Um Gerador Automático de Código para Arquiteturas VLIW com Capacidade de Execução Condicional

Anna Dolejsi Santos e João Francisco Pereira Neto
Deptº da Ciência de Computação – UFF[1]
e-mail: annads@dcc.uff.br

Sumário

Nesse artigo apresentamos um gerador automático de código paralelo para o modelo CONDEX, uma arquitetura Superescalar VLIW, que implementa o conceito de execução condicional. A arquitetura é formada por múltiplas unidades funcionais independentes que podem operar concorrentemente, processando diversas instruções procedentes do mesmo programa de aplicação. Além da descrição do modelo de arquitetura, examinamos a implementação do gerador de código paralelo que emprega a compactação condicional, uma técnica de compactação global. Finalmente, avaliamos a qualidade do código paralelo gerado, interpretando um conjunto de programas de teste.

Abstract

In this paper we present a parallel code generator for the CONDEX machine model, which is a VLIW machine supporting the conditional execution concept. The basic machine includes multiple functional units that can operate in parallel, executing several machine instructions proceeding from the same application program. In addition to the description of the architecture model, the paper presents the parallel code generator implementation using a conditional compaction, a global compaction technique. By interpreting the paralel code derived from a suite of test programs, we evaluated the quality of the generated code.

1. Introdução

Equipadas com múltiplas unidades funcionais independentes que podem ser ativadas em paralelo, as Arquiteturas Superescalares, são capazes de executar simultaneamente, durante cada ciclo de máquina, diversas instruções procedentes de um mesmo programa de aplicação [6, 11].

Máquinas *Very Large Instruction Word* (VLIW) são exemplos de processadores Superescalares com algoritmo de escalonamento de instruções estático [2, 6, 19], que surgiram como conseqüência do desenvolvimento de técnicas de compactação global de micro-programas [8, 17, 24].

É tarefa do algoritmo de escalonamento, detectar e selecionar as instruções do programa de aplicação que podem ser executadas durante cada ciclo de máquina. Tendo como objetivo explorar efetivamente, o paralelismo potencial do processador Superescalar e reduzir o tempo de execução de programas do usuário, ele modifica a ordem de execução das instruções e determina quais são as unidades funcionais da arquitetura que devem ser ativadas.

O escalonamento (ou compactação) das instruções das arquiteturas VLIW é realizado, na fase de geração de código, por um algoritmo implementado pelo *software* de suporte da arquitetura. Independentemente da técnica empregada, o escalonamento deve considerar a interdependência das instruções (i.e., as dependências de dados) e a limitação de recursos existentes na arquitetura [6, 19].

Desse modo, a unidade de controle de uma arquitetura VLIW faz em cada ciclo do processador, a busca do grupo de instruções que serão executadas, conforme

[1]Universidade Federal Fluminense, Praça do Valonguinho s/n, Instº de Matemática 4.º andar, Deptº da Ciência de Computação, Centro, Niterói, RJ, CEP 24210-130.

especificado em tempo de geração de código pelo algoritmo de escalonamento de instruções.

Denominamos o grupo de instruções que terão suas execuções iniciadas no mesmo ciclo de máquina, de "instrução longa" (ou "Instrução Multifuncional" – IMF).

Existe na IMF, um campo indicando a função que deverá ser executada por cada unidade funcional presente no processador. Todas as instruções indicadas nos vários campos de uma IMF serão executadas em paralelo. Códigos de operação do tipo NOP (*no operate*) deverão ser introduzidos nos campos correspondentes às unidades funcionais que não devem iniciar uma nova instrução no momento que a instrução longa for executada.

A Figura 1.1 ilustra uma instrução longa. Cada campo da IMF contém o código de operação (OPCODE) e os operandos necessários para a execução da instrução especificada (Ri, Rj e Rk).

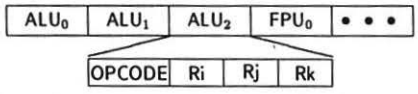

Figura 1.1: Uma Instrução Longa e o Detalhamento de um dos seus Campos

Durante o escalonamento (ou compactação) de instruções podem ser empregadas duas estratégias básicas: a compactação local e a compactação global.

Na compactação local, o alcance das atividades do algoritmo restringe-se aos comandos do bloco básico. Somente após o tratamento de um bloco básico, é que o algoritmo de compactação local examina um outro bloco [12].

Por sua vez, as técnicas de compactação global não se restringem às fronteiras dos blocos básicos. Examinam o programa integralmente, movimentando instruções de um bloco básico para um outro [8, 17, 24]. Conseqüentemente os algoritmos de compactação global, permitem empacotar um número maior de instruções nas instruções multifuncionais.

Apesar do avanço verificado na área de geração de código paralelo para arquiteturas VLIW, ainda hoje observa-se que o nível de ociosidade de suas unidades funcionais é considerável. A ocorrência de freqüentes instruções de desvio no programa de aplicação é um dos principais fatores que limitam o desempenho do algoritmo de escalonamento [4].

Ao especificar uma arquitetura VLIW com a capacidade de execução condicional, estávamos interessados em reduzir o custo imposto pelas instruções de desvio. Mais especificamente, ao invés de preenchermos os campos vazios da IMF com NOPs, a capacidade de execução condicional nos permite o escalonamento de instruções provenientes de blocos básicos distintos numa mesma IMF.

No modelo de processamento empregado, o início da execução das instruções compreendidas em uma IMF, depende do conteúdo dos *flags* do processador. Desse modo, somente as instruções pertencentes ao caminho determinado pela instrução de desvio é que serão iniciadas.

Esse trabalho está organizada em cinco seções. A próxima seção descreve o modelo de processamento que possibilita a execução condicional das instruções empacotadas em IMFs do programa executável. A Seção 3 examina a automatização do processo da geração de código para o nosso modelo de arquitetura. Na Seção 4 descrevemos os experimentos realizados para avaliar a qualidade do código paralelo, e finalmente destinamos a Seção 5 às conclusões.

2. O Modelo de Processamento

A execução condicional vem sendo investigado por diversas universidades e fabricantes de processadores [9, 18, 19, 22].

Ao definir e simular, um modelo de arquitetura que incorpora o conceito de execução condicional, tínhamos entre outros objetivos, obter uma plataforma para o desenvolvimento e avaliação de alternativas técnicas de geração de código paralelo.

Nessa seção apresentamos inicialmente o conceito de execução condicional e em seguida descrevemos a arquitetura modelo para a qual estamos gerando código paralelo automaticamente.

2.1. A Execução Condicional

No nosso modelo básico de processador VLIW, as instruções presentes em uma instrução longa, podem ser executadas condicional ou incondicionalmente. Desse modo, podemos empacotar em uma mesma instrução longa, instruções provenientes de blocos básicos distintos, tornando assim desnecessário introduzir no projeto da máquina, mecanismos de predição de desvios e código de reparo [10, 13, 21, 23].

Dizemos que um "bloco sucessor" (ou "bs"), é o bloco básico para onde o fluxo de controle é dirigido, quando na linguagem de alto nível temos uma estrutura do tipo IF *condição* THEN... (isto é, não existe ELSE correspondente ao THEN), e em tempo de execução, *condição* é verificada como sendo falsa.

A idéia fundamental é que durante o processo de compactação do código seqüencial, sejam empacotadas em uma mesma instrução longa, instruções oriundas de blocos básicos correspondentes as estruturas em linguagem de alto nível, do tipo "THEN e ELSE," ou "THEN e bloco sucessor". Durante a ativação da instrução longa, são executadas apenas as instruções cujas condições forem verdadeiras.

Para facilitar a apresentação, abreviamos os termos "bloco THEN" e "bloco ELSE" como "bt" e "be" respectivamente.

```
if (i and 1) = 1
  then odd := odd + 1
  else even := even + 1;
```

(a)

```
if ( (entr and mask) <> 0 )
  then sum := sum + wgh;
wgh := wgh * 2;
```

(b)

Figura 2.2: Trechos dos Programas *Branch* e *BCD*

No trecho de programa mostrado na Figura 2.2(a), vemos que o bloco básico correspondente ao "bt" é composto pelo comando $odd := odd + 1$, e o correspondente ao "be" é composto pelo comando $even := even + 1$. Nesse caso, ao compactarmos em instruções longas, instruções provenientes dos dois blocos básicos examinados, é possível em tempo de execução, executar somente aquelas instruções cuja condição for satisfeita. Em outras palavras, se $(i \text{ and } 1) = 1$, as instruções provenientes do "bt" são executadas. Contudo se a condição for falsa então são executadas as instruções do "be."

Já na Figura 2.2(b) o bloco básico correspondente ao "bt" é formado pelo comando $sum := sum + wgh$, ao passo que o "bs" é formado pelo comando $wgh := wgh * 2$. Dependendo da condição teremos em tempo de execução, instruções dos dois blocos sendo executadas simultaneamente, ou apenas as do bloco sucessor. Isto é, se $(entr \text{ and } mask) <> 0$, então todas as instruções contidas na instrução longa são executadas. Se a condição for falsa, então apenas as instruções oriundas do "bs" são executadas. Podemos então concluir que as instruções do "bs" são sempre executada, ou seja, são executadas incondicionalmente.

2.2. O Modelo CONDEX

Por simplicidade, chamamos o modelo de arquitetura Superescalar VLIW com capacidade de EXecução CONDicional de CONDEX [7, 19, 20].

O modelo CONDEX, ilustrado na Figura 2.3, representa uma família de máquinas VLIW. Cada componente dessa família consiste de múltiplas unidades funcionais que podem operar em paralelo. O modelo básico é dotado de quatro tipos de unidades funcionais:

- ALU – unidade aritmética e lógica para inteiros;
- FPU – unidade para aritmética em ponto flutuante;
- MEM – unidade de acesso à memória; e
- BR – uma única unidade de processamento de desvios.

Além das unidades funcionais, o modelo possui um Banco de Registradores e diversos Conjuntos de Indicadores de Condição. A partir desse modelo é possível configurar com várias unidades funcionais de cada tipo (exceto a unidade de processamento de desvios, que é unica), máquinas derivadas do modelo básico.

Um conjunto de indicadores de condição do modelo CONDEX, é constituído por dois *flags* (zero (Z) e negativo (N)), e possui largura de dois bits (um bit para cada *flag*). Os *flags* preservam seu conteúdo até que uma instrução de comparação (ICMP ou FCMP), os modifique.

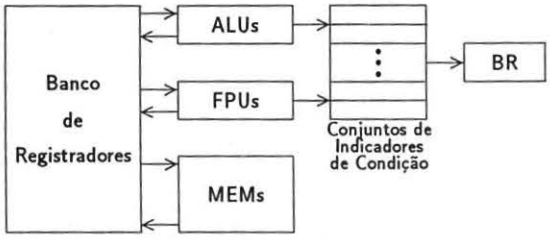

Figura 2.3: Esquema da Arquitetura Modelo – CONDEX

As instruções de comparação possuem um campo indicando qual dos conjuntos será afetado. Os *flags* Z e N refletem o resultado da execução de uma instrução de comparação do seguinte modo:

- Z = 1 e N = 0 $\Longrightarrow$ se o resultado da comparação for zero (igualdade);
- Z = 0 e N = 0 $\Longrightarrow$ se o resultado da comparação for positivo (maior que);
- Z = 0 e N = 1 $\Longrightarrow$ se o resultado da comparação for negativo (menor que).

A existência de múltiplos conjuntos de indicadores de condição, viabiliza que instruções provenientes das estruturas "bt-be"e "bt-bs" aninhadas, sejam compactadas em instruções multifuncionais comuns (isto é, IMFs que podem conter intruções dos diversos blocos aninhados). Em outras palavras, se um outro conjunto de indicadores de condição for especificado, uma instrução de comparação aninhada ao ser executada, não destruirá o conteúdo dos *flags* modificados por uma instrução de comparação executada anteriormente.

Para permitir a execução condicional, além do código de operação e dos operandos (OPCODE, Ri, Rj, e Rk), cada campo da intrução longa do modelo CONDEX, possui três campos adicionais que especificam as condições que devem ser satisfeitas para que a instrução seja executada. A Figura 2.4 ilustra um campo de uma IMF do modelo CONDEX.

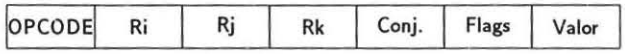

Figura 2.4: Um Campo da IMF do Modelo CONDEX

Na Figura 2.4, os campos "Conj.", "Flags" e "Valor" habilitam a execução da instrução e especificam respectivamente:

- um dos conjuntos de indicadores de condição;
- os valores que os bits testados devem conter; e
- quais os bits do conjunto devem ser testados.

Em tempo de execução, os bits especificados no campo "Valor," do conjunto de indicadores de condição selecionado, é comparado com os bits correspondentes no campo "Flags." Se o teste for verdadeiro, a instrução é executada pela unidade funcional correspondente ao campo. Caso contrário, a instrução é ignorada. Esse teste é realizado para cada instrução contida na IMF.

Para explorar mais eficientemente a concorrência do código seqüencial (eliminando as anti-dependências), fragmentamos cada ciclo de processador do CONDEX em quatro subciclos. A leitura dos registradores fonte das operações é feita pelas unidades funcionais no primeiro subciclo. e a atualização dos registradores destinos das operações ocorre no quarto subciclo. Isso permite, por exemplo, que uma unidade funcional que use um registrador como operando fonte, obtenha seu conteúdo no primeiro subciclo, possibilitando que o mesmo registrador possa ser usado como

destino por uma outra unidade funcional (ou a mesma), já que o valor do registrador somente será modificado no último subciclo.

As unidades funcionais do modelo CONDEX, são capazes de executar um conjunto de instruções que se enquadram nas seguintes categorias: transferência com memória, aritmética com ponto fixo, aritmética com ponto flutuante, lógicas, de desvio, transferência entre registradores e as operações NOP e HALT.

Finalmente, para não limitar o desempenho do modelo, alongando o ciclo de processador para igualar a latência da instrução mais complexa, definimos latências diferenciadas, que vão de 1 até 4 ciclos de máquina, para as diferentes instruções executadas pelas unidades funcionais.

3. A Geração do Código Paralelo

Programas de aplicação, escritos em uma linguagem Pascal-*like*, são traduzidos pelo compilador [15], para uma forma intermediária seqüencial. O código intermediário produzido, é composto pelas instruções, que as unidades funcionais do modelo CONDEX executam. A forma geral dessas instruções é:

CÓDIGO DE OPERAÇÃO	[DESTINO]	$OPERANDO_1$	...	$OPERANDO_n$

A escolha das instruções do programa de aplicação que serão executadas em cada ciclo de máquina, é feita durante a geração do código paralelo, isto é, durante o processo de compactação do código intermediário.

A geração de código paralelo para o modelo CONDEX, compreende duas etapas distintas:

- a da compactação local, e
- a da compactação global.

Os algoritmos *Trace Scheduling* [8] e *Percolation Scheduling* [17], são exemplos clássicos de algoritmos de compactação global para processadores VLIW. Eles porém não implementam o conceito de execução condicional e por esse motivo não são apropriados para paralelizar código para o nosso modelo de processador.

Para superar essa dificuldade, desenvolvemos uma técnica alternativa de compactação global, denominada "compactação condicional."

3.1. A Etapa da Compactação Local

O programa responsável pela compactação local (o Compactador Local [15], recebe como entrada o código intermediário seqüencial, identifica os blocos básicos do programa e constrói o grafo do fluxo de controle correspondente [6].

O processo de compactação local é realizado separadamente para cada bloco básico do grafo de fluxo do programa. A partir das instruções de cada bloco básico, obtemos como resultado, uma seqüência de instruções multifuncionais. Durante o processo de compactação local, os campos "Conj", "Flags" e "Valor," que habilitam a execução de instruções presentes nas IMFs, não são considerados.

Inicialmente o Compactador Local, associa uma instrução longa a cada instrução de um bloco básico B. Em seguida, procura movimentar instruções uma a uma, para cima, tentando grupá-las em instruções longas. Esse processo emprega uma versão do algoritmo *list-scheduling* [12].

A movimentação de instruções leva em conta as dependências de dados, os conflitos na utilização dos recursos presentes na configuração da máquina e o tempo de latência das unidades funcionais. No final da compactação, o algoritmo garante que o bloco resultante é semanticamente equivalente ao bloco original.

O Compactador Local é controlado por parâmetros. São exemplos de parâmetros que influenciam o processo de compactação de código, o número e tipo de unidades funcionais e o tempo de latência das diferentes operações.

Finalmente, após escalonar as intruções de cada bloco básico do programa, o Compactador Local atualiza o grafo de fluxo de controle, modificando os endereços das instruções de desvio de modo a refletir os novos endereços do programa paralelizado.

3.2. A Etapa da Compactação Condicional

O processo de compactação condicional [19], automatizado recentemente [16], utiliza como entrada os blocos básicos gerados pelo Compactador Local e produz como resultado um programa objeto paralelo que pode ser interpretado no simulador do modelo CONDEX.

```
1. for i = 1 to penúltimo bb do programa do
      if bb_i é um bb ainda não examinado and bb_i é um bloco do tipo bt
        then if ∃ o bloco be correspondente
                then begin
                       marca o par de blocos como bt-be;
                       marca o bloco be como já examinado;
                     end
             else if ∃ o bloco bs correspondente
                then begin
                       marca o par de blocos como bt-bs;
                       marca o bloco bs como já examinado;
                     end;
```

Figura 3.5: Identificação dos Pares "bt-be" e "bt-bs"

```
2. while ∃ par bt-be do
   begin
     i := 1;                                          {i controla o n.º de IMFs do bt}
     for j = 1 to n.º de IMFs do be do                {j controla o n.º de IMFs do be}
        while ∃ instrução na IMF_j do
           begin
             for k = 1 to n.º de instruções na IMF_j do  {k controla o n.º de instr. na IMF_j}
               if instrução_k ≠ NOP
                 then begin
                        if instrução_k não causa conflito de recursos na IMF_i do
                        bt and por todos os n ciclos de latência da instrução_k não causa
                        conflito de recursos com IMF_{i+1}, ... , IMF_{i+n} do bt
                          then begin
                                 move instrução_k para IMF_i do bt;
                                 reserva a unidade funcional correspondente por n
                                 ciclos;
                                 introduz máscara no campo apropriado da IMF_i;
                               end;
                      end;
                 i := i + 1;
                 if i > n.º de IMFs do bt
                   then goto 20;
           end;
20: if ∃ IMF vazia no be and ∃ UF reservada nesse ciclo
      then elimina IMF;
    if endereço da última IMF de be > endereço da última IMF de bt or endereço da
    última IMF de bt > endereço da última IMF de be
      then introduz instrução JUMP na IMF de endereço menor para saltar IMFs do
      bloco maior;
    elimina IMFs anteriores ao bt que contêm as instruções de desvio;
   end;
```

Figura 3.6: Segunda Fase do Algoritmo – Fusão dos Pares "bt-be"

Inicialmente, as estruturas "bt e be" (i.e., pares de blocos básicos "THEN" e "ELSE" provenientes do comando "IF" da linguagem Pascal) e "bt e bs" (pares de blocos básicos provenientes de um comando "IF" sem a cláusula "ELSE") são identificadas. Em seguida, o algoritmo de compactação condicional tentará alocar num único bloco de IMFs, instruções pertencentes aos blocos básicos de cada par. Isto é, sempre que for possível, instruções do bloco "be" serão escalonadas nas IMFs contendo instruções do bloco "bt," e as instruções do bloco "bs" serão alocadas nas instruções multifuncionais contendo instruções do bloco "bt." Um esboço dos passos realizados durante o processo de compactação condicional está ilustrado nas Figuras 3.5 a 3.8. Para facilitar a apresentação dessas figuras, o termo bloco básico foi abreviado para "bb."

```
3. while ∃ par bt-bs do
   begin
     i := 1;                                          {i controla o n.º de IMFs do bt}
     for j = 1 to n.º de IMFs do bs do                {j controla o n.º de IMFs do bs}
         while ∃ instrução na IMF_j do
            begin
              for k = 1 to n.º de instruções na IMF_j do  {k controla o n.º de instr. na IMF_j}
                if instrução_k ≠ NOP
                  then begin
                         if instrução_k não causa conflito de recursos na IMF_i do
                         bt and a instrução_k não causa conflito de dados com IMF_i do
                         bt and por todos os n ciclos de latência da instrução_k não causa
                         conflito de recursos com IMF_{i+1}, ... , IMF_{i+n} do bt
                            then begin
                                   move instrução_k para IMF_i do bt;
                                   reserva a unidade funcional correspondente por n
                                   ciclos;
                                   introduz máscara no campo apropriado da IMF_i;
                                 end;
                       end;
                  i := i + 1;
                  if i > n.º de IMFs do bt
                    then goto 30;
            end;
30: if ∃ IMF vazia no bs and ∄ UF reservada nesse ciclo
      then elimina IMF;
    if endereço da última IMF de bs < endereço da última IMF de bt
      then introduz instrução JUMP na última IMF de bs para saltar IMFs do bt;
    elimina IMFs anteriores ao bt que contêm as instruções de desvio;
  end;
```

Figura 3.7: Terceira Fase do Algoritmo – Fusão dos Pares "bt-bs"

```
4. Atualiza endereços das instruções de desvio de todo o programa compactado;
```

Figura 3.8: Quarta Fase do Algoritmo – Atualização dos Endereços

A primeira fase do algoritmo de compactação condicional realiza a identificação dos pares de blocos "bt e be" e "bt e bs". A Figura 3.5 ilustra essa fase.

Após a identificação dos pares "bt e be" e "bt e bs," o algoritmo realiza a fusão dos pares que foram marcados durante a primeira fase. As Figuras 3.6 e 3.7

ilustram respectivamente, o escalonamento de instruções do "be" em IMFs do "bt" e de instruções do "bs" em instruções longas do "bt."

Finalmente na quarta fase, os endereços das instruções de desvio são corrigidos (vide Figura 3.8).

Durante o processo de compactação condicional, o algoritmo movimenta as instruções para cima (uma instrução por vez), tentando grupá-las em instruções longas parcialmente preenchidas pelo processo de compactação local. Analogamente ao que ocorre durante a compactação local, o processo de compactação condicional leva em conta as dependências de dados (verdadeiras e de saída) e os conflitos na utilização de recursos.

Durante a fusão dos pares "bt-be" e "bt-bs," o Compactador Condicional preenche os campos "Conj," "Flags" e "Valor" convenientemente, remove as IMFs que tornaram-se vazias, e atualiza os endereços de desvio.

Se for vantajoso, o algoritmo pode introduzir instruções de desvio incondicional para transferir o fluxo de controle diretamente para IMFs relacionadas com os blocos "be" ou "bs." Isso ocorre toda vez que o algoritmo detecta que parte do bloco resultante da fusão é formada somente por instruções pertencentes a um dos blocos. Nesse caso específico, a introdução de uma instrução de desvio evitará a busca daquelas IMFs contendo apenas instruções do bloco "THEN" (ou do bloco "ELSE") e que não serão executadas se a condição do comando "IF" for falsa (ou verdadeira).

4. Os Experimentos

Para avaliar a qualidade do código paralelo produzido pelo processo de compactação condicional automatizado, utilizamos três configurações distintas do modelo CONDEX: M_1, M_2 e M_3. Temos então:

- M_1 com 2 ALUs, 1FPU, 1 MEM e 1 BR;
- M_2 com 4 ALUs, 2FPU, 2 MEM e 1 BR; e
- M_3 com 8 ALUs, 4FPU, 4 MEM e 1 BR.

Desse modo, M_1, M_2 e M_3 são máquinas CONDEX que podem executar respectivamente, até 5, 9 e 17 operações em paralelo. As três configurações estão equipadas ainda com 4 conjuntos de indicadores de condição.

A avaliação de desempenho foi realizada com os seguintes programas de teste: *Livermore Loop número 24* [14], *Branch* [5], *BCD-bin* [1], *Simpson* [3]; e *Árvore* [25].

Com essas configurações da máquina CONDEX e com esse conjunto de programas de teste, realizamos dois tipos de experimentos: estáticos e dinâmicos. Os experimentos estáticos nos possibilitam avaliar a explosão de código, e os dinâmicos a variação no tempo de execução.

Com o auxílio do compilador, geramos inicialmente o código seqüencial de cada programa de teste. Em seguida, através do processo automático de compactação, obtivemos código paralelo para cada configuração de máquina (isto é, para cada programa de teste produzimos três programas objeto).

O código paralelo foi então interpretado no simulador do modelo CONDEX (configurado segundo M_1, M_2 e M_3), permitindo-nos desse modo, comparar a qualidade do código obtido automaticamente (através do Compactador Condicional [16], com a qualidade do código produzido manualmente [7, 19, 20].

Nas Tabelas 4.1 e 4.2 temos respectivamente, os resultados estáticos e dinâmicos, observados durante os experimentos. Nessas tabelas "MAN" refere-se ao código gerado manualmente e "AUT" ao gerado automaticamente.

Na Tabela 4.1, que contém o número de instruções longas de cada programa executável, vemos que a compactação condicional automática é melhor para os programas *Livermore* e *BCD*, e a manual mostrou-se mais eficiente para os programas *Branch*, *Simpson* e *Árvore*. Essa situação já era esperada, já que o ser humano,

é capaz de visualizar oportunidades que o processo compactação automático, não consegue detectar.

	M_1		M_2		M_3	
PROGRAMA	MAN	AUT	MAN	AUT	MAN	AUT
Livermore	110	105	66	66	54	54
Branch	73	80	56	60	51	53
BCD	201	190	142	132	111	102
Simpson	308	325	210	224	188	189
Árvore	271	312	197	231	171	202

Tabela 4.1: Nº de IMFs do Código Executável

A Tabela 4.2 mostra a aceleração (*seedup*) obtida durante a execução dos códigos gerados manualmente e automaticamente. O *speedup* de referência (isto é, *speedup* igual a 1,000) corresponde à execução seqüencial [19], de cada programa de teste.

	M_1		M_2		M_3	
PROGRAMA	MAN	AUT	MAN	AUT	MAN	AUT
Livermore	1,001	1,022	1,691	1,486	1,692	1,487
Branch	1,159	1,159	1,457	1,457	1,500	1,500
BCD	0,997	1,008	1,429	1,429	1,950	1,765
Simpson	1,195	1,204	1,669	1,687	1,803	1,787
Árvore	1,211	1,156	1,633	1,538	1,885	1,778

Tabela 4.2: *Speedups* Atingidos com Referência à Execução Seqüencial

Na Tabela 4.2 vemos que o *speedup* alcançado durante a interpretação do código gerado automaticamente, também difere do observado durante a interpretação do gerado manualmente. Em boa parte dos casos, o código gerado manualmente possibilita uma maior aceleração. Contudo é importante lembrar que produção manual de código é lenta e inviável para grandes programas de teste.

5. Conclusões

Automatizamos o processo de geração de código paralelo para o modelo CONDEX, uma arquitetura com capacidade de execução condicional.

A geração automática de código nos propicia a obtenção mais rápida de código executável, permitindo-nos realizar um maior número experimentos, isto é, será possível interpretar um número maior de programas de teste em um conjunto mais amplo de configurações CONDEX.

Esperamos, com esses novos experimentos, definir uma configuração do CONDEX que apresente uma relação "custo × desempenho" ideal, além de obter dados para estabelecer o critério para a compactação das operações de blocos do tipo Bloco Sucessor em instruções que contém instruções de blocos THEN, de modo a garantir a eliminação da anomalia constatada durante a pesquisa descrita em [19].

Atualmente estamos expandindo o processo de compactação condicional, de modo que, instruções oriundas de mais de dois blocos básicos sejam compactadas em IMFs comuns, isto é, pensamos escalonar em instruções multifuncionais, instruções cujas execuções sejam dependentes do conteúdo de diferentes conjuntos de indicadores de condição. O modelo CONDEX está equipado com diversos desses conjuntos e nos propicia essa alternativa. Esperamos assim, explorar mais intensivamente as características do nosso modelo de processador, e em conseqüência, obter resultados de desempenho mais expressivos.

Finalmente, lembramos que paralelamente estamos investigando outros métodos para geração de código executável para o nosso modelo de processador: pretendemos utilizar de técnicas de compactação baseadas no perfil de execução dos programas [8, 19].

Referências

[1] A. K. Uht, "Hardware Extraction of Low Level Cocurrency from Sequential Instruction Streams," Ph.D. Thesis, Carnegie-Mellon University, December 1985.

[2] R. P. Colwell, R. P. Nix, J. J. O' Donnell, D. B. Papworth and P. K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," IEEE Transactions on Computers, Vol. 37, No. 8, August 1988, pp. 967–979.

[3] S. D. Conte, "Elementary Numerical Analysis," McGraw-Hill, NY, USA, 1965.

[4] J. W. Davidson and D. B. Whalley, "Reducing the Cost of Branches by Using Registers," The 17th Annual ISCA, Washington, 1990, pp. 182–191.

[5] D. R. Ditzel, e H. R. Mclellan, "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero," Proceedings of the 14th Annual ISCA, 1987, pp. 2–9.

[6] Edil S. T. Fernandes e Anna Dolejsi Santos, "Arquiteturas Super Escalares: Detecção e Exploração do Paralelismo de Baixo Nível," VIII Escola de Computação, 1992.

[7] Edil S. T. Fernandes, Anna Dolejsi Santos and Claudio L. de Amorim, "Conditional Execution: an Approach for Eliminating the Basic Block Barriers," Microprocessing and Microprogramming The Euromicro Journal, North Holland, Vol. 40, Numbers 10–12, December 1994, pp. 668–692.

[8] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," IEEE Transactions on Computers, Vol. C-30, No. 7, July 1981, pp. 478–490.

[9] S. Gray and R. Adams, "Using Conditional Execution to Exploit Instruction Level Concurrency," Technical Report no. 181, School of Information Sciences, Division of Computer Science, University of Hertfordshire, March 1994.

[10] J. K. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," IEEE Computer, Vol. 17, No. 1, January 1984, pp. 6–22.

[11] M. Johnson, "Superscalar Microprocessor Design," Prentice Hall, 1991.

[12] D. Landskov, S. Davidson, B. Shriver, e P. W. Mallet, "Local Microcode Compaction Techniques," Computing Surveys, Vol.12, No.3, Sept.1980, pp.261-294.

[13] David J. Lilja, "Reducing the Branch Penalty in Pipelined Processors," IEEE Computer, Vol. 21, No. 7, July 1988, pp. 47–55.

[14] F. H. McMahon, "Fortran Kernels: MFLOPS," Lawrence Livermore National Laboratory,1983.

[15] Leonardo Cardoso Monteiro , "Geração de Código Paralelo para Máquinas Super Escalares," Projeto de Fim de Curso, IM, UFRJ, 1993.

[16] João Francisco Pereira Neto, "A Implementação de um Compactador Condicional," Projeto de Fim de Curso, DCC, IM, UFF, 1995.

[17] A. Nicolau and R. Potasman, "Realistic Scheduling: Compaction for Pipelined Architectures," Proceedings of the 23rd Annual International Workshop on Microprogramming and Microarchitecture MICRO-23, ACM and IEEE Computer Society, November 1990, pp. 69–79.

[18] D. N. Pnevmatikatos and G. S. Sohi, "Guarded Execution and Branch Prediction in Dynamic ILP Processors," Proceedings of the 21st Annual ISCA, 1994, pp. 120–129.

[19] Anna Dolejsi Santos, "Efeito da Execução Condicional em Arquiteturas Paralelas," Tese de Doutorado, COPPE/UFRJ, Programa de Engenharia de Sistemas e Computação, 1994.

[20] Anna Dolejsi Santos e Edil S. T. Fernandes, "Extração do Paralelismo em Arquiteturas com Capacidade de Execução Condicional," VII SBAC-PAD, 1995, pp. 77–99.

[21] J. E. Smith, "A Study of Branch Prediction Strategies," The 8th Annual International Symposium on Computer Architecture, 1981, pp. 135–148.

[22] F. L. Steven, G. B. Steven and L. Wang, "An Evaluation of the iHARP Multiple Instruction Issue Processor," Euromicro 94, September,1994.

[23] Daniel Tabak, "Advanced Microprocessors," Mc Graw-Hill, Inc., USA, 1991.

[24] M. Tokoro, T. Takizuka, E. Tamura, and I. Yamaura, "A Technique of Global Optimization of Microprograms," Proceedings of the 11th Annual Microprogramming Workshop, 1978, pp. 41–50.

[25] N. Wirth, "Algorithms+Data Structures=Programs," Prentice-Hall, Inc., 1976.