Compactação Local de Código para um SPARC Superescalar

Mario João Jr.
junior@nce.ufrj.br

Júlio S. Aude
salek@nce.ufrj.br

NCE/UFRJ
Caixa Postal 2324
Rio de Janeiro - RJ - 20001-970

Resumo

O uso de arquiteturas superescalares em microprocessadores RISC vem se manifestando como uma clara tendência nos últimos anos para obtenção de maior desempenho. Este trabalho analisa alternativas de implementação de uma arquitetura SPARC VLIW. Através de experiências de simulação, é medido o efeito produzido pela compactação local de código utilizado é descrito em detalhe nas suas diversas fases: determinação dos blocos básicos, construção dos grafos de dependência direta, renomeação de registradores e compactação propriamente dita com o uso do algoritmo *List Scheduling*. Os resultados obtidos com o método de compactação de código proposto aplicado a programas do *benchmark* SPEC indicam que a existência de unidades funcionais trabalhando em paralelo e a duplicação da ALU podem produzir reduções no tempo de execução de código da ordem de 50%.

Abstract

The use of superscalar architetures within RISC microprocessors is a clear trend in the last years for achieving higher performance. This work analyses alternatives for the implementation of a SPARC VLIW architecture. Simulation experiments are used to measure the effect produced by local code compaction when a SPARC architecture with multiple functional units is considered. The adopted local code compaction approach is described in detail throughout its several phases: the determination of basic blocks, the construction od direct dependence graph (DDG), the renaming of registers and the compaction itself based on the List Scheduling algorithm. The results produced by the proposed code compaction approach applied to programs of the SPEC benchmark show that the use of mutiple functional units working in parallel and the duplication of the ALU can yield code execution time reductions of the order of 50%.

1. Introdução

A utilização de microprocessadores superescalares vem se tornando muito freqüente na busca de maior capacidade de processamento em máquinas de alto desempenho. As arquiteturas superescalares possuem mais de uma unidade funcional e são capazes de despachar mais de uma instrução de máquina por ciclo.

Esta última característica é fonte de grande pesquisa, pois a escolha de quais instruções podem ser despachadas requer uma análise das dependências de dados que a instrução possa ter e da disponibilidade das unidades funcionais, já que uma instrução só pode começar a ser executada quando os resultados que ela necessita estiverem prontos e quando houver uma unidade funcional disponível para sua execução.

A literatura mostra duas tendências para solucionar o problema do despacho em máquinas superescalares: utilização de algoritmos em *hardware* ou utilização de compiladores especiais.

Na primeira solução o *hardware* possui um algoritmo (e.g. Tomasulo **[Tomasulo67]**) capaz de escolher a instrução que deve ser despachada e onde será executada. Para isso, o *hardware* deve fazer um controle das dependências de dados e da ocupação das unidades funcionais. Com isso, a máquina se torna mais complexa e, por conseguinte, mais cara.

Na segunda solução o *hardware* é capaz de despachar um determinado número de instruções por ciclo, uma para cada unidade funcional, e cabe ao compilador gerar o código eficientemente, de forma que cada unidade funcional execute uma instrução cujos dados já estão disponíveis. Estas arquiteturas, por possuírem uma palavra de memória que contém um número suficiente de campos para exercer controle sobre todas as unidades funcionais, receberam o nome de **máquinas VLIW** (*Very Large Instruction Word*).

O presente trabalho se relaciona com este último tipo de arquitetura. Seu objetivo é transformar o código objeto do processador SPARC seqüencial para uma máquina VLIW e, a partir do novo código, observar a eficiência da transformação. Optou-se por realizar a transformação diretamente no código objeto, pois, apesar de ser mais trabalhosa devido à decodificação, torna o programa independente da disponibilidade do código fonte.

A escolha da arquitetura SPARC deveu-se ao envolvimento, já de alguns anos, do grupo atuante no projeto MULTIPLUS **[Aude94]**, em desenvolvimento no NCE/UFRJ, com este tipo de arquitetura. Além dela estar sendo usada nos elementos de processamento do multiprocessador MULTIPLUS, está sendo desenvolvido também, em tecnologia CMOS 1.2, o projeto do NCESPARC **[Barbosa90]** que é uma implementação da arquitetura SPARC versão 7. O estudo de alternativas para implementação de arquiteturas SPARC superescalares é uma evolução natural da linha de pesquisa que conduziu ao desenvolvimento do NCESPARC.

Primeiramente, este trabalho abordará algumas características da arquitetura SPARC. A próxima seção tratará da separação do código objeto em blocos básicos. A quarta seção irá mostrar a construção dos DDG's para os blocos básicos. A seguir, a quinta seção abordará a técnica da renomeação de registradores. A sexta seção apresentará o algoritmo para a compactação local do código propriamente dito. Por último, é apresentada a forma de avaliação, bem como os resultados obtidos para alguns dos programas do *benchmark* SPEC.

2. Algumas Características da Arquitetura SPARC

A arquitetura SPARC **[Catanzaro94]** segue a filosofia RISC e é uma arquitetura do tipo LOAD/STORE, ou seja, todos os dados são trazidos (ou levados) da memória através destas instruções para depois serem manuseados. Logo, não existem instruções que somam um registrador com o conteúdo de uma posição de memória, por exemplo.

Alguns aspectos da arquitetura SPARC devem ser mencionados para facilitar a compreensão das soluções particulares tomadas para os problemas apresentados no decorrer deste trabalho.

Primeiramente, pode-se distinguir na arquitetura SPARC, cinco tipos de unidades funcionais. São elas:

- ALU: responsável pelas instruções aritméticas e lógicas.
- Deslocador, responsável pelas instruções de *shift*.
- Acesso à memória, responsável pela execução das instruções de *LOAD* e *STORE*.
- Desvio, responsável pelas instruções de *branch* (condicional ou não), e pelas chamadas a sub-rotinas.
- FPU, responsável pelas intruções de ponto flutuante.

As instruções na arquitetura SPARC possuem tempo de execução médio de 1 ciclo. Isto só é possível devido a esta arquitetura utilizar a técnica de *pipeline* com *interlock*, que permite não só que uma instrução seja despachada por ciclo, como também torna desnecessária a inserção de uma instrução nula (NOP) entre duas instruções seguidas onde o resultado da primeira é operando da segunda.

Entretanto, algumas instruções levam mais de um ciclo para serem executadas. É o caso das instruções de desvio e de acesso a memória. Estas instruções causam uma dificuldade adicional na compactação mostrada na seção 6.

Oútra característica a ser analisada, é o uso do *delayed branch*. Na arquitetura SPARC, todas as instruções de desvio são deste tipo, o que significa que a instrução imediatamente após um desvio é executada antes deste se efetuar. Por exemplo:

```
1)  LOAD [R1 + R2], R3
2)  ADD R3, 10, R4
3)  JUMP 6
4)  SUB R5, R6, R5
5)  ADD R3, 100, R2
6)  OR R3, 0, R5
```

Se o conjunto de instruções acima fosse executado em uma arquitetura que usasse um esquema normal de desvios, sua ordem de execução seria: 1, 2, 3 e 6. Em uma arquitetura utilizando *delayed branch*, sua ordem seria: 1, 2, 3, 4, 6.

Uma última característica a ser ressaltada é a utilização de janelas de registradores. Esta técnica consiste em dividir os registradores em blocos (janelas) sobrepostos, de forma que os últimos registradores de uma janela (registradores *out*), são os primeiros da próxima janela (registradores *in*). Estes registradores sobrepostos são usados como um mecanismo alternativo para a passagem de parâmetros para uma função, bem como para guardar o retorno desta função.

Além dos oito registradores *out* e dois oito registradores *in*, cada janela possui oito registradores locais. E existem ainda oito registradores globais de uso geral.

3. Criação de Blocos Básicos

A paralelização de um programa é um problema NP-completo. Para isso, buscam-se técnicas para torná-lo computacionalmente viável. Observou-se então, que se o problema fosse reduzido a pequenas porções de código, estes poderiam ser paralelizados. Para isso criou-se o conceito de **Bloco Básico**.

Blocos básicos são trechos seqüenciais de código, onde só existe um ponto de entrada (sua primeira instrução) e não mais de uma saída (possivelmente sua última instrução).

Com esta definição pode-se garantir que, uma vez iniciado o bloco básico, todas as instruções do mesmo serão executadas seqüencialmente até o seu final. Isto garante que qualquer alteração na ordem das instruções não irá afetar o resto do programa, desde que a semântica do bloco básico seja mantida.

Com o uso desta definição torna-se bastante óbvio uma forma de dividir um trecho de código em blocos básicos, pois a única maneira de um trecho de código deixar de ser executado seqüencialmente é a presença de uma instrução de desvio.

Portanto, o algoritmo deve percorrer o código acrescentando as instruções a um bloco básico. Ao encontrar uma instrução de desvio, o bloco básico é encerrado, e um novo bloco é criado.

A partir da descrição acima, segue-se um primeiro algoritmo para separação de código em blocos básicos:

```
Inicializar primeiro bloco básico;

para cada instrução faça
     se instrução for desvio então
          encerra bloco básico atual;
          inicializa um novo bloco básico;
     senão
          insere instrução no bloco básico atual;
fim
```

O algoritmo acima funciona em quase todos os casos, mas uma dificuldade adicional é apresentada: os desvios para o interior de um bloco básico. Como, pela definição de bloco básico, só existe um único ponto de entrada, caso haja um desvio para o interior de um deles, este deixaria de ser um bloco básico.

Para solucionar este problema, fazem-se necessárias algumas mudanças no algoritmo apresentado anteriormente. A primeira delas é o armazenamento do destino das instruções de desvio. A outra mudança é acrescentar uma outra fase no algoritmo, na qual os desvios armazenados são analisados e, caso pertençam ao interior de algum bloco básico, este é dividido em dois, garantindo assim, um único ponto de entrada para cada bloco básico. O algoritmo passaria a ser então:

```
Inicializar primeiro bloco básico;

para cada instrução faça
     se instrução for desvio então
          encerra bloco básico atual;
          inicializa um novo bloco básico;
          guarda endereço destino;
     senão
          insere instrução no bloco básico atual;
fim

para cada bloco básico faça
     Se houver algum endereço armazenado que pertença ao
     interior do bloco básico então
          dividir o bloco básico no endereço armazenado;
fim
```

Após esta fase, o código inicial se torna um conjunto de blocos básicos, no qual as demais fases da otimização irão atuar.

4. Construção dos Grafos de Dependências

Tendo separado o código objeto inicial em blocos básicos, a tarefa seguinte é a criação dos Grafos de Dependências Diretas (*Direct Dependence Graph*) para cada bloco. Estes grafos estabelecem uma ordem parcial entre as instruções, o que faz com que uma instrução só seja executada após suas anteriores.

Esta ordem parcial é estabelecida pelo fluxo de dados durante a execução do programa, ou seja, pela necessidade que uma instrução tem de ter seus operandos prontos antes de sua execução.

O restante desta seção apresenta algumas definições necessárias para a compreensão dos DDG's e o algoritmo para a construção dos mesmos.

4.1. Definições

Antes de definir as dependências, faz-se necessária a definição de alguns termos utilizados:

- **Out (i)**: É o conjunto de valores gerados pela instrução i. Este conjunto é sempre unitário ou vazio;
- **In (i)**: É o conjunto de valores utilizados pela instrução i.

Segue-se então as definições das dependências propriamente ditas:

- **Dependência direta** - Também conhecida como dependência verdadeira. Uma instrução (i) tem dependência direta de outra (j) se:

$$\mathbf{Out\ (i)} \cap \mathbf{In(j)} \neq \varnothing$$

- **Anti-Dependência** - Uma instrução (i) é anti-dependente de uma instrução (j) se:

$$\mathbf{In\ (i)} \cap \mathbf{Out(j)} \neq \varnothing$$

- **Dependência de saída** - Uma instrução (i) tem dependência de saída de outra instrução (j) se:

$$\mathbf{Out\ (i)} \cap \mathbf{Out(j)} \neq \varnothing$$

No grafo de dependências, cada nó representa uma instrução, e cada aresta representa uma dependência (S - dependência de saída, D - dependência direta e A - anti-dependência). Com isso, para o exemplo abaixo, teríamos o grafo de dependências da figura 3.1.

1) A = B + C
2) D = A
3) B = E + F
4) D = B + A

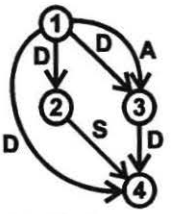

Figura 3.1: Grafo de Dependências

O grafo acima é um grafo de dependências, e não um DDG, pois existem dependências indiretas. Estas dependências se caracterizam pela existência de mais de um caminho entre dois nós do grafo.

Para eliminar as dependências indiretas, basta, ao inserir um nó no grafo, testar se o nó dependente de outro não é seu descendente. Para ilustrar, suponha a inserção do nó 4 do exemplo anterior. Como o nó 4 é descendente do nó 1, então a aresta (1, 4) não existiria.

Outra providência a ser tomada para reduzir o número de arestas desnecessárias é a inserção de apenas uma única aresta entre dois nós, quando existe mais de um tipo de dependência entre eles. Neste caso uma ordem de precedência tem que ser assumida. No caso da implementação apresentada neste trabalho, a ordem é: dependências diretas, anti-dependências e dependências de saída.

Após as duas observações acima, o grafo da figura 3.1 passa a ser um DDG ilustrado na figura 3.2.

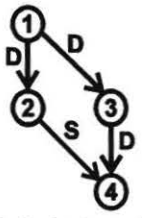

Figura 3.2: Grafo de Dependências Diretas

4.2 Algoritmo de Construção de DDG

O algoritmo para construção dos DDG's é bastante simples, mas sua eficiência deixa um pouco a desejar, já que para a construção de um DDG, as dependências de uma instrução têm de ser testadas para cada instrução precedente dentro do bloco básico.

O algoritmo teria então complexidade $O(n^2)$. Para melhorar um pouco o tempo de execução, pode-se utilizar o fato de que um DDG não possui dependências indiretas. Isto faz com que a complexidade não mude, mas melhore o tempo de execução, pois se uma instrução depende de outra, aquela depende também dos ancestrais desta, evitando

assim o teste de dependência. Para ilustrar, considere o exemplo apresentado na subseção anterior. A inserção da aresta (2,4), significa que o nó 4 depende não só do nó 2 como do nó 1, já que o nó 2 depende do nó 1.

Segue-se então o algoritmo propriamente dito.

```
para cada bloco básico faça
     i = última instrução do bloco básico;

     enquanto i >= primeira instrução do bloco básico  faça
          j = i - 1;
          enquanto j >= primeira instrução do bloco
          básico faça
               se instrução i não depender de instrução j
               então
                    verifica dependências diretas;

                    se houver dependência direta então
                         instrução i depende dos
                         ancestrais de j;
                         j = j + 1;
                         continua enquanto;
                    fim
               fim
               se instrução i não depender de instrução j
               então
                    verifica anti-dependências;

                    se houver anti-dependência
                    então
                         instrução i depende dos
                         ancestrais de j;
                         j = j + 1;
                         continua enquanto;
                    fim
               fim
               se instrução i não depender de instrução j
               então
                    verifica dependências de saída;

                    se houver dependência de saída então
                         instrução i depende dos
                         ancestrais de j;
                         j = j + 1;
                         continua enquanto;
                    fim
               fim
               j = j - 1;
          fim
          i = i - 1;
     fim
fim
```

O algoritmo acima representa exatamente o que foi descrito. A única exceção é a verificação das dependências, que, além de analisar as dependências, constrói o DDG.

5. Renomeação de Registradores

Dos três tipos de dependências apresentadas na seção anterior, duas delas (anti-dependências e dependências de saída) são consideradas dependências falsas, pois não existe a necessidade de um valor gerado, e sim um conflito no uso dos registradores.

Isto ocorre porque as máquinas têm um número limitado de registradores, o que faz com que os algoritmos para geração de código sejam obrigados a reutilizar ao máximo estes registradores, mesmo que para funções diferentes.

No caso de uma máquina superescalar, esta reutilização é bastante prejudicial, pois restringe o paralelismo.

Uma técnica para minimizar este problema é a chamada renomeação de registradores (*registers rename*), que consiste em avaliar quais os registradores "sem função", em um determinado ponto do programa, e utilizá-los para substituir aqueles que causam as dependências falsas.

Para poder aplicar a renomeação, o otimizador de código deve conhecer o fluxo de execução do programa, para, então, poder avaliar quais registradores podem ser usados e, quando possível, realizar a substituição.

5.1. Criação do Grafo de Fluxo

O fluxo de execução de um programa pode ser representado através de um grafo, onde cada nó representa um bloco básico e as arestas representam a ordem com que os blocos básicos podem ser executados.

O grau de saída de um nó depende da forma como um bloco básico termina. Este pode terminar por:

- uma chamada de função, neste caso, após a execução da função, o programa continua a ser executado a partir do bloco básico seqüencialmente após o bloco corrente;
- um desvio incondicional, onde a execução continua a partir do bloco alvo do desvio;
- uma instrução que não seja de controle de fluxo, sendo a execução seguida seqüencialmente;
- um desvio condicional, o que traz duas possibilidades: ou o programa continua a partir do bloco seqüencialmente posterior, ou continua a partir do bloco alvo do desvio.

Analisando as possibilidades acima, tem-se que um nó pode ter grau de saída um ou dois.

O grafo de fluxo acima descrito representa não um programa, mas uma função. Isto se deve à estrutura de janelas da arquitetura SPARC, pois, a cada nova função, uma nova janela é utilizada, não acessando assim os registradores da janela anterior. Quanto aos registradores globais, a abordagem adotada foi não realizar renomeação nestes registradores, mesmo porque estes são raramente acessados.

Segue, então, o algoritmo para construção do grafo de fluxo:

```
Para cada função faça
     Para cada bloco básico faça
          analisar a forma como o bloco termina;
          inserir as arestas necessárias;
     fim
fim
```

5.2. Algoritmo para Renomeação

Tendo o grafo de fluxo, para que se possa fazer a renomeação, é necessário saber quais são os registradores não utilizados. Para isso tem-se que percorrer o grafo de fluxo e descobrir, para cada bloco básico, os registradores ativos. Isto é feito aplicando a seguinte forma para cada bloco básico:

$$ativos\ (B) = (ativos\ (prox\ (B)) \cup gen\ (B)) - kill\ (B)$$

Onde:

- *prox (B)* é o conjunto formado por todos os blocos básicos B_i tais que exista uma aresta (B, B_i);
- *gen (B)* é o conjunto formado por todos os registradores usados como operandos cujos valores não são calculados no bloco básico;
- *kill (B)* é o conjunto formado por todos os registradores cujos valores são calculados no bloco básico.

Os registradores usados para renomeação são os registradores locais, pois é garantido que estes só têm seus valores mantidos enquanto estiver sendo executada a função da qual o bloco básico pertence.

Devido a estrutura de janelas de registradores, nem sempre é possível realizar a renomeação, mesmo que existam registradores disponíveis. Pois imediatamente após uma chamada a sub-rotina, os registradores *out*, contém possíveis valores retornados pela função e não podem ser renomeados, o que não pode acontecer também antes de uma chamada de sub-rotina, pois neste caso os registradores *out* contém parâmetros a serem passados.

Estas limitações são o grande motivo pelo qual, nem sempre, são eliminadas as anti-dependências.

6. Compactação Local de Código

Compactar localmente o código nada mais é do que rearrumar as instruções em grupos de forma que as instruções de cada grupo não possuam conflitos de recursos. Para tal existem algoritmos que foram apresentados e analisados em **[Landskov80]**. Dentre estes algoritmos, foi escolhido o de Lista de Escalonamento (*List Scheduling*) por sua eficiência e facilidade de implementação.

Abaixo segue o algoritmo de Lista de Escalonamento.

```
para cada bloco básico faça
     insere instruções sem dependências na lista de
     instruções prontas;

     inicializa primeira instrução longa;

     enquanto houver instruções na lista faça
          para cada instrução da lista faça
               se houver unidade funcional disponível
               então
                    aloca instrução na instrução larga
                    atual;
                    retira instrução da lista e do DDG;
               fim
          fim

          insere instruções sem dependências na lista;
          aloca mais uma instrução larga;
     fim
fim
```

Uma observação deve ser feita a respeito da inserção na lista de instruções sem dependências. Esta segue as orientações descritas em **[Landskov80]**, ou seja, a lista é ordenada de acordo com o número de descendentes de cada nó.

Como descrito na Seção 2, algumas instruções levam mais de um ciclo para executar. Neste caso, ao invés de ser retirada da lista de instruções prontas na primeira vez, estas permanecem até que o número de instruções longas necessárias para sua execução seja gerado.

No caso de uma instrução de *branch*, alguns cuidados adicionais devem ser tomados. Com a existência de uma unidade funcional com este propósito, estas instruções são alocadas paralelamente a outras instruções duas palavras anteriores, desde que estas instruções não influam na condição do desvio (se este for condicional). Caso esta alocação não seja possível, é inserida uma palavra nula após aquela contendo o desvio.

7. Método de Avaliação e Resultados

O método de avaliação empregado para medir a eficiência da otimização local de código seqüencial para uma arquitetura SPARC foi a avaliação dinâmica, na qual o programa a ser analisado é executado, com pontos de parada (*breakpoints*) no início de cada bloco básico. Como uma vez iniciado o bloco básico todas suas instruções são executadas, basta armazenar o número de vezes que o programa passou por cada bloco.

Ao final da execução do programa, o algoritmo de compactação é executado e, para cada bloco básico, são anotados o número de ciclos na execução seqüencial e na otimizada. Para calcular o número de ciclos total, o número de ciclos de cada bloco é multiplicado pelo número de vezes que este foi executado. Com isto, obtem-se a redução no tempo de execução.

Para comprovar a eficiência da compactação local de código, foram utilizados três dos programas que compõem o *benchmark* SPEC, sendo eles **compress, eqntott** e **espresso**.

Buscando uma maior otimização, variou-se o número de unidades funcionais descritas na Seção 2. Por se tratar de um *benchmark* inteiro, o número de FPU's não alterou a redução no tempo de execução. Observou-se também que o número de deslocadores pouco alterou a redução (cerca de 0,1%).

Seguem tabelas indicando a redução no tempo de execução conforme a variação no número de ALU's e de unidades de acesso a memória.

Compress:

Porcentagem de redução (número de ALU's por número de unidades de acesso a memória)

Sem Renomeação

	1	2	3
1	35.56	42.09	43.58
2	39.38	46.76	48.04

Com Renomeação

	1	2	3
1	36.41	42.94	44.43
2	43.21	50.59	51.87

Eqntott:

Porcentagem de redução (número de ALU's por número de unidades de acesso a memória)

Sem Renomeação

	1	2	3
1	44.27	46.88	46.97
2	45.01	47.84	47.97

Com Renomeação

	1	2	3
1	44.62	47.23	47.32
2	45.40	48.19	48.32

Espresso:

Porcentagem de redução (número de ALU's por número de unidades de acesso a memória)

Sem Renomeação

	1	2	3
1	35.30	37.72	38.29
2	40.84	43.50	44.08

Com Renomeação

	1	2	3
1	35.92	38.39	38.96
2	41.65	44.36	44.95

Para o melhor desempenho obtido (3 ALU's e 2 unidades de acesso a memória) a porcentagem de tempo que as unidades executam alguma instrução útil é:

	Compress	Eqntott	Espresso
Desvio	25.93	33.31	23.59
Acesso a memória	58.99	60.20	60.82
Acesso a memória	22.78	2.78	20.63
ALU	59.62	86.51	51.39
ALU	22.17	7.76	8.84
ALU	5.05	1.35	1.99
Deslocamento	11.54	1.01	12.90
FPU	0.0	0.0	0.0

Este resultados indicam que a melhor solução, tendo em vista o compromisso custo/benefício, seria a utilização de uma arquitetura onde apenas a ALU seria duplicada.

8. Considerações Finais

O presente trabalhou apresentou um algoritmo de compactação local de código para uma arquitetura SPARC superescalar. Após sua aplicação, observou-se que apenas o número de ALU's deveria ser duplicado.

Algumas técnicas poderiam ser utilizadas para melhorar a compactação, como é o caso da compactação global e seus diversos algoritmos. Alterações na arquitetura também poderiam ser feitas para que a redução do tempo de execução aumentasse.

9. Agradecimentos

Os autores agradecem à FINEP e ao CNPq/RHAE pelo apoio dado, sem o qual este trabalho não poderia ter sido realizado.

10. Referências Bibliográficas

[Aude94] Aude, J. S., "Multiplus/Mulplix: An Integrated Enviroment for the Development of Parallel Applications", Procedings of the IEEE/USP International Workshop on High Performance Computing - WHPC'94, March 1994, pp. 245-255

[Barbosa90] Barbosa M. A. S. et al., "Implementação de Mmicroprocessador RISC com arquitetura SPARC", Anais do V Simpósio Brasileiro de Concepção de Circuitos Integrados (SBCCI), outubro 1990, pp. 121-131

[Catanzaro94] Catanzaro, B., "Multiprocessor System Achitectures", Sun Microsystems - Prentice-Hall, 1994

[Landskov80] Landskov D., Davidson S., Shriver B., and Mallet P., "Local Microcode Compactation Techiniques", Computing Surveys, Vol. 12, No. 3, September 1980, pp. 261-294

[Tomasulo67] Tomasulo R. M., "An Efficient Algorithm for Exploring Multiple Arithmetic Units", IBM Journal os Research and Development, January 1967, pp.339