igual (um pouco pior) ao obtido com *Trace Scheduling*. Entretanto, o consumo de memória é drasticamente menor e, se estas duas variáveis tiverem mesma importância, as vantagens de *Tree Compaction* tornam-se bastante evidentes. Outras vantagem é que esta técnica não afeta a estrutura topológica dos blocos. Isto é importante porque facilita a análise de erros (*debug*) no código otimizado. Normalmente, o código produzido por *Trace Scheduling* possui estrutura completamente diferente do programa original, dificultando bastante a análise de erros [3].

5. IMPROVED TRACE SCHEDULING COMPACTION (ITSC)

O ITSC [5] foi desenvolvido para reduzir os problemas de espaço requerido e tempo de execução de *Trace Scheduling* na presença de ciclos nos caminhos. O algoritmo ITSC é baseado em: um conjunto modificado de regras de movimentação de microinstruções entre blocos; um algoritmo de *trace scheduling* melhorado e um algoritmo especial de URCR (*unrolling*, *compacting*, e *rerolling*) para compactação de ciclos. Adotando as novas regras de movimentação é possível aumentar as chances de movimentar MOP's através das fronteiras de blocos básicos, simplificar o procedimento de *bookkeeping* e reduzir o número de cópias desnecessárias.

As regras modificadas de movimentação de MOP's entre blocos básicos são mostradas na Tabela 2.

O algoritmo melhorado de *Trace Scheduling* (ITS) pode ser divido em:

1. partição das MOP's de um caminho em dois conjuntos de acordo com a possibilidade de cada MOP gerar cópias desnecessárias ou não. Parte *MAIN* consiste de MOP's no caminho crítico do DAG e, a outra parte, é formada por desvios que não estão contidos no caminho crítico do DAG;

2. compactação com *List Scheduling* da parte *MAIN*;

3. arranjo de cada MOP remanescente de acordo com seu tipo e características no grafo de fluxo para prevenir movimentos ou cópias desnecessárias.

E o algoritmo URCR para compactação de ciclos tem como etapas: (a) desenrolamento do corpo do ciclo e criação de dois corpos; (b) compactação do ciclo desenrolado com *List Scheduling*; e (c) busca por um novo corpo de ciclo, exclusão de MOP's redundantes e enrolamento do ciclo.

Os resultados obtidos com a compactação de microcódigo usando ITSC mostram que: o tempo de execução do código otimizado é muito parecido com o de *Trace Scheduling*; a melhora percentual no uso de espaço varia de 9% a 24%; e a compactação de ciclos realizada pelo algoritmo URCR consegue fazer com que o código compactado seja comparável ao código compactado manualmente.

6. COMPARAÇÃO ENTRE OS ALGORITMOS

[4] fez experimentos para comparar algoritmos de compactação global de microcódigo. Todos os algoritmos estudados até aqui fazem parte do estudo. Os testes foram realizados em duas arquiteturas diferentes (PUMA e VAX) e os resultados comparados entre si e com as compactações local e manual (tida como ótima).

Os algoritmos apresentados apresentam relações de compromisso diferentes entre vantagens e custos quando comparados com *Trace Scheduling* (TS). Os dados obtidos não permitem conclusões definitivas sobre os métodos, uma vez que só duas arquiteturas foram testadas e os resultados são bastantes próximos.

Em média, *Tree Compaction* sempre apresentou tempo de execução pior e uso de memória melhor que TS. Por sua vez, dentre os algoritmos estudados é o que possui menor complexidade computacional. SRDAG consegue resultados, em média, sempre melhores que TS, tanto para tempo de execução quanto para uso de memória. ITSC apresenta o mesmo rendimento médio. Como desvantagem, ambos os métodos possuem complexidade maior que TS, sendo que o que mais se aproxima de TS é ITSC. Nos testes realizados, no máximo, os algoritmos igualaram o desempenho de otimizações manuais. Comparando os resultados dos algoritmos apresentados com compactação local, fica clara a superioridade de otimização global em relação à primeira.

7. CONCLUSÃO

Este trabalho inicialmente apresentou o método de *Trace Scheduling* para otimização global de microcódigo. Em seguida, variações de otimização global baseadas em TS foram abordadas, entre elas: SRDAG, *Tree Compaction* e ITSC. Uma comparação entre os algoritmos pôde ser estabelecida através do trabalho de [4]. Ficou evidente a superioridade de métodos de otimização global em relação a métodos que realizam otimização local de microprogramas. Uma sugestão dada por [4] na escolha do método apropriado de otimização global para uma arquitetura é comparar os resultados disponíveis de otimização local com os de otimização manual. Quanto maior for a diferença entre os resultados, maior é a chance que métodos globais mais complexos ofereçam resultados melhores que métodos globais mais simples.

8. REFERÊNCIAS

[1] J. A. Fisher. Trace scheduling: a technique for global microcode compaction. *Instruction-level parallel processors*, pages 478–490, 1981.

[2] J. Lah and D. E. Atkins. Tree compaction of microprograms. *SIGMICRO Newsletters*, 14(4):23–33, 1983.

[3] J. Linn. Srdag compaction – a generalization of trace scheduling to increase the use of global context information. In *The Proc. of 16th Annu. Workshop on Microprogramming*, pages 11–22. ACM Press, 1983.

[4] B. Su and S. Ding. Some experiments in global microcode compaction. In *MICRO 18: Proceedings of the 18th annual workshop on Microprogramming*, pages 175–180. ACM Press, 1985.

[5] B. Su, S. Ding, and L. Jin. An improvement of trace scheduling for global microcode compaction. In *Proc. of the 17th annual workshop on Microprogramming*, pages 78–85. ACM Press, 1984.

Regra	De	Para	Condições
R1	B2	B1 e B4	a MOP está livre no topo de B2
R2	B1 e B4	B2	cópias idênticas da MOP estão livres no final de ambos B1 e B4
R3	B2	B3 e B5	a MOP está livre no final de B2
R4	B3 e B5	B2	cópias idênticas da MOP estão livres nos topos de B3 e B5
R5	B2	B3 (ou B5)	para MOP's exceto de desvio: a MOP está livre no final de B2 e todos os registradores escritos pela MOP estão mortos em B5 (ou B3). Condições adicionais para MOP's de desvio: sua saída é a mesma do desvio B no fim de B2 ou Condição(MOP) E Condição(MOP B) = falso
R6	B3 (ou B5)	B2	a MOP está livre no topo de B3 (ou B5) e todos os registradores escritos pela MOP estão mortos em B5 (ou B3)
R7	B2	B1 (ou B4)	a MOP, exceto de desvio, está livre no topo de B2 e todos os registradores escritos pela MOP estão mortos abaixo da MOP B em B1 (ou B4)
R8	B1 (ou B4)	B2	a MOP, exceto de desvio, está livre no fim de B1 (ou B4) e todos os registradores escritos pela MOP estão mortos no topo de B2

Table 2: ITSC: Regras de movimentação de MOP entre blocos básicos referentes à Figura 1.