Trace Scheduling - Um algoritmo para o Problema da Compactação Global de Microcódigo

Renato Teixeira Resende da Silva - RA 035627

ABSTRACT

Chama-se *Compactação de Microcódigo* a transformação de um código escrito de forma seqüêncial, em um microcódigo eficiente que possa ser executado de forma paralela. Existem dois tipos básicos de compactação, a saber, local e global. A compactação de código local tenta atingir seu objetivo dentro apenas dos blocos básicos do programa enquanto que, a global, faz uma análise utilizando-se de um ou mais blocos básicos ao mesmo tempo.

Este artigo discorre em mais detalhes sobre o *Trace Scheduling*, uma técnica desenvolvida por Fisher para o problema da *Compactação de Microcódigo Global*. São também apresentadas alternativas a este algoritmo, bem como correções, aprimoramentos e incrementos que adicionam mais restrições ao algoritmo original como, por exemplo, restrições de tempo. Também são apresentadas as primeiras avaliações feitas utilizando esta técnica quando do seu surgimento e publicação, no ano de 1981.

1. INTRODUÇÃO

Em máquinas multiprocessadas, a forma com a qual o microcódigo é gerado tem fundamental importância na utilização dos recursos providos por tal arquitetura. Isto é feito indicando-se muitas vezes quais microoperações (*MOP's*) podem executar em paralelo. Tal indicação pode ser feita pelo próprio programador, o que na maioria das vezes é uma tarefa árdua e complexa ou, de forma mais plausível, pode ser descoberta e indicada pelo próprio compilador.

Existem duas abordagens básicas para este problema que são a *Compactação Local* e a *Compactação Global*. A análise feita do microcódigo para a compactação local se restringe aos blocos básicos do programa. Um bloco básico é uma seqüência de instruções sem instrução de *jumps* para dentro deste bloco, exceto a primeira, e sem instrução de *jump* para fora do bloco, exceto a última. Esta técnica foi amplamente utilizada, visto que foi comprovado por diversas pesquisas a sua eficiência na geração de código paralelo. Porém, um grande problema aqui é que os blocos básicos tendem a ser extremamente pequenos, limitando muitas vezes a quantidade de instruções paralelizadas. Daí a motivação para a utilização de um método global.

A geração de um código paralelo ótimo a partir de um código seqüêncial, é um problema NP completo [9]. Esta complexidade deve-se ao fato de que, para gerar um código paralelo, diversas microoperações do código original são trocadas de lugar. No entanto essas trocas devem, obviamente, manter a semântica do programa original.

Apesar da solução ótima deste problema ser computacionalmente inviável de ser atingida, soluções que estejam muito próximas da ótima podem ser obtidas através de diversas técnicas que utilizam heurísticas, dentre elas, a técnica aqui descrita com maiores detalhes denominada *Trace Scheduling* [4].

Este documento será apresentado na seguinte forma: na Seção 2, será apresentado o algoritmo de *Trace Scheduling* propriamente dito. Na seção 5, será apresentado brevemente alguns aprimoramentos para o algoritmo proposto por Fisher. Já na seção 6 será apresentado brevemente o algoritmo de *Trace Scheduling* que leve também em consideração restrições de tempo. Já na seção 7.1 será apresentado um outro algoritmo para o problema de Compactação Global de Microcódigo, proposto por Upendra. Na seção 7.2 será apresentado também um novo paradigma para o problema da Compactação Global de Microcódigo denominado *Wavefront Scheduling*. Finalizando, a seção 7.3 apresenta um comparativo para a solução deste problema utilizando heurísticas com algoritmos que encontram soluções ótimas.

2. TRACE SCHEDULING - CONCEITOS BÁSICOS E DEFINIÇÕES

2.1 Definições e conceitos básicos

Primeiramente serão apresentados aqui alguns conceitos básicos e definições que serão utilizados dentro do algoritmo proposto.

A compactação do código seqüêncial para o paralelo modifica a posição original de tipos básicos de dados, a saber, as *microoperações* (MOP's). Estas microoperações representam instruções no qual os processadores são capazes de executar. Uma ou mais MOP's são chamadas de *Microinstruções* (ou MI'S). Por sua vez, esta última é a menor unidade com a qual o algoritmo de *Trace Scheduling* trabalha. Abaixo, seguem algumas definições que serão necessárias para o entendimento do algoritmo:

Definição 1: uma microinstrução (MI) será a unidade básica com a qual o algoritmo irá trabalhar. O conjunto

de todas as microinstruções é denominado P, que é o programa dado propriamente dito.

Existe também a função *compacted*: $P \rightarrow$ {true, false}. No começo da execução do algoritmo todas as MI's têm valor falso. Quando isto acontecer, denominaremos estas MI's como MOP.

Definição 2: Considere um conjunto de registradores do processador em questão. Daí, podemos definir outras duas funções, a saber, *readregs*, *writeregs*: $P \rightarrow$ *subconjunto de registradores*. Para uma determinada MI m, *readreg(m)* e *writereg(m)* são, respectivamente, o conjunto de registradores lidos e escritos por essa MI.

Definição 3: existe a necessidade também de uma outra função, chamada *recurso compatível* (ou *resource compatible*) de $P \rightarrow$ {true, false}. O significado desta função para duas ou mais microinstruções é que, em um mesmo ciclo de processador, elas não irão disputar pelos mesmos recursos, ou seja, poderão ser colocadas para executar em paralelo.

Durante o processo de compactação das MOP's em MI's, a seqüência original de algumas MOP's são alteradas. Entretanto, esta alteração tem que ser feita respeitando-se alguns critérios para que a semântica do programa original não seja alterada. Para que tais alterações ilegais não sejam feitas, o algoritmo deverá respeitar mais algumas definições:

Definição 4: dada uma seqüência de MI's $\{m_1, m_2, \ldots, m_t\}$, podemos definir a ordem parcial ($\ll$) entre elas. Quando $m_i \ll m_j$, dizemos que m_i tem uma precedência de dados sobre m_j. Abaixo segue um exemplo para deixar este conceito mais claro:

- se m_i escreve em um dado registrador e, em seguida, m_j lê o valor escrito por m_i, então $m_i \ll m_j$.
- se m_j lê um determinado registrador e, em seguida, m_k escreve neste mesmo registrador, então m_k não poderá sobre escrever este valor até que ele seja lido. Portanto, neste caso $m_j \ll m_k$.

Definição 5: o operador de ordem parcial $\ll$ define um grafo direcionado acíclico no conjunto de MI's denominado *Grafo de precedência de dados - DAG*. Os nós deste grafo correspondem às MI's. As arestas são desenhadas de m_i para m_j se e só se $m_i \ll m_j$.

Definição 6: dado um DAG de um certo programa P, define-se a função *successors*: $P \rightarrow$ subconjuntos de P. Para m_i e $m_j \in P$, se $i < j$, m_j está em *successors(m_i)* se existe uma aresta no grafo ligando m_i à m_j. Vale notar que estas relações são transitivas, ou seja, se $m_i \ll m_j$ e $m_j \ll m_k$, então $m_i \ll m_k$. Portanto, pode-se utilizar outros algoritmos para a remoção de arestas redundantes.

Definição 7: dado um programa P, um DAG extraído deste mesmo programa, defini-se uma *compactação* (ou *scheduling*) uma partição de P, em uma seqüência de subconjuntos disjuntos de P, $S = (S_1, S_2, \ldots, S_u)$, onde os S_i's são conjuntos de MI's, com as seguintes propriedades:

- Para todo k, $1 \leq k \leq u$, $resource_compatible(S_k)$. Isto significa que, para cada cada conjunto disjunto de microinstruções, não pode haver conflito de recursos entre elas.
- Se $m_i \ll m_j$, m_i está em S_k, e m_j está em S_h, então k < h. Isto é necessário para preservar a precedêcia dos dados utilizados pelas MI's.

3. MÉTODO MENU E TRACE SCHEDULING PARA A COMPACTAÇÃO GLOBAL

3.1 Método Menu

Tendo estas definições feitas, podemos explicar o funcionamento básico do algoritmo. Porém antes, daremos apenas uma breve noção do método utilizado por programadores para fazer esta compactação *on the fly*, para efeito de comparação. Esta forma é utilizada de maneira implícita pela maioria dos programadores que fazem a compactação dos seus códigos manualmente:

Algorithm 1 Método *Menu* para compactação de código global feita manualmente

Dado:
Um conjunto de tarefas com a ordem parcial definida entre elas (um DAG) e um número C de processadores idênticos. Uma função que determina a prioridade de cada tarefa de acordo com algum método.
Devolve:
Uma seqüência de instruções que podem ser executadas em paralelo respeitando-se o DAG. A prioridade de cada tarefa é levada em consideração mas só será respeitada se isto for possível devido aos problemas de precedência.

$CICLO = 0$
$DRS \leftarrow$ Conjunto de tarefas sem precedentes no DAG.
while $DRS \neq \varnothing$ **do**
  $CICLO \leftarrow CICLO + 1$
  Aloque as primeiras C tarefas sem precedentes do DAG aos C processadores de acordo com suas respectivas prioridades se assim for possível
  Remova as tarefas alocadas do DAG
end while

Existem na literatura diversas sugestões para o problema da compactação global de microcódigo utilizando esta heurística. Entretanto, esta possui algumas limitações:

- Somente códigos sem a presença de *loop's* são considerados;
- Cada bloco básico é compactado separadamente, o que acaba por prejudicar a exploração do movimento de MOP's entre os blocos básicos e, conseqüentemente, o nível de paralelização;

3.2 Trace Scheduling

Como dito anteriormente, o problema de compactação de microcódigo em blocos locais é um problema NP completo. Todavia, existem heurísticas diferentes do *Método Menu* que

conseguem soluções próximas da solução ótima. O que o algoritmo de *Trace Scheduling* pretende fazer então, é combinar os vários blocos locais e utilizar um algoritmo semelhante de compactação local nestes blocos. Para tanto, é necessário descobrir qual caminho é percorrido dentro destas microinstruções na maioria das vezes. Isto é, para uma certa combinação de dados, roda-se o programa diversas vezes e descobre-se qual caminho de blocos é mais comumente percorrido. Com isto em mãos, é possível deslocar diversas microperações entre os blocos de modo a obter uma melhor paralelização. Pesquisas [5] indicam que a maior parte do código paralelizável se encontra entre as microoperações de diferentes blocos. Daí o encorajamento em utilizar tal técnica. Porém, estas movimentações têm de respeitar todas as restrições impostas pelas definições citadas em 2.1. A complexidade de tal ação é equivalente a percorrer uma grande árvore de estados. Daí vem a característica de NP completude do problema.
Primeiramente, para que o requisito de precedência de dados seja respeitado, criamos a seguinte definição:

Definição 8: um *Grafo de Fluxos* é um grafo direcionado, tendo como nós os blocos básicos de um dado programa P. Uma aresta é desenhada do bloco B_i ao bloco B_j se, após executar por completo as instruções de B_i poder-se-á começar a executar as instruções de B_j.

A Figura 1 contém um exemplo de uma parte de um programa com seus blocos básicos(2(a)) e o correspondente grafo de fluxos (2(b)). Este grafo pode ser acíclico, quando não houver iterações nos blocos, ou cíclico, caso houver.

O algoritmo de *Trace Scheduling* irá operar então em *rastros* (*Traces*), ao invés de blocos básicos. Um *Trace* é definido como sendo uma seqüência de microcódigo sem *loop's*, que podem ser executados continuamente para uma certa escolha de dados. Apesar desta variante atuar em trechos sem *loop's*, com pequenas modificações é possível tratar este tipo de situação.
Abaixo estão outras definições para se formalizar o funcionamento do algoritmo:

Definição 9: existe uma função *followers:* $P \rightarrow$ subconjuntos de P. Dada uma microinstrução m, dizemos que o conjunto *followers(m)* são todas as outras microinstruções que podem ser executadas após o término da execução de m.

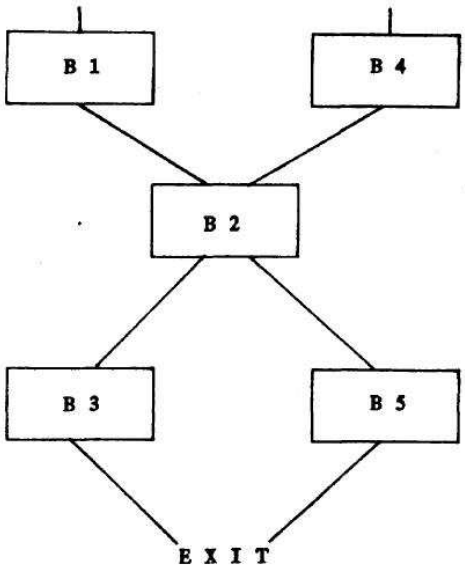
Um *Trace* é, então, uma seqüência qualquer de microinstruções distintas $(m_1, m_2, \ldots, m_t)$, tais que para cada j, $1 \leq j \leq t-1, m_{j+1}$ está em *followers*(m_j). Este *Trace* acaba por definir um *DAG*. Independente do bloco onde uma dada instrução m_i se encontrar, ela só poderá ser executada respeitando-se as restrições de dados impostas pelo programa original.
Com o conceito do *Trace* feito, podemos então formalizar agora o conceito de *DAG* para este caso:

Definição 10: dado um *Trace* $T = (m_1, m_2, \ldots, m_t)$, a função *followers*$(m_i)$ define um grafo direcionado acíclico, denominado *Grafo de Precedência de Dados sobre um*

Figure 1: Exemplo de um programa com seus blocos básicos e microintruções

MOP NUMBER	BLOCK NAME	REGISTERS WRITTEN	REGISTERS READ	FOLLOWERS (DEFAULT FALLS THROUGH TO NEXT MOP ON LIST)
ENTRANCE				m1
ENTRANCE				m15
m1	B1:	R3	R1,R2	
m2		R5	R3,R4	
m3		R6	R3	
m4		R7	R5,R6	
m5		R8	R7,R3	
m6	B2:	R10	R9	
m7		R11	R9	
m8			R10	m9, m17
m9	B3:	R13	R12	
m10		R15	R10	
m11		R16	R13,R14	
m12		R17	R16,R11	
m13		R18	R17,R5	
m14		R19	R18	EXIT
m15	B4:	R5	R3,R4	
m16		R8	R7,R3	m6
m17	B5:	R12	R13	

(a) Exemplo de um programa com seus blocos básicos e microintruções



(b) DAG definido entre os blocos das microinstruções

Trace. Ele é semelhante à um *DAG* criado sobre os blocos básicos de um programa P, e tem as mesmas propriedades.

Para que o algoritmo funcione agora, basta que seja computado o caminho que se percorre mais comumente dentre as MOP's que não estiverem compactadas, criar o *DAG* referente à este caminho e compactá-lo. É interessante notar que pode haver a duplicação de diversas MOP's dentro do *Trace* gerado. Esta duplicação pode ser explicada pela seguinte situação: imagine um programa $P = (B_1, B_2, B_3, B_4, B_5)$, onde os B_i's são os blocos básicos. Suponha que, ao computar o caminho mais comumente percorrido dentro deste código, se encontre, $B_1, B_2, B_3, B_2, B_1, B_5$). Ao gerar o *Trace* deste código, os blocos (ou parte deles) B_1 e B_2 serão duplicados. Para fins de simplificação deste exemplo foi colocado como se todo o bloco fosse percorrido. Porém, na prática isto ocorrerá com cada microinstrução separadamente, independente dos blocos. Esta situação poderá gerar muitas vezes um excesso de códigos duplicados, da onde se retira um dos maiores problemas do algoritmo *Trace Scheduling*.
Com todas as definições feitas, podemos então colocar, em alto nível, o algoritmo *Trace Scheduling* propriamente dito:

Algorithm 2 *Trace Scheduling*

Dado:
P, um conjunto de microinstruções sem *loop*, com as seguintes funções já definidas em P:
followers, *readregs*, *writeregs*, *resource_compatible*.
Devolve:
Um programa P', compactado, com MI's originadas do programa P original, que deverá rodar em um tempo significativamente menor do que o original e que seja semanticamente equivalente.
Usa:
T, variável do tipo *Trace*,
S, variável do tipo *Schedule*,
funções *pick_trace*, *schedule*, *bookkeep** e subrotinas (serão detalhadas posteriormente).

```
for all (m_i ∈ P) and compacted(m_i) = false do
  while ∃m_i ∈ P and compacted(m_i) = false do
    chama pick_trace(T);
    S = schedule(T);
    chama bookkeep(S);
  end while
end for
```

Abaixo se encontram os algoritmos das subrotinas utilizadas pelo algoritmo *Trace Scheduling*:

Algorithm 3 *Pick Trace*

Dado:
Programa P
Devolve:
Um *Trace* dos elementos de P

Acha uma MOP com a maior expectativa de ser utilizada;
Gera um *trace* em voltar desta MOP, analisando quais outras são mais utilizadas antes e após esta ser executada;

No algoritmo 4, existem outras duas novas funções. A função *priority_set(T)* é responsável por definir as prioridades

Algorithm 4 *Schedule*

Dado:
Programa P, passado para o algoritmo de *pickup_trace*
Devolve:
S, que será as instruções compactadas

```
chama build_DAG(T);
chama priority_set(T);
chama list_schedule(T);
```

das MOP'S de T seguindo alguma heurística. Já a função *list_schedule(T)* é uma heurística utilizada para a compactação de microcódigo local, mas que também é aplicada dentro do algoritmo de *Trace Schedule*. A diferença aqui é que, ao invés dela ser aplicada à blocos básicos, ela será aplicada à um *Trace* gerado pelo algoritmo 3.
O último algoritmo, *Bookkeep*, é um tanto mais complexo que os anteriores. De acordo com [4], a sua implementação é demasiadamente complexa e não intuitiva. Como este trabalho segue a idéia do trabalho supracitado, e a fim de que não se saia do escopo deste tema, também optaremos por não publicar os detalhes do funcionamento deste algoritmo. Porém em uma linguagem de mais alto nível, o que o algoritmo *Bookkeep* faz é duplicar as MOP's dentro do código gerado. É passado para ele o programa P e as variáveis *trace* T e S, conseguidas nos algoritmos 3 e 4, respectivamente. A partir daí, com o intuito de manter a semântica do programa original, serão feitas tantas replicações de MOP's quantas forem necessárias para atingir o objetivo proposto.

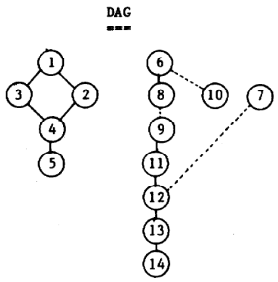
3.3 Exemplo de execução do algoritmo

Considere agora o programa representado pela Figura 1. Suponha que, ao rodar o algoritmo *pick_trace*, o *trace* obtido fossem os blocos B_1, B_1 e B_3. A Figura 3(a) demonstra um *DAG* gerado para este *trace*, bem como o arranjo (*schedule*) das MOP's. Utilizou-se uma heurística onde as MOP's com maior prioridade tentam ser executadas primeiro. Nesta mesma Figura pode ser encontrado a lista de prioridades dadas às MOP's.
O ganho obtido ao se executar desta maneira foi significante: 7 ciclos, contra 11 se fosse utilizado o *Método Menu*, o que vem a mostrar o grande potencial do algoritmo.

4. AVALIAÇÃO DO ALGORITMO DE FISHER

Esta seção vem a mostrar uma breve análise feita por *Grishman* e *Bogong* no algoritmo de *Trace Scheduling*. Em [6], foram implementadas todas as funções mostradas pelos algoritmos, 4, 2, bem como o algoritmo de *bookkeep*. Porém o algoritmo *pick_trace* não foi utilizado. Para isto, os autores preferiram indicar manualmente qual caminho seria o mais percorrido dentro do código proposto.
A implementação foi feita em uma linguagem, até então, de alto nível, denominada *Setl*, rodando em uma máquina *VAX-11/780*. Segundo o próprio autor, a utilização desta linguagem de alto nível facilitou tão somente a implementação dos algoritmos. Entretanto gerou um código ineficiente, pois demorava cerca de 7 minutos para fazer o *schedule* de um segmento de microcódigo com um total de 49 MOP's.
Os trechos de microcódigos foram retirados de microprogramas para um sistema de computação chamado *Puma*. O *Puma* é um emulador para o *Control Data 6600*, com su-

Figure 2: Resultado da compactação e novo grafo de fluxo de dados

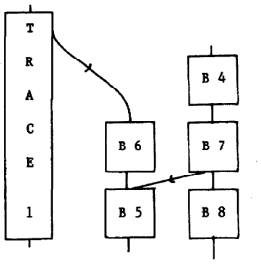


SCHEDULE

CYCLE	MOP(S)
1	m6
2	m8,m10
3	m1,m9
4	m11,m2,m7
5	m3,m12
6	m4,m13
7	m5,m14

PRIORITY LIST: 6, 8, 9, 1, 7, 11, 2, 3, 12, 4, 13, 5, 10, 14

(a) MOP's que teriam que ser copiadas para novos blocos durante a fase de *bookkeeping*



CONTENTS OF NEW BLOCKS

BLOCK	CONTAINS COPIES OF MOPs
B6	1,2,3,4,5,7
B7	6,7,8
B8	9,10,11,12,13,14

(b) Novo grafo de fluxos para o *Trace* gerado

porte a microprogramação horizontal (paralela). Este possui a capacidade de executar até três operações concorrentes: uma operação de *load* ou *store*, uma operação na unidade aritmética e uma operação aritmética exponencial.

De acordo com a Figura 3, os tempos obtidos com o algoritmo *Trace Schedule* foram favoráveis à sua utilização, preferivelmente à utilização de técnicas como Compactação Local ou, até mesmo, a compactação feita pelo próprio programador. Como era de se esperar, o método seqüêncial foi, de todos, o mais ineficiente. A utilização da técnica de Compactação Local se mostrou mais ineficiente do que o próprio programador fazer a compactação. Apesar disso, sua utilização é ainda encorajada pois o trabalho manual passa a se tornar inviável de acordo com o crescimento do tamanho do código dos programas.
Estes resultados comprovam a teoria citada em [5], onde a maior parte da paralelização dentro de um código se encontra entre as microinstruções de diferentes blocos básicos.

5. MELHORIAS NO ALGORITMO DE TRACE SCHEDULING

O algoritmo de *Trace Scheduling* proposto por Fisher mostra-se bastante eficiente comparado com outros métodos de compactação global de microcódigo. Entretanto, algumas de suas operações demonstram ser muito demoradas e ineficientes. Em [2] é feito uma breve análise de alguns dos gargalos encontrados na proposta de Fisher. Dentre elas, a que mais se destaca é o algoritmo de *bookkeeping*, onde se encontram as regras para as movimentações das MOP's entre os blocos básicos e duplicação das mesmas quando necessário. Em resumo, o que foi proposto era a continuação da redução do

Figure 3: Tempos de execução (em ciclos) para os vários paradigmas de compactação de microcódigo

Code Segment	Sequential	Hand Compacted	Trace Scheduled	Locally Compacted
Floating Add				
Initialization	15.5	7.5	9.5	13
Coefficient Shift (n = number of shift operations)	$3 + 3n$	$3 + n$	$3 + 2n$	$3 + 2n$
Add	9	6	6	8
Floating Multiply				
Initialization	33	14	14	22
Floating Divide				
Initialization	19.5	8.6	7.5	14.5
Loop (for each of 48 iterations)	4	3	3.4	4
Normalize	10.5	4.5	5.5	7.5

Table 1: Espaço ocupado pelo microcódigo horizontal

	Compactação Manual	Trace Schedule	Compactação ITS
MP1	26	29	29
MP2	19	25	19
MP3	27	33	29
MP4	19	22	20
MP5	26	26	26

tempo de execução dos programas compactados, com uma menor quantidade de cópias duplicadas. À estas melhoras, os autores chamaram de algoritmo *ITSC* (*Improved Trace Scheduling Compactio*).
Para a implementação e realização dos testes propostos em [2], foi utilizada a linguagem *PASCAL* em uma máquina *PDP-11/03*. O programa continha aproximadamente 3500 linhas de código e demorava aproximadamente 350 segundos para compactar um microcódigo de 49 MOP's. Estas MOP's também foram retiradas do microprogramas do emulador *PUMA* para o *Control Data 6600*.
As MOP's foram divididas em 5 grupos, a saber, {MP1, MP2, ..., MP5}. O critério de separação destes grupos foram os *branch*'s condicionais, isto é, cada *branch* condicional encontrado no caminho definia um novo grupo. A partir daí, mediu-se quantas replicações foram necessárias fazer das MOP's. O resultado obtido pode ser visto na Tabela 1.
Como pode ser visualizado, foi obtido uma significativa melhora na questão do espaço ocupado. A porcentagem desta melhoria se encontra entre 9% e 24%, comparando os algoritmos de *Trace Scheduling* original e o *ITS*.

6. RESTRIÇÕES DE TEMPO

Os estudos apresentados até então limitaram-se em analisar apenas restrições de dados e recursos dos programas a serem compactados. Incluir a restrição de tempo no algoritmo de *Trace Scheduling* pode também contribuir para estender suas funcionalidades. Além disso, esta modificação causa um grande impacto em sua eficiência e modo de implementação. Existem estudos e algoritmos para o problema de Com-

pactação Local de Microcódigo sob Restrição de Tempo [3]. Em [3], o autor propõe um algoritmo para este problema denominado *CAS* que vem a resolver este problema.
Para a Compactação Global de Microcódigo, a restrição de tempo é adicionada em cima do algoritmo proposto por Fisher. A proposta do autor mexe, basicamente, com o *DAG* gerado pelo algoritmo de *Trace Scheduling*. Agora, ao se fazer a movimentação de uma MOP, além da restrição de dados e recursos a serem observados, a restrição de tempo também deverá ser analisada. Esta restrição pode ser entendida como, ao se mover uma MOP entre os blocos básicos, a sua não interferência no tempo de execução de alguma MOP do bloco original, além da sua própria garantia de término dentro do tempo previsto. Isto deve-se porque, agora, cada MOP possui um limite de tempo para finalizar sua execução. Caso duas ou mais MOP's exijam terminar em um tempo pré-determinado, e ambas não possam ser escalonadas mutuamente, então acaba por existir um conflito de tempo entre elas.
A solução para estes casos passa por definir mais regras na construção do *DAG*. Agora, uma aresta deste grafo poderá ir da MOP M_1 para a MOP M_2 quando, além de respeitar todos os requisitos anteriores, todas as microinstruções contidas em M_1 finalizarem dentro do prazo estipulado para que se possa escalonar a MOP M_2.

7. OUTROS ALGORITMOS PARA O PROBLEMA DA COMPACTAÇÃO GLOBAL DE MICROCÓDIGO

Esta seção tem o intuito de mostrar outros algoritmos desenvolvidos para o mesmo problema. A maioria deles exploram os pontos fracos do *Trace Scheduling* mas, o que se pode notar, é que todos possuem uma semelhança muito grande com o algoritmo desenvolvido por Fisher.

7.1 Algoritmo de Upendra Rao

O algoritmo de *Upendra Rao* [7] possui a mesma idéia básica utilizada em [4]. Ele leva em consideração a possibilidade de mover as várias MOP's entre os blocos básicos de um determinado programa *P*. Para que ele seja melhor entendido, ilustramos alguns conceitos propostos pelo autor:

7.1.1 Conceitos básicos

Abaixo se encontram alguns conceitos básicos necessários para o entendimento deste algoritmo:

Conceito 1: Dizemos que uma microoperação MO_i é precedida MO_j ou, de outra forma, MO_j tem dependência de dados sobre MO_i se, MO_i usa o conteúdo de algum registrador que posteriormente MO_j irá sobreescrever, ou MO_i produz algum dado que posteriormente MO_j irá utilizar, ou MO_i escreve em algum registrador que posteriormente também será escrito por MO_j. Este tipo de dependência produz o mesmo *DAG* descrito nas seções anteriores.

Conceito 2: Dizemos que um registrador está *vivo* em um certo ponto se seu conteúdo, a partir daquele ponto, é utilizado antes de ser sobrescrito;

Conceito 3: Dizemos que uma microperação está *livre no topo* de um bloco B_i se ela não depende de dados de nenhuma outra operação dentro de B_i;

Conceito 4: Dizemos que uma microperação está *livre no final* de um bloco B_i se nenhuma outra operação dentro de B_i irá necessitar dos dados produzidos por ela.

Conceito 5: Seja o peso denominado *Block Dependency Weight* (BDW_i) de um bloco B_i. Este peso é determinado como sendo a quantidade de outros blocos que dependem dos resultados gerados por B_i.

7.1.2 Regras de Movimentação das Microperações

Com o intuito de compactar o código de forma a deixá-lo próximo da solução ótima, definimos algumas regras para que as movimentações possam ocorrer. Tais regras podem ser vistas abaixo:

Regra 1: Uma MO_i livre no final de um bloco B_i poderá ser movida para os blocos imediatamente posteriores (por ex. B_j e B_k), se os registradores escritos por MO_i não estão vivos em B_j ou B_k, respectivamente;

Regra 2: Uma microperação MO_i, livre no final, poderá ser movida para os blocos imediatamente posteriores (isto implica em poder fazer várias cópias);

Regra 3: Uma microperação MO_i, livre no topo de um bloco B_i, pode ser movida para os blocos imediatamente anteriores (também implica em poder fazer várias cópias);

Regra 4: Uma microperação em comum, MO_{jk}, que está livre no final, por exemplo, nos blocos B_j e B_k, poderá ser movida para o bloco imediatamente posterior (por exemplo, B_i);

Regra 5: Uma microperação em comum, MO_{jk}, que está livre no topo, por exemplo, nos blocos B_j e B_k, poderá ser movida para o bloco imediatamente anterior (por exemplo B_i);

Regra 6: Sejam os blocos B_i, B_j e B_k arranjados de tal forma que, B_i é uma raíz e tem como filhos B_j e B_k. Uma microperação MO_j, livre no topo de B_j, onde seus registrados não estão vivos em B_k, pode ser movida para B_i;

Regra 7: Sejam os blocos B_i, B_j e B_k arranjados de tal forma que B_j e B_k estão em paralelo, e ambos apontam para B_i. Uma microperação livre no final de B_j, com seus registrados não estando vivos em B_k, pode ser movida para o bloco B_i;

Regra 8: A transferência de operações condicionais entre os blocos é restrita, isto é, só serão feitas em raros casos;

7.1.3 Tipos básicos de microoperações

Ao analisar um bloco básico, digamos B_i, para sua compactação, existirá a possibilidade de se mover algumas de suas microperações para seus blocos descendentes. Além disso, existe também a possibilidade de que operações de blocos anteriores sejam movidos para este mesmo B_i. Dependendo da natureza da microperação a ser movida, podemos classificá-las em três categorias:

***Delay Microoperations*:** são operações presentes em B_i que podem ter sua execução adiada, isto é, podem ser movidas para blocos descendentes de B_i. Geralmente são as instruções livre no final do bloco B_i.

***Early Microoperations*:** são operações em B_i que podem ter sua execução adiantada, isto é, podem ser movidas para blocos predecessores de B_i. Geralmente são as instruções livre no topo do bloco B_i.

***Critical Microoperations*:** são operações em B_i que não podem ser movidas para nenhum outro bloco. Geralmente são instruções que ficam entre as duas anteriores.

7.1.4 Algoritmo de Compactação Global

Agora descrevemos o algoritmo de *Upendra* para o problema da compactação global. Dado um programa P dividido em blocos básicos, gere o Grafo de Dependência de Blocos (BDG). Analise quais são os registrados vivos em determinados pontos e siga os passos a seguir:

Primeiro Passo: marque todos os blocos como não compactados;

Segundo Passo: se todos os blocos estão compactados, PARE;

Terceiro Passo: pegue o bloco não compactado com maior BDW. Seja este bloco B_i;

Quarto Passo: Se este bloco já possui predecessores compactados, tente então mover operações de B_i para seus blocos predecessores, de forma que elas possam também ser executadas em paralelo, isto é, que não aumente a quantidade de operações em seqüência a serem realizadas no bloco anterior;

Quinto Passo: Selecione um caminho entre os blocos, onde B_i é o bloco raíz, e que a soma dos BDW dos blocos seja máxima;

Sexto Passo: Construa um conjunto de *Delay Microoperations* de B_i;

Sétimo Passo: Crie um novo conjunto de operações que possam ser executadas em paralelo;

Oitavo Passo: Construa um conjunto, denominado *D-set*, que possue todas as microoperações que já possuem os dados disponíveis para sua execução no caminho criado no **Quinto Passo**;

Nono Passo: Para cada microoperação em *D-set*, modifique os valores de suas prioridades. O novo valor será igual $D_j + C_j - P_j$, onde, D_j é o número de outras microoperações que dependem da microoperação j, C_j é a quantidade de conflitos causados entre a microoperação j e as outras microoperações, e P_j é um *valor de redução de prioridade*, que possue um valor padrão pré-definido;

Décimo Passo: Selecione as MO's do conjunto *D-set*, em ordem descrescente, e as adicione no conjunto criado no **Sétimo Passo**;

Décimo Primeiro Passo: Se todas as MO's contidas no bloco raiz B_i estão compactadas, então vá para o **Décimo Segundo Passo**, senão, vá para o **Sexto Passo**;

Décimo Segundo Passo: caso existam MO's ainda presentes no conjunto *D-set*, tente colocá-las dentro do conjunto gerado no **Sétimo Passo**;

Décimo Terceiro Passo: Execute o algoritmo de *Bookkeeping* se necessário; vá para o **Segundo Passo**;

7.1.5 Complexidade do algoritmo

Considere um programa P que possui M blocos básicos e N microoperações no toal. Assumindo que estas N microoperações estejam uniformemente distribuídas entre os M blocos, a complexidade deste algoritmo, no pior caso, é $O(N\rho^2)$, onde $\rho = (\frac{N}{M})$.

7.2 Wavefront Scheduling

Em [1] é apresentada uma técnica denominada *Wavefront Scheduling*. Esta técnica foi implementada em compiladores para a arquitetura *IA-64*, que é um arquitetura rica em recursos para a exploração de paralelismo a nível de instrução.

Wavefront Scheduling é uma implementação particular de uma técnica denominada *Global Code Scheduler*. Ela também se baseia fortemente em grafos de dependência de dados e resolução de conflitos. Assim como as outras técnicas citadas em [4, 7], existe em diversos momentos a utilização de heurísticas para determinar prioridades de microoperações e definir caminhos sem ciclos no grafo de dependência de dados.

Um dos principais pontos do desenvolvimento deste algoritmo é uma técnica chamada *P-ready code motion*. Esta técnica se resume em um conjunto de regras, semelhantes às apresentadas na seção 7.1.2, para a movimentação das microoperações dentro do novo código gerado, com o intuito de gerar a menor quantidade de duplicatas possíveis, e que possam rodar em um menor tempo.

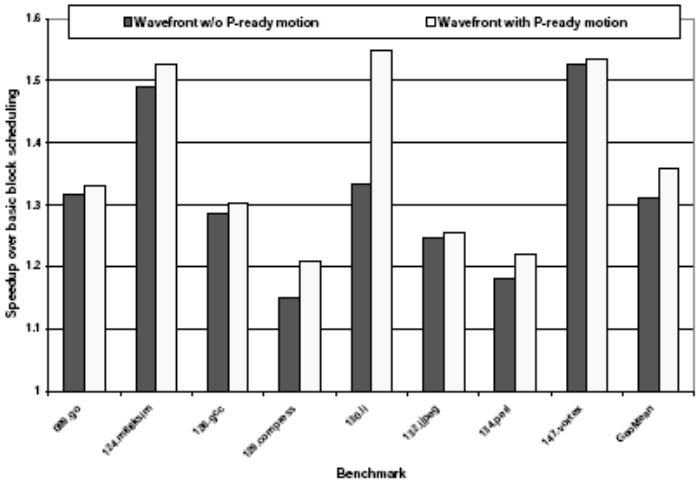
Após a implementação do algoritmo foram feitas várias medidas de desempenho utilizando o *benchmark suit* SPEC95. A Figure 4 mostra o *speedup* conseguido utilizando o *Wavefront Scheduling* com e sem a técnica *P-Ready*. Este teste foi realizado no processador *Merced*, da *Intel Corporation*. A melhora alcançada chegou a 50%, como no caso do *benchmark 124.m88skin*.

7.3 Algoritmos ótimos versus Heurísticas para o problema da Compactação Global de Microcódigo

Em todos os algoritmos apresentados até então, heurísticas foram utilizadas em diversos pontos do problema. A utilização deste tipo de algoritmo é principalmente encorajada devido a complexidade de se achar a solução ótima. O que foi visto é que, embora não tenhamos certeza da possibilidade de melhorias no código gerado, em grande parte das instâncias obtivemos bons resultados, o que acabou por consolidar este tipo de abordagem. Além disso, na maioria das vezes o código gerado utilizando as técnicas descritas superaram o código paralelo escalonado manualmente.

Porém em [8] o autor propõe a utilização de uma técnica denominada *ILP* (*Integer Linear Programming*) para a resolução do problema de escalonamento das microinstruções.

Figure 4: Speedup alcançado com a utilização do *Wavefront Scheduling* sem e com a utilização da técnica *P-Ready motion*



Esta técnica foi testada e implementada em um compilador para a arquitetura *Intel Itanium*®. Com ela, é possível chegar sempre na solução ótima do problema, embora isto não possa ser conseguido a um baixo custo.

Os resultados obtidos em [8] foram satisfatórios do ponto de vista do ganho alcançado sobre o rendimento dos novos códigos. Em média, códigos gerados com esta resolução obtiveram um *speedup* de 10% sobre códigos gerados utilizando-se da técnica *GCS*. Além disso, o tamanho dos novos códigos obtiveram uma redução aproximada de 32%.

Apesar deste ganho aparentemente suficiente para que se faça sempre sua utilização, o fator tempo foi preponderante para contrabalancear seus ganhos. Ela se mostrou eficiente para trechos de códigos que tenham entre 500 e 750 instruções. Menos do que esta quantidade, o algoritmo se mostrou tão eficiente quanto os outros. Para maiores quantidade, o tempo levado para a compilação do código foi demasiadamente alto, o que torna proibitivo a utilização deste algoritmo em grandes programas. Entretanto, se o tempo de compilação não for um requisito altamente necessário em certos casos, a utilização deste algoritmo poderá ser perfeitamente viável.

Testes práticos foram feitos em 5 *integer benchmarks* da *suite SPEC*®*CPU 2006*. A tabela 2 mostra quais *benchmarks* foram utilizados, e que ganho se obteve com eles.

8. CONCLUSÕES

Neste artigo foram abordadas diversas técnicas para o problema da Compactação Global de Código, onde se deu maior ênfase no algoritmo de *Trace Scheduling* [4], proposto por Fisher em 1981. Foram feitas comparações com métodos manuais até então utilizados pelos programadores, bem como com a utilização de outras técnicas propostas na mesma época. Após isso, discorreu-se sobre técnicas mais atuais, como o *Wavefront Scheduling*, que já foram implementadas e testadas em processadores modernos como, por exemplo, a arquitetura *IA-64*. Encerramos a análise com uma implementação para este problema utilizando-se da técnica conhecida como *ILP*, ou Programação Linear Inteira. Para esta abordagem, que consegue chegar na solução ótima do problema, também foram feitas implementações em processadores modernos bem como testes que provaram sua eficácia na geração de código paralelo, em contrapartida com sua demora no processo de geração do novo código.

Table 2: *Speedup* nos códigos gerados utilizando técnicas *GCS* e resolução com *ILP*

Benchmark	Rotinas	Instr.	Blocos	Speedup
400.perlbench	32	25702	8589	12%
401.bzip2	11	11729	1625	10%
445.gobmk	44	27263	6339	7%
458.sjeng	14	9362	2342	11%
473.astar	3	725	90	10%
Total	104	74781	18985	10%

9. REFERENCES

[1] J. Bharadwaj, K. Menezes, and C. McKinsey. Wavefront Scheduling: Path Based Data Representation and Scheduling of Subgraphs. In *32nd Annual International Symposium on Microarchitecture*, pages 262–271, 1999.

[2] S. Bogongo, D. Shiyuan, and J. Lan. An Improvement of Trace Scheduling for Global Microcode Compaction. In *MICRO - 17th annual workshop on Microprogramming*, pages 78–85, December 1984.

[3] S. Bogongo, J. Wang, and J. Xia. Microcode Compaction with Timing Constraints. In *20th Annual Workshop on Mcroprogramming*, pages 59–68, 1987.

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

[5] C. Foster and E. M. Riseman. Percolation of code to enhance parallel dispatching and execution. *IEEE Trans. Comput.*, C-21:1411–1415, December 1972.

[6] R. Grishman and S. Bogong. A Preliminary Evaluation of Trace Scheduling for Global Microcode Compaction. *IEEE Transactions on Computers*, C-32 Issue-2:1191–1194, December 1983.

[7] S. U. Rao and A. K. Majumdar. An Algorithm for Global Compaction Of Horizontal Microprograms. *ACM SIGMICRO Newsletter*, 17 Issue 2:19–30, July 1986.

[8] S. Winkel. Optimal versus heuristic global code scheduling. In *40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 43–55, December 2007.

[9] S. Yau S., C. Schowe A., and M. Tosuchiya. On storage optimization of horiontal microprograms. In *7th Annu. Microprogramming Workshop*, pages 98–106, 1974.