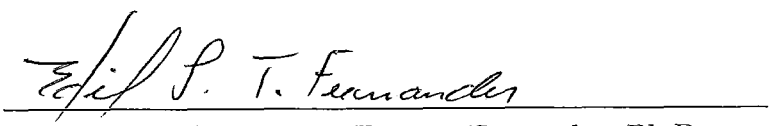
EFEITO DA REORGANIZAÇÃO DO CÓDIGO BINÁRIO NO DESEMPENHO DE *CACHE* DE INSTRUÇÕES EM ARQUITETURAS RISC

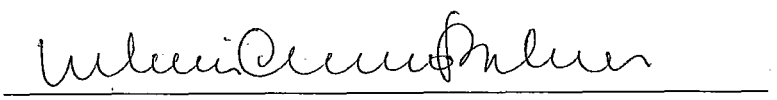
Ricardo Gonçalves Quintão

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

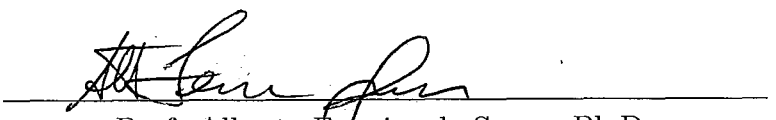
Aprovada por:



Prof. Edil Severiano Tavares Fernandes, Ph.D.



Prof. Valmir Carneiro Barbosa, Ph.D.



Prof. Alberto Ferreira de Souza, Ph.D.

RIO DE JANEIRO, RJ - BRASIL

MARÇO DE 2004

QUINTÃO, RICARDO GONÇALVES

Efeito da Reorganização do Código Binário no Desempenho de *Cache* de Instruções em Arquiteturas RISC [Rio de Janeiro] 2004

XVII, 74 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2004)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1 - Análise do Desempenho de Memórias *Cache* de Instrução

I. COPPE/UFRJ II. Título (série)

À minha filha Lorena Quintão....

Agradecimentos

Quando ingressei para o Mestrado em informática na Coppe, percebi que iria realizar o meu sonho em ser um profissional e pesquisador nesta área. A minha graduação foi em Física, o que trouxe uma certa dificuldade em algumas disciplinas e até mesmo no entendimento de alguns artigos.

Durante o mestrado, fiz novos amigos que muito contribuíram para a continuidade do curso. Agradeço ao, José Afonso, Tatiana Cavalcante, Sérgio Braúna, Luciana Ferrari, Eduardo Aquilar, Anderson Faustino, Patrícia Kayser, Marluce Rodrigues que me ajudaram nos momentos finais do mestrado, além da Paula Faragó, André Barbosa, Ana Paula, Thobias Trevisan, Leonardo Bidese, Rodrigo Fonseca e todos que me acompanharam durante estes quatro anos.

Agradeço aos meus amigos da UERJ, Eduardo Almeida por toda ajuda que me deu sobre o Linux que foi o Sistema Operacional básico utilizado na minha pesquisa, ao Maxmiliano, a Mileni Britto e a Josefina.

Um agradecimento especial a minha esposa Fátima Quintão que, entendendo as minhas dificuldades, abriu mão de muitas coisas para me apoiar e ajudar, possibilitando a conclusão desta pequisa.

Durante o mestrado, minha filha Lorena nasceu o que me levou a ter menos tempo para dedicar a tese. Agradeço a todos que me apoiaram durante esta época e ao meu orientador que entendeu as minhas dificuldades e me ajudou ao máximo possível, principalmente por ter passado por períodos de ausência devido a minha filha e a necessidade de aumentar a minha carga horária no trabalho.

Por fim agradeço aos meus pais e a minha irmã todo apoio que me deram para conseguir conquistar mais esta etapa na minha vida.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc)

EFEITO DA REORGANIZAÇÃO DO CÓDIGO BINÁRIO NO DESEMPENHO DE *CACHE* DE INSTRUÇÕES EM ARQUITETURAS RISC

Ricardo Gonçalves Quintão

Março/2004

Orientador: Edil Severiano Tavares Fernandes

Programa: Engenharia de Sistemas e Computação

Esta tese avalia o efeito provocado pela reorganização do código binário de arquiteturas do tipo RISC no desempenho da Memória *Cache* de Instruções. Para tal foi empregado o simulador funcional *sim-fast* do pacote *SimpleScalar* e criamos um modelo parametrizável de Memória *Cache* de Instruções (e respectivos mecanismos de gerenciamento) para simular os programas inteiros do conjunto SPEC95. Avaliamos as taxas de acertos (*hits*) e falhas (*misses*) de cada membro da família de *caches* de instruções produzidas por diferentes algoritmos de reordenação de código.

Definimos tipos específicos de Memória *Cache* de Instruções e de algoritmos de reorganização do código binário. Comparando o desempenho do binário original com o dos binários reorganizados, verificamos que uma simples mudança no *layout* do código binário pode oferecer um grande aumento no desempenho.

Ao reordenarmos os blocos básicos de um binário precisamos também ajustar o campo de endereço de cada instrução de desvio para que as instruções alvo sejam atingidas. Considerando que estamos manipulando código binário e que o respectivo programa fonte nem sempre está disponível, então desenvolvemos um esquema de mapeamento que, apesar de sua simplicidade, garante a execução precisa de código reordenado dispensando o ajuste exigido pelos métodos convencionais de otimização de código: compila–obtém perfil de execução–reordena–*link*.

Apesar do custo adicional desse esquema de mapeamento, ele permite a execução de binários que se reorganizam dinamicamente. Com o consistente aumento de desempenho apresentado por recentes máquinas, acreditamos que a reorganização dinâmica de binários seja viável e que o mecanismo de mapeamento aqui sugerido seja aperfeiçoado e devidamente incorporado no *hardware* de futuros processadores.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

THE EFFECT OF BINARY CODE REORDERING ON INSTRUCTION CACHE PERFORMANCE IN RISC ARCHITECTURES

Ricardo Gonçalves Quintão

March/2004

Advisor: Edil Severiano Tavares Fernandes

Department: Computing and Systems Engineering

This thesis evaluates memory cache instructions performance provoked by the binary code reorganization. For this we used the functional simulator sim-fast of the SimpleScalar package and created a parameterized model of instruction Cache Memory (and respective mechanisms of management) to simulate the entire SPEC95 set programs. We evaluate the taxes of hits and misses in each member of the instructions cache family produced by different code reordering algorithms.

We define specific types of Instruction Cache Memory and reordering binary code algorithms. After compare the performance of the origin binary code with the reorganized binary, we verify that a simple change in the structure of the binary code can offer a great increase in the performance.

The simple basic blocks reordering of a binary does not allow it's execution: the address field of each branch instruction must be correctly adjusted that the target instruction are reached. Considering that we are manipulating binary code and that the respective source program is not always available, then we develop a mapping project that, although its simplicity, guarantees the right execution of rearranged code excusing the adjustment demanded for the conventional methods of code optimization: compile - gets execution profile - rearranges - link.

Despite the additional cost required by our mapping project, it thus makes possible the reordering of the binary basic blocks in execution time, becoming very useful in the execution of binary that are reorganized dynamically.

Considering the consistent increase in the processing capacity presented in recent machines, we believe that the topical binary reorganization would be more explored and that the mapping mechanism here suggested being improved and incorporated in the future processor hardware.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O modelo de Von Neumann estabelece que "para ser executado, o programa deve estar armazenado na memória". A memória é organizada em grupos de *bits* chamados células (ou palavras), onde as informações são armazenadas. Para acessá-las, foi criado um mecanismo de identificação de cada uma destas células denominado **endereço de memória**, através dele podemos acessar (ler ou gravar) qualquer palavra da Memória Principal.

Para que um programa seja executado, é necessário transferir suas instruções da memória para o interior da CPU. Existe na CPU um registrador apontando para a próxima instrução a ser executada. Este registrador é chamado de PC (*Program Counter*).

Os registradores também são palavras de uma memória especial localizada no interior da CPU (denominado memória local) e estão conectados às unidades funcionais da mesma. Eles atuam como operandos fonte e destino das operações realizadas pela CPU.

Há algum tempo atrás, tanto a CPU como as memórias tinham velocidades de trabalho muito próximas uma das outras [17]. Com o passar do tempo, tornou-se necessário memórias com maior capacidade de armazenamento motivando o aparecimento da Memória Dinâmica. Este tipo de memória permitiu aumentar a capacidade de armazenamento, mas limitou sua velocidade. Já na CPU, é utilizada uma tecnologia diferente que permite uma alta velocidade de processamento (tecnologia de Memória Estática). Com isso, a CPU foi se tornando cada vez mais rápida enquanto que as memórias não foram capazes de acompanhar tal aumento de velocidade. Como é necessário trazer informações da memória para os registradores, esta diferença de velocidade acabou se tornando um grande problema (denominado "*Memory Gap Problem*").

Para reduzir o efeito causado por esta diferença de velocidade, estudos sobre o comportamento dos programas foram realizados e destes estudos foram estabelecidos dois princípios: Localidade Temporal e Localidade Espacial [4, 5].

A Localidade Temporal caracteriza aqueles programas que apresentam uma grande quantidade de *loops* (repetição de trechos de códigos) e estabelece que **uma vez acessada uma determinada posição de memória, existe uma grande probabilidade desta posição ser novamente acessada em um curto intervalo de tempo**. Isto é, devido ao fato da grande maioria dos programas serem formados por *loops* e que muitos deles têm um alto grau de repetição, a CPU ficará executando um pequeno grupo de instruções durante longos períodos do tempo de processamento.

O princípio da Localidade Espacial, por outro lado, estabelece que **uma vez que uma determinada posição de memória é acessada, existe uma grande probabilidade de que as posições vizinhas também sejam acessadas**. Isto é, a execução de um programa é naturalmente seqüencial, a não ser que a instrução executada seja um desvio (transferência de controle), a próxima instrução estará no endereço adjacente. Como o programa pode ser formado por uma quantidade muito maior de outras instruções do que de transferências de controle, então existe uma grande probabilidade da próxima instrução estar no endereço adjacente.

Recentes estudos realizados na UFRJ verificaram que cerca de 50% das instruções do código binário nunca são utilizadas (vide [6] e [7]). No estudo conduzido em 1971 sobre o comportamento de programas, D. E. Knuth já havia verificado que 4% das instruções de programas FORTRAN eram responsáveis por 50% do tempo de execução [13].

A reduzida capacidade de armazenamento das memórias estimulou a criação de mais um nível na hierarquia do sistema de memória na década de 70. Este nível é formado por uma memória de alta velocidade, mas com pequena capacidade de armazenamento, que é conhecida por Memória *Cache*. Cópias de trechos da Memória Principal podem estar armazenados nesta Memória *Cache* e toda vez que a CPU fizer um acesso de leitura à memória, a *cache* é examinada antes. Se a cópia da palavra estiver na *cache*, dizemos que ocorreu um acerto (*cache hit*) e neste caso o tempo entre o pedido e o recebimento da informação é mais curto. Caso a informação solicitada não esteja na *cache*, dizemos que ocorreu uma falha (*cache miss*) e neste caso o tempo entre o pedido e o recebimento da informação será bem maior que o anterior, pois a palavra será buscada na Memória Principal que continua sendo de baixa velocidade.

Como foi mencionado no trabalho de Wilkes [17], as memórias utilizadas nos computadores sempre tiveram um desempenho inferior ao da CPU. Em 1970 surgiram as primeiras memórias de semicondutores facilitando a sua interface e tornando o seu desempenho próximo ao da CPU. O natural seria que as velocidades das memórias e das

CPUs evoluíssem na mesma taxa, mas não foi o ocorrido. Devido a diferente tecnologia utilizada nas memórias para conseguir uma alta densidade de *bits* (DRAM - Memórias Dinâmicas), tivemos uma perda na sua velocidade. Desde 1980, esta diferença de velocidades vem se agravando: a CPU tem seu desempenho aumentado por volta de 60% ao ano enquanto que as memórias DRAM aumentam por volta de 10% ao ano.

Para se ter uma noção da gravidade do problema (*the memory gap problem*), em uma estação Alpha 21264 de 500Mhz, uma falha na Memória *Cache* interrompe o processador por 128 ciclos. Para que a Memória *Cache* atinja uma alta velocidade, é utilizada a mesma tecnologia usada na fabricação da CPU (SRAM - Memórias Estáticas). Devido a uma restrição desta tecnologia, não é possível ter uma alta densidade de *bits*, o que resulta na pequena capacidade de armazenamento e na elevada taxa de falhas.

Foram propostos três algoritmos de reorganização que resultaram em uma queda média de até 43,62% na taxa de falhas, mostrando que estas técnicas podem resultar em bons ganhos de desempenho.

Esta tese está organizada em 7 capítulos. O Capítulo 2 apresenta os conceitos de Memória *Cache*. O Capítulo 3 resume os trabalhos relacionados a esta pesquisa. No Capítulo 4 é descrito o problema abordado, a metodologia proposta e o ambiente experimental utilizado. No Capítulo 5 temos as características dos experimentos realizados. A análise dos resultados obtidos é feita no Capítulo 6. Finalmente, no Capítulo 7, apresentamos as conclusões e sugestões para trabalhos futuros. Nos apêndices são apresentados todos os resultados obtidos nos experimentos.

Capítulo 2

A Memória *Cache* e suas Vantagens

A Memória *Cache* (também denominada "*cache*" externa) está localizada entre a CPU e a Memória Principal (MP) como mostra a Figura 2.1.

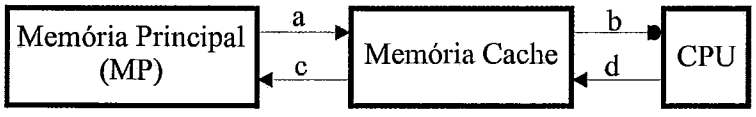


Figura 2.1: Sistema de Memória

A Memória *Cache* pode ser caracterizada pela:

- **Política de Mapeamento** (Estrutura da *Cache*): esta característica especifica como as palavras da Memória Principal são armazenadas na Memória *Cache*.
  - Mapeamento Direto
  - Puramente Associativo
  - Associativo por Conjunto
- **Política de Substituição**: escolhe a posição da Memória *Cache* cujo conteudo será substituído pelo de uma outra palavra da Memória Principal.
  - Aleatório
  - FIFO ("*First In First Out*")
  - LFU (Menos Freqüentemente Usado)
  - LRU (Menos Recentemente Usado)
- **Política de Atualização**: define quando os dados alterados pela CPU na Memória *Cache* serão transferidos para a Memória Principal.

- *Write Through*
- *Write Back*
- *Write Once*

2.1 Organização da Memória *Cache*

A *Cache* externa é formada por linhas (*frame*), cada uma capaz de armazenar uma certa quantidade de células da Memória Principal. Conforme ilustra a Figura 2.2, cada linha possui alguns *bits* de controle. O *bit* Válido(V) indica se o conteúdo da linha é válido e o *bit* Modificado(M) informa se ela sofreu alguma alteração. Existe também o campo TAG que contém parte do endereço (do bloco na Memória Principal) que foi mapeado na linha da *cache*. Para a Memória *cache* é como se a Memória Principal estivesse dividida em blocos com o mesmo número de células existentes em cada linha da *cache*.

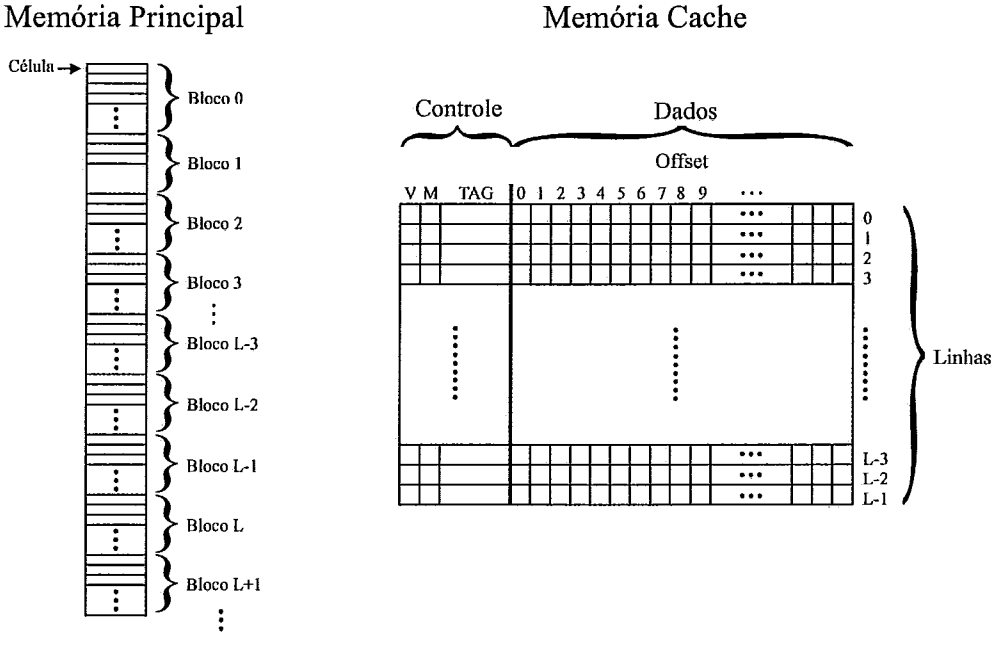


Figura 2.2: Organização Interna da Memória *Cache* e da Memória Principal

2.1.1 Memória *Cache* com Mapeamento Direto

Nesta organização, cada bloco da Memória Principal só pode ser carregado numa determinada linha da *Cache*, não sendo possível carregá-lo em uma outra linha. O Bloco "0" é carregado na linha "0", o bloco "1" na linha "1" e assim sucessivamente. Como a Memória Principal é muito maior que a Memória *Cache*, haverá muito mais blocos do que

linhas. Neste caso, quando o endereço do bloco de memória for maior que o número de linhas da *cache*, é calculado o resto da divisão entre o tamanho da *cache* e o endereço do bloco de memória. Veja a Figura 2.3

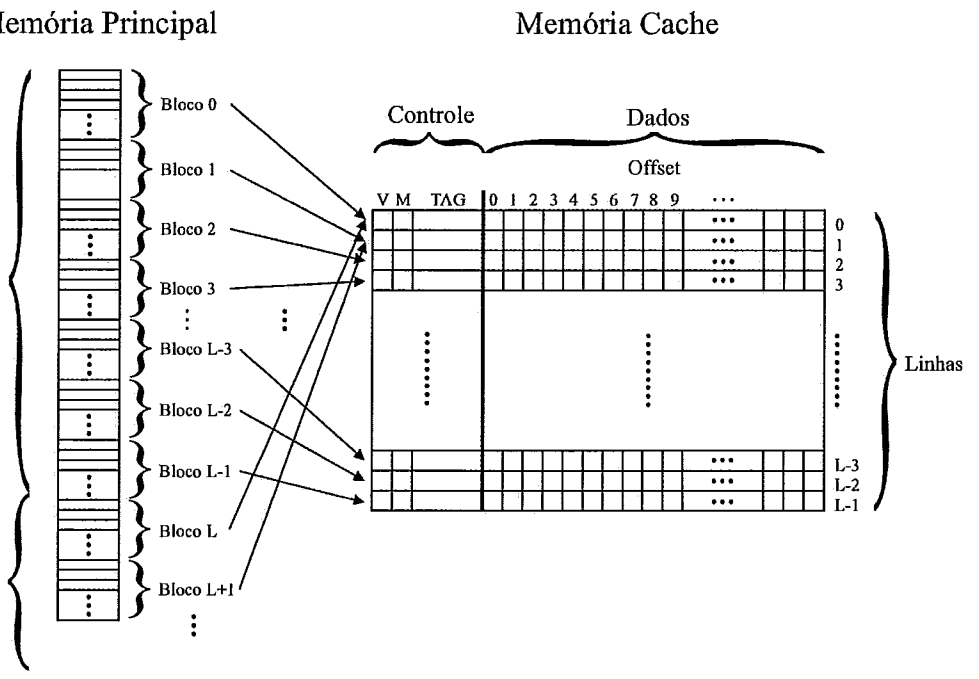


Figura 2.3: Memória *Cache* de Mapeamento Direto

Diferentes endereços de blocos de memória podem estar mapeados na mesma linha. Como saber de que região da memória pertence o conteúdo de uma determinada linha da *cache*? É aí que entra o TAG. Ele guarda os *bits* mais significativos do endereço permitindo que a *cache* identifique com exatidão de que lugar da Memória Principal pertence o conteúdo carregado naquela linha.

Normalmente, o carregamento do bloco na linha se dá sob demanda, isto é, conforme a necessidade. No momento que a CPU solicita o conteúdo de uma determinada célula da Memória Principal, a Memória *Cache* identifica qual é o número do bloco na Memória Principal e examina o TAG da linha correspondente. Se o TAG armazenado na linha coincidir com os *bits* mais significativos do endereço, teremos um acerto, e neste caso a Memória *Cache* fornece o seu conteúdo numa velocidade muito superior caso o acesso tivesse sido feito à Memória Principal. Agora, se o TAG não coincidir com os *bits* mais significativos do endereço, teremos uma falha, e neste caso a Memória *Cache* terá que carregar todo o bloco em questão para a respectiva linha de *cache*, atualizando o valor do TAG. Repare que neste caso aguardaremos o tempo necessário para acessar a Memória

Principal. Outra observação importante é que ao carregar todo o bloco na linha de *cache*, está sendo feita a suposição, baseando-se no conceito de localidade espacial e temporal, de que todos os dados deste bloco serão utilizados pela CPU. Caso a CPU não os utilize, teremos desperdiçado tempo e espaço de *cache*.

No Mapeamento Direto, se um processo referenciar repetidamente endereços de memória que são mapeados na mesma linha, porém pertencentes a localizações diferentes, haverá uma grande ocorrência de falhas forçando ao repetido carregamento destes dados, provocando uma grande queda no desempenho do processador. Este problema é conhecido como *Ping-Pong*. Por exemplo: suponha que dentro de um determinado *looping*, existam chamadas a várias funções ou procedimentos do programa. Se houver a coincidência de duas ou mais destas funções ou procedimentos estarem mapeadas na mesma linha da Memória *Cache*, haverá a ocorrência de diversas falhas devido à alternância entre estas funções ou procedimentos.

Na política de mapeamento direto, o endereço do programa é dividido em três partes: deslocamento dentro da linha (*Offset*), número da linha e TAG. O número da linha junto com o valor do TAG formam o endereço na Memória Principal de um bloco de células. Veja a Figura 2.4.

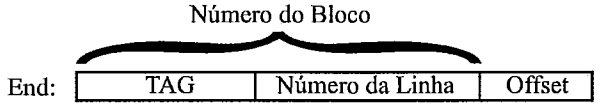


Figura 2.4: Formato do Endereço da Memória Principal Visto por uma Memória *Cache* com Mapeamento Direto

2.1.2 Memória *Cache* Puramente Associativa

Neste tipo de *cache*, o bloco da Memória Principal pode ser carregado em qualquer uma das linhas existentes na Memória *Cache* (veja a Figura 2.5). Esta versatilidade resolve o problema de endereços conflitantes mencionado na Memória *Cache* de Mapeamento Direto. Sendo assim, a variedade de blocos da Memória Principal que podem vir a ser carregados em uma determinada linha é bem maior que a da *cache* de Mapeamento Direto, necessitando então de um TAG com mais *bits*. Enquanto no Mapeamento Direto o TAG armazenava os *bits* mais significativos, neste caso, o TAG irá armazenar o endereço do bloco, já que pode ser carregado em uma determinada linha qualquer bloco da Memória Principal.

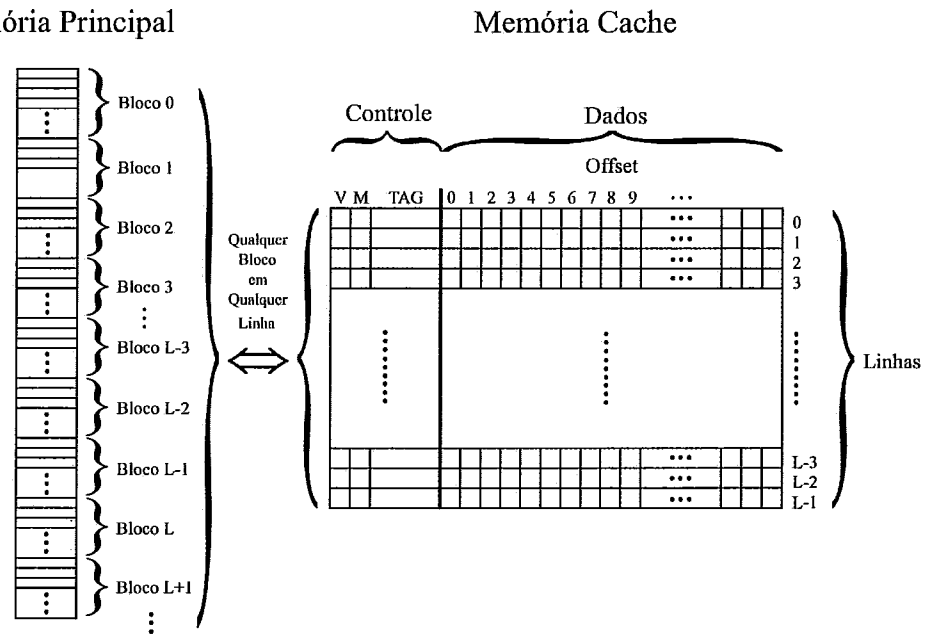


Figura 2.5: Memória *Cache* Puramente Associativa

O carregamento do bloco da memória Principal na linha da *cache* se dá sob demanda. Quando a CPU solicita o conteúdo de uma determinada célula da Memória Principal, a Memória *Cache* verifica se algum TAG já contém o endereço. Como o bloco pode estar carregado em qualquer linha, a Memória *Cache* (que é do tipo associativo) compara o endereço do bloco, com os TAGs de todas as linhas simultaneamente. Caso ocorra alguma coincidência, então teremos um acerto, e neste caso a *cache* fornece o conteúdo correspondente numa velocidade muito superior ao da Memória Principal. Contudo, se nenhum TAG contiver o endereço do bloco, então teremos uma falha, forçando a Memória *Cache* a carregar este bloco em uma de suas linhas. Diferente do Mapeamento Direto esta *cache* possui uma vasta opção de linhas para carregar o bloco, sendo então necessário estabelecer um método de escolha da linha. É aí que foram criadas as políticas de substituição.

No mapeamento Puramente Associativo, o endereço da Memória Principal é dividido em dois campos: deslocamento dentro da linha (*Offset*) e o TAG. Veja a Figura 2.6.

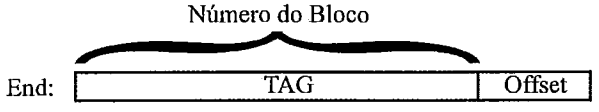


Figura 2.6: Formato do Endereço da Memória Principal Visto por uma Memória *Cache* Puramente Associativa

2.1.3 Políticas de Substituição

As políticas de substituição têm por objetivo escolher qual das linhas cujo conteúdo será descartado para que um outro bloco seja carregado. A esta linha damos o nome de Linha Vítima. Como na maioria das vezes, fazer o carregamento de um bloco implica no descarregamento das informações referentes à linha vítima, então este método de escolha irá influenciar no desempenho da Memória *Cache*. A seguir temos quatro métodos de escolha da linha vítima.

- Aleatório: Este método faz uma escolha casual de uma linha para ser a linha vítima. Como este método não leva em consideração os acessos anteriores à Memória *Cache*, existirá uma grande chance de se fazer uma má escolha.
- FIFO: Este método faz a escolha da linha vítima de acordo com a ordem de carregamento (*First-in First-out*). O método de escolha é simples e é levado em consideração a ordem de carregamento, que só é atualizado quando a linha for carregada.
- LFU (Menos Freqüentemente Usada): Este método usa como base para a escolha da linha vítima a quantidade de vezes que as linhas foram referenciadas. A linha que foi referenciada menos vezes, é a linha vítima. Este método usa como critério a freqüência de acessos a uma determinada linha.
- LRU (Menos Recentemente Usada): O critério de escolha deste método é a ordem dos acessos feitos às linhas de *cache*. A linha que não é acessada a mais tempo é a candidata a linha vítima.

2.1.4 Memória *Cache* Associativa por Conjunto

Neste tipo de *cache*, as suas linhas são agrupadas formando diversos conjuntos de linhas dentro da *cache*. O bloco da Memória Principal será mapeado para um desses conjuntos de forma semelhante ao Mapeamento Direto. O bloco 0 será carregado no conjunto 0, o bloco 1 no conjunto 1, e assim sucessivamente. Quando o bloco referente ao último conjunto da *cache* for carregado, o bloco seguinte voltará para o conjunto "0" iniciando mais um ciclo. Veja a Figura 2.7

Diferentes endereços de blocos de memória podem estar mapeados no mesmo conjunto. Neste tipo de *cache*, a TAG guarda os *bits* mais significativos do endereço permitindo identificar de que posição da Memória Principal pertence o conteúdo carregado naquela linha dentro daquele conjunto.

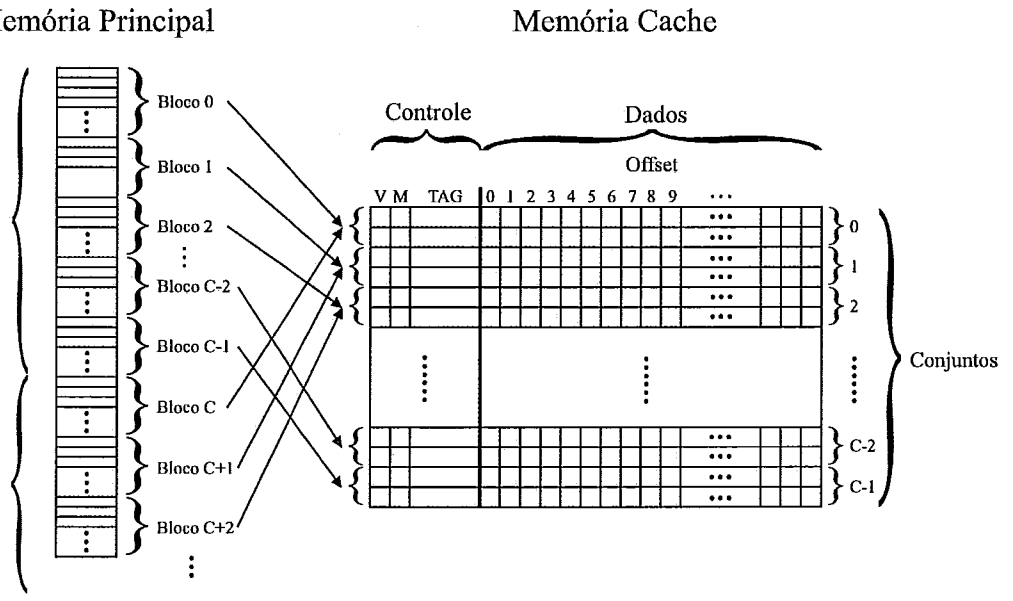


Figura 2.7: Memória *Cache* Associativa por Conjunto com grau de associatividade igual a dois (*two-way set associative*)

Considerando que este mapeamento é feito para um conjunto de linhas, então, poderão ser carregados a mesma quantidade de blocos que linhas existentes no conjunto, diminuindo a possibilidade de falhas na *cache* provocadas por colisão. Dentro de um determinado conjunto, o bloco pode ser carregado em qualquer uma das linhas. Apesar desta *cache* não ser tão versátil quanto a Puramente Associativa, ela resolve o problema de endereços conflitantes mencionado na Memória *Cache* de Mapeamento Direto. Sendo assim, a variedade de blocos da Memória Principal que podem ser carregados em uma determinada linha é bem inferior que a da *cache* Puramente Associativa, necessitando então de um TAG com uma capacidade menor de *bits*.

O carregamento do bloco na linha se dá sob demanda. No momento que a CPU solicita o conteúdo de uma determinada célula da Memória Principal, a Memória *Cache* identifica qual é o número do bloco, o número do conjunto e o número do TAG. Como este bloco pode estar carregado em qualquer linha do respectivo conjunto, a Memória *Cache*, através de um circuito do tipo associativo, compara os *bits* mais significativos do endereço com os TAGs de todas as linhas do grupo ao qual ele está mapeado. Caso haja alguma coincidência, então teremos um acerto, e neste caso a *cache* fornece o seu conteúdo numa velocidade muito superior caso o pedido tivesse sido feito à Memória Principal. Agora, se não for encontrado nenhum TAG igual aos *bits* mais significativos do endereço, teremos uma falha, forçando a Memória *Cache* a carregar este bloco em alguma das linhas do

conjunto. Como esta Memória *Cache* possui varias linhas para colocar o mesmo bloco, também será necessário estabelecer alguma política de substituição.

Formato do Endereço de Memória

Neste tipo de Memória *Cache*, o endereço da Memória Principal será dividido em três partes: deslocamento dentro da linha (*Offset*), número do conjunto e o TAG. O número do conjunto junto com o valor do TAG forma o número do bloco. Veja a Figura 2.8.

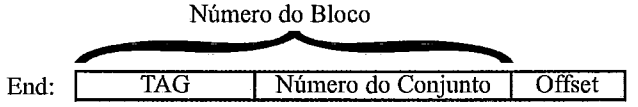


Figura 2.8: Formato do Endereço da Memória Principal Visto por uma Memória *Cache* de Associativa por Conjunto

2.1.5 Políticas de Atualização

Como as escritas feitas pela CPU são inicialmente realizadas na Memória *Cache*, de alguma forma estas informações deverão ser repassadas para a Memória Principal. Estas formas de repassar são chamadas de políticas de atualização. Vou citar três formas de realizar estas atualizações:

- *Write Through*: Este método faz a atualização da Memória Principal no momento em que foi feita a escrita na Memória *Cache*. A vantagem deste método é de manter a Memória Principal sempre atualizada, só que para cada escrita feita pela CPU, será consumido o tempo referente ao da Memória Principal. A vantagem da *cache* será apenas para as operações de leitura que por serem muito mais freqüentes, ainda terá algum ganho com este método, mas, suponha o seguinte caso: um determinado programa possui um *looping* baseado em uma variável que é alterada a cada iteração, por exemplo a variável "i". Suponha que devido a um grande número de variáveis existentes no *looping*, a variável "i" não pode ser mantida em registrador tendo que ter seu conteúdo transferido para a Memória *Cache*. Neste tipo de política de atualização, a cada alteração feita na variável "i" também será feita uma alteração na Memória Principal levando a uma perda de tempo relativamente grande que poderá se agravar mais ainda se este *looping* for muito repetitivo. O problema mencionado anteriormente não se aplica a Memória *Cache* de instruções (I-*cache*),

já que não se faz escritas nas áreas de código. Se por ventura acontecer de haverem escritas, estas serão uma raridade dentre as operações de leitura.

- *Write Back*: Para conseguir aumentar o desempenho nas operações de escrita e com isso resolver o caso mencionado na técnica *Write Through*, esta memória atrasa as atualizações na Memória Principal, mantendo os dados mais recentes apenas na Memória *Cache*. A atualização será realizada somente no momento em que a respectiva linha tiver que ser substituída por outra, a não ser que ela não tenha sido alterada pela CPU. Um problema resultante desta política de atualização é a possibilidade da Memória Principal não ter as mesmas informações que a Memória *Cache*. Se houver algum outro dispositivo neste equipamento que solicite informações à Memória Principal, ele corre o risco de receber dados desatualizados. Uma forma de resolver este problema é selecionar algumas áreas da Memória Principal para serem do tipo não "cacheável", isto é, seus conteúdos não são enviados para a Memória *Cache*. Desta forma podemos usar esta região de memória para transferências entre dispositivos que compartilham o uso da Memória Principal, por exemplo, o controlador de DMA (Acesso Direto à Memória).

- *Write Once*: Esta política de atualização é uma mistura dos dois métodos anteriores e tem como objetivo trazer vantagens tanto nas operações de leitura como nas operações de escrita em sistemas com mais de uma CPU. Este é um caso em que mais de um dispositivo está acessando a Memória Principal. Para que não aconteça o envio de uma informação errada para uma das CPUs, a Memória *Cache* utiliza do recurso da primeira escrita para avisar as demais *caches* do sistema que aquele bloco da Memória Principal passará a ser de uso exclusivo dela. A partir deste momento, todas as operações de escrita e leitura naquele bloco da Memória Principal ficarão restritos a esta *cache*. Quando alguma outra CPU necessitar das informações deste bloco, será feita uma solicitação de atualização à *cache* portadora do bloco. Esta *cache* irá então atualizar a Memória Principal possibilitando que as demais CPUs recebam o dado mais recente.

2.2 Redução dos *Cache Misses*

Para analisarmos formas de reduzir os *Misses* [10], vamos dividi-los em três categorias:

- Compulsório: Quando um processo começa a ser executado, ele está na Memória

Principal. Quando a CPU começa a fazer leituras de memória, a primeira vez que uma linha de *cache* é referenciada certamente ocorrerá um *miss*, já que a informação ainda não foi carregada na *cache*. Sendo assim, todas as primeiras referências a uma linha de *cache* resultam em um *miss*. É o que chamamos de compulsório.

- Capacidade: Caso a Memória *Cache* não tenha capacidade de armazenar todos os blocos de um determinado processo, em algum momento haverá a necessidade de descartar um bloco de informações para poder armazenar um outro. Devido a isso obteremos *misses* que tiveram sua origem na incapacidade da *cache* de armazenar todos os blocos necessários para a execução do processo.

- Conflito: Devido a estratégia de escolha da linha vítima, pode acontecer de um determinado bloco que está carregado na *cache* ser retirado para o carregamento de outro e em momento futuro este mesmo bloco ser referenciado pela CPU acarretando em um *miss* sendo então necessário o seu carregamento. Este tipo de *miss* teve como origem o conflito de um ou mais blocos na mesma linha.

2.2.1 Uma Maior Capacidade de Linha

Considerando o princípio da Localidade Espacial, se aumentarmos o tamanho da linha e com isso o tamanho do bloco da Memória Principal a ser carregado na Memória *Cache*, estaremos diminuindo os *misses* compulsórios já que um número maior de informações será carregado em função de um único *miss* fazendo com que os demais acessos a este bloco leve a um *hit*, a menos que este tenha sido substituído.

Considerando que a capacidade da Memória *Cache* seja constante, ao aumentarmos a capacidade de cada uma de suas linhas, iremos diminuir o total de linhas existente. Sendo assim, a partir de uma determinada capacidade de linha, passaremos a ter uma maior quantidade de *miss* por conflito. Teremos também um aumento no tempo de carregamento da linha e com isso aumentando a penalidade de um *miss*.

2.2.2 Uma Maior Associatividade

Ao aumentarmos a associatividade da Memória *Cache*, isto é, aumentarmos a quantidade de linhas por conjunto, iremos diminuir a quantidade de *misses* por conflito, porém, o aumento da associatividade leva ao aumento do custo do *hit*, isto é, para descobrirmos se houve um *hit*, teremos que gastar mais tempo.

2.2.3 *Cache* de Linhas Vítimas

Como visto anteriormente, o aumento da associatividade também aumenta o tempo para verificar se ocorreu um *hit*, mas a associatividade melhora muito o desempenho: muitas vezes temos o problema de retirar um bloco da linha para carregar um outro bloco e logo depois precisarmos carregar o primeiro bloco novamente. O famoso problema do *Ping-Pong*. Uma maneira de amenizar este problema sem causar prejuízo no tempo de *hit* é a inclusão de uma pequena Memória *Cache* totalmente associativa para armazenar as linhas que foram descartadas a pouco tempo, sendo chamada de *Cache* de Vítimas [11]. Sendo assim, no caso de um *hit*, a *Cache* funcionaria normalmente sem qualquer prejuízo no desempenho e no caso de um *miss*, antes de ir até a Memória Principal, seria feita uma procura nesta *Cache* de Vítimas para saber se o bloco está lá. Em caso positivo, o bloco da Memória *Cache* será trocado com o bloco da *Cache* de Vítimas diminuindo o tempo gasto no *miss*. Esta diminuição só ocorrerá para os casos em que o bloco se encontra na *Cache* de Vítimas, que é o caso do fenômeno do *Ping-Pong*.

2.2.4 *Cache* Pseudo-Associativa

Uma outra forma de possuirmos um taxa de *miss* equivalente a uma Memória *Cache* Associativa por Conjunto e o desempenho de uma com Mapeamento Direto no caso de *hits*, é com uma *cache* chamada de Pseudo-Associativa. O procedimento em caso de *hits* é o mesmo ao do Mapeamento Direto em compensação, no caso de um *miss*, antes de ir à Memória Principal, uma outra linha da *cache* é verificada para saber se o bloco se encontra nela. Uma forma simples de mapear esta segunda linha é invertendo o *bit* mais significativo do número da linha.

Caches Pseudo-Associativas possuem na verdade dois tempos de *hit* correspondente ao *hit* normal e ao *pseudo-hit*. Para que não haja uma queda no desempenho no caso de haverem muitos *pseudo-hits* é colocado o bloco de maior acesso na linha correspondente ao *hit* normal.

2.2.5 Pré-Busca de Instruções Através do Hardware

Uma forma de diminuir o tempo gasto em caso de *miss* é aproveitar o conceito de Localidade Espacial e ao invés de esperar o pedido da CPU por uma determinada instrução para então verificar a sua existência na Memória *Cache* ou ir na Memória Principal, a Memória *Cache* anteciparia a busca do bloco adjacente ao atualmente em uso, colocaria este

bloco em um *buffer* e caso a CPU faça uma referência a ele, seria então transferido para a Memória *Cache* sendo feita uma nova antecipação. Enquanto houver esta confirmação de acesso da CPU, o tempo gasto em um *miss* será muito menor.

2.2.6 *Caches* Associativas por Coluna

Como é comentado por Agarwal e Pudar [1] uma característica importante a ser considerada é o tempo de *hit*, isto é, o tempo que a Memória *Cache* leva para encontrar o dado e transferi-lo para a CPU. Quanto menor for a associatividade da Memória *Cache*, menor será este tempo, sendo assim, a *cache* de associatividade 1 (Mapeamento Direto) terá o melhor desempenho no caso de um acerto. Infelizmente, este tipo de *cache* possui uma taxa de *miss* por conflito muito grande, proporcionando um grande consumo de tempo caso sejam feitos acessos alternados a blocos que estejam mapeados para a mesma linha de *cache*.

A ideia fundamental por traz de uma *cache* associativa por coluna (*column-associative cache*) é a de resolver os conflitos dinamicamente, escolhendo outras linhas através de diferentes funções de *hashing* quando existirem os conflitos de dados. Uma função simples é a inversão do *bit* mais significativo do número da linha.

Esta implementação permite usar a própria *cache* para armazenar os dados conflitantes onde uma simples função de *hashing* é utilizada para acessar esta informação. Comparando com a *cache* de linhas vítimas, esta precisa de uma região puramente associativa separada para armazenar os dados conflitantes. Ela não só consome uma área extra, como também pode ser mais lenta devido a necessidade de uma busca associativa e a lógica para manter uma política de substituição LRU (Menos Recentemente Usada).

- Múltiplas Funções de *Hashing*.

  As *caches* associativas por coluna usam duas ou possivelmente mais funções de *hashing* distintas, h_1 e h_2, para acessar a *cache*, onde $h_1[a]$ refere-se a linha obtida aplicando h_1 ao endereço a. Se $h_1[a_i]$ indexar um dado válido, ocorrerá um acerto (*hit*) de primeira mão; se falhar, $h_2[a_i]$ é então usado para acessar a *cache*. Se ocorrer um acerto de segunda mão o dado será então entregue. Os dados das duas linhas de *cache* serão trocados para que o próximo acesso resulte em um acerto de primeira mão. Se no segundo acesso tiver ocorrido uma falha, então o dado será trazido da Memória Principal e colocado na linha indexado por $h_2[a_i]$ e então trocado com o dado da primeira localização como mostra a Figura 2.9.

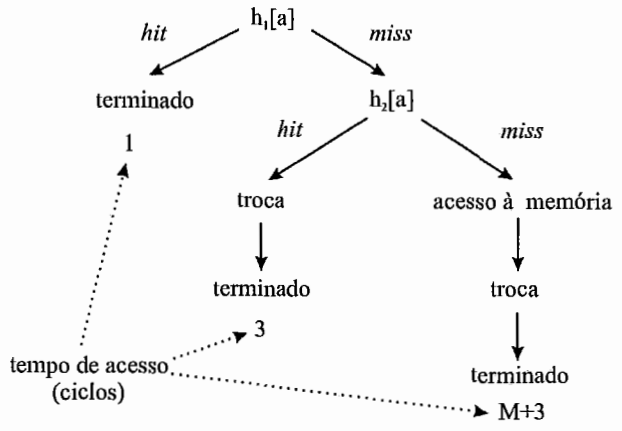


Figura 2.9: Árvore de decisão para o algoritmo *hash-rehash*

- *Rehash bits*

Para que uma *cache* associativa por coluna seja efetivamente implementada, é necessário inibir o acesso "*hasheado*" se a localização alcançada pelo primeiro acesso já possuir um dado "*hasheado*". Para resolver este problema, cada linha da *cache* conterá um *bit* de controle extra para indicar se a linha foi "*hasheada*". Este algoritmo é mostrado na Figura 2.10. Este algoritmo é similar ao *hash-rehash* sendo a diferença no fato de que quando uma linha for substituida, uma linha "*hasheada*" terá sempre a preferência.

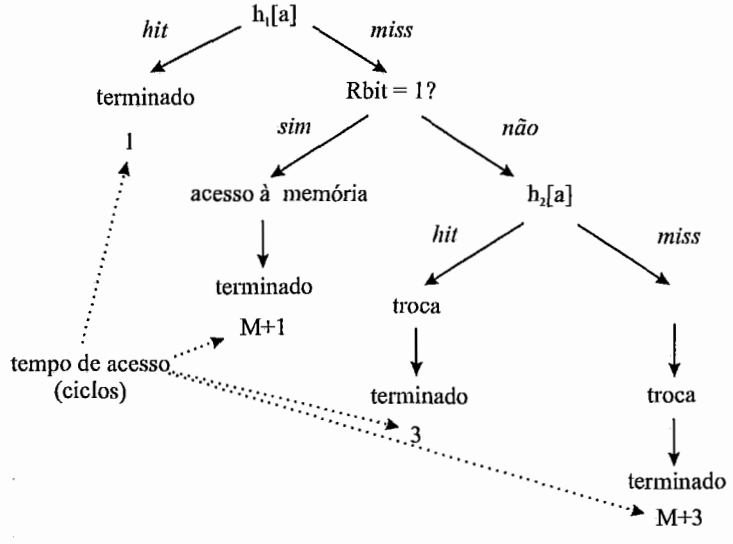


Figura 2.10: Árvore de decisão para uma *cache* associativa por coluna

Capítulo 3

Trabalhos Relacionados

Otimização de código é um tópico de pesquisa que vem evoluindo ao longo dos anos. Inicialmente, competia ao programador a tarefa de organizar as instruções do programa na memória principal. Mesmo com o posterior aparecimento de linguagens Simbólicas (*Assembly language*) o mapeamento de instruções na memória continuou sob responsabilidade do programador.

Naquela ocasião, o grande desafio era produzir programas suficientemente compactos que pudessem ser acomodados integralmente na memória principal (cuja capacidade de armazenamento era bastante reduzida).

Na década de 60, com a criação dos conceitos de **hierarquia de memória** e **memória virtual,** o problema ganhou um novo foco: agora, o tamanho do programa não era um fator limitante; mas sim as interrupções provocadas por páginas ausentes (o processador permanecia interrompido, aguardando pela chegada da página faltosa). O trabalho desenvolvido por D. Ferrari em Berkeley [8] é um exemplo clássico do esforço para reduzir o número de páginas ativas de um programa. No seu trabalho, o autor propõe um método para o mapeamento (posicionamento na memória) das instruções em linguagem de máquina que reduz o número de páginas ativas.

Já havia naquela época a preocupação em reorganizar as instruções de um programa (i.e., gerar um outro *code layout*) de modo a torná-lo mais eficiente.

Com a criação de novos componentes na hierarquia de memória, mais especificamente, na criação da memória *cache*, ficou evidente a necessidade de novas técnicas de reorganização de código que consideram o tamanho da *cache* de instruções. Neste capítulo, faremos um breve resumo de importantes trabalhos na área de reorganização de código.

A técnica de reorganização de Blocos Básicos utilizada tem como objetivo diminuir a

poluição da Memória *Cache* fazendo com que apenas os blocos que estão sendo utilizados sejam carregados, proporcionando um aumento no seu desempenho, já que a maior parte do código não é executado independente dos seus valores de entrada [6].

Pettis e Hansen [14] propuseram uma reorganização dos Blocos Básicos dentro de um procedimento, visando diminuir os desvios internos. Os procedimentos eram então divididos em duas partes: uma com maior freqüência de uso e outra com a menor freqüência. Em seguida, os procedimentos são mapeados na memória de forma que dois procedimentos que se chamam mutualmente são posicionados próximos uns dos outros.

Larus e Thomas [2] propuseram uma técnica de instrumentação onde todos os possíveis caminhos de um determinado procedimento são identificados e depois feita uma totalização da quantidade de vezes que um determinado percurso é realizado. Esta técnica possui uma sobrecarga muito baixa se comparada com as demais técnicas utilizadas.

Hashemi, Kaeli e Calder [9] utilizam uma técnica de reorganização onde é feita uma análise do código de forma estática, isto é, sem que ele seja executado, não havendo então, a necessidade da geração de um perfil. Primeiramente é dado um valor ao procedimento inicial e de acordo com o tipo de desvio existente será feita uma modificação neste valor. Após toda esta análise, os procedimentos serão posicionados de forma a haver a menor sobreposição possível nas linhas de *cache*. Esta técnica proporcionou uma redução de aproximadamente 20% na taxa de *miss*.

John e Kaeli [12] propuseram uma técnica de carregamento de procedimentos em *cache* de forma a diminuir os conflitos. Por exemplo, se for carregado na *cache* dois procedimentos P_1 e P_2 um consecutivo ao outro em termos de linhas de *cache*, quando for solicitado o carregamento de P_3, este seria carregado em linhas que pertencem tanto ao P_1 como ao P_2. A idéia é fazer este carregamento de forma a prejudicar apenas um dos procedimentos e não os dois. Para facilitar a realização desta tarefa, é necessário descobrir a região de maior conflito entre os procedimentos e juntá-las. Foi utilizada uma técnica onde são marcadas as regiões de um procedimento que possuem a maior quantidade de conflitos com outros procedimentos. Estas regiões são chamadas de regiões quentes (*hot regions*). Esta técnica de reorganização proporcionou uma redução de *misses* da ordem de 20% comparado ao melhor algoritmo de reorganização de procedimentos.

Ramirez e Barroso [15] propuseram uma técnica de reorganização de código em aplicações do tipo OLTP ("*on-line transaction processing*") devido a maior parte delas possuirem uma elevada taxa de *miss*. A técnica proposta proporciona uma redução de 55% a 65% no total de *miss* em *caches* de 64 a 128*Kbytes* em uma simulação sem a interferência

do Sistema Operacional. Já com o Sistema, a redução dos *misses* ficou na ordem de 45% à 60%. Esta reorganização do código também proporcionou melhoramentos em outras regiões da memória como por exemplo na TLB e na *cache* unificada de segundo nível. O impacto geral da reorganização foi de 1,33 vezes.

Fernandes e Barbosa [7] discutem o fato da maior parte do código dos programas nunca serem executados, proporcionando um grande potencial para reorganização. Foi feita uma simulação de execução de um processador de Blocos Básicos que utiliza como executável um arquivo bidimensional, isto é, dividido em Blocos Básicos. A vantagem desta forma de execução é de ter a certeza de que ao começar a execução de um determinado Bloco Básico, todas as suas instruções serão executads sequencialmente. Esta idéia também é comentada no trabalho de Fernandes e da Silva [6], que fala da facilidade oferecida no uso da Memória *Cache* e no despacho de multiplas instruções em micro-processadores super-escalares. Durante as simulações realizadas com o SPEC95, foi constatado que 90% das ativações foram feitas por uma fração bem modesta dos códigos.

Os nossos algoritmos mostraram um ganho médio de até 43,62% sendo que o ganho máximo obtido foi de 99,98%. O trabalho de Pettis e Hansen [14] proporcionou um ganho de 98,20%. A técnica utilizada por Hashemi, Kaeli e Calder [9] levou a um ganho de 20%, mesmo resultado alcançado por John e Kaeli [12]. Ramirez e Barroso [15] chegaram a ganhos que vão de 55% à 65%, sem a presença do Sistema Operacional, e de 45% à 60% com o Sistema.

Capítulo 4

Nossa Estratégia e o Ambiente Experimental

No Capítulo 3 apresentamos um resumo dos trabalhos relacionados com o tópico "reorganização de código". Apresentaremos aqui o ambiente experimental que utilizamos e a estratégia de reorganização de código que adotamos.

4.1 Nossa Estratégia

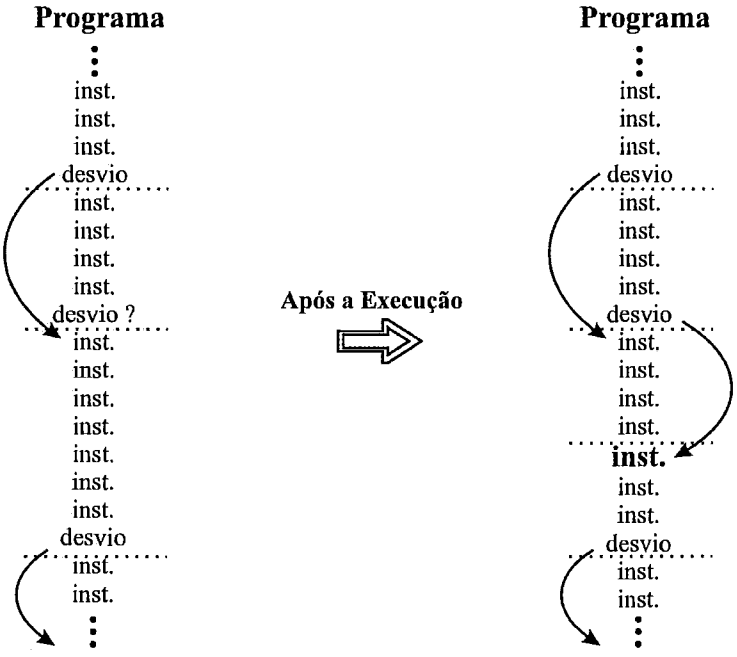
Devida a pequena capacidade das Memórias *Caches*, elas apresentam uma certa taxa de *miss*, que leva a intervalos de tempo muito longos de ociosidade na CPU. Aumentar a capacidade da *cache* implica em custos muito elevados e as *caches* que possuem uma melhor adminstração de suas linhas normalmente possuem uma complexidade tamanha que acarreta em um aumento no tempo gasto para o acesso da informação.

Com base no fato de que os programas normalmente utilizam uma fração pequena do seu código durante as execuções, é proposta a estratégia de reorganização do código binário visando diminuir a taxa de *miss* mantendo a complexidade da *cache* pequena.

Para realizar esta reorganização, serão utilizadas informações referentes ao comportamento dos programas guardadas em um arquivo de perfil. Para a criação deste perfil, será realizada uma simulação integral do programa. Em seguida será feita a sua reorganização utilizando este perfil como base. Após esta reorganização, será repetida a simulação utilizando o código reorganizado.

Antes de reorganizarmos o programa, é necessário dividi-lo em diversos Blocos Básicos que possuem como característica o fato de terem um único ponto de entrada e um único ponto de saída. Sendo assim, uma vez iniciada a execução de um Bloco Básico, todas as suas instruções serão obrigatoriamente executados.

Para demarcarmos um Bloco Básico, precisamos identificar o seu início e fim. Primeiramente é feita a localização de todas as instruções de desvio. Elas marcam o final de um bloco e a instrução adjacente marca o início do bloco seguinte, como pode ser visto na Figura 4.1a.



(a) - antes da geração do perfil (b) - depois da geração do perfil

Figura 4.1: Identificação dos Blocos Básicos

Nem todos os Blocos Básicos tem sua fronteira demarcada pela existência de uma instrução de desvio. Existem situações onde o fluxo de controle é transferido para uma instrução que não é precedida espacialmente por uma de desvio. Neste caso, ela representa o início de um Bloco Básico e a sua antecessora espacialmente marca o final do bloco anterior. O destino de alguns desvios só poderão ser identificados em tempo de execução. Desta forma, durante a criação do perfil, será feita a análise dos desvios e no caso de ser localizado um novo Bloco Básico este será incluido no perfil como mostra a Figura 4.1b.

Como foi mencionado por Fernandes [6], um programa está repleto de Blocos Básicos que nunca são executados, a Memória *Cache* de Mapeamento Direto está sujeita a uma sub-utilização devido ao mapeamento destes blocos em suas linhas, aumentando assim, a incidência de conflitos entre os blocos que são executados. A Figura 4.2 ilustra este fenômeno. Os Blocos marcados de preto e cinza, são os blocos executados, enquanto que os blocos de branco nunca foram executados. Como pode-se observar, parte da Memória

Cache ficou vazia fazendo com que outros blocos fossem mapeados em linhas já utilizadas trazendo a necessidade de uma substituição apesar de haver espaço ocioso na *cache*.

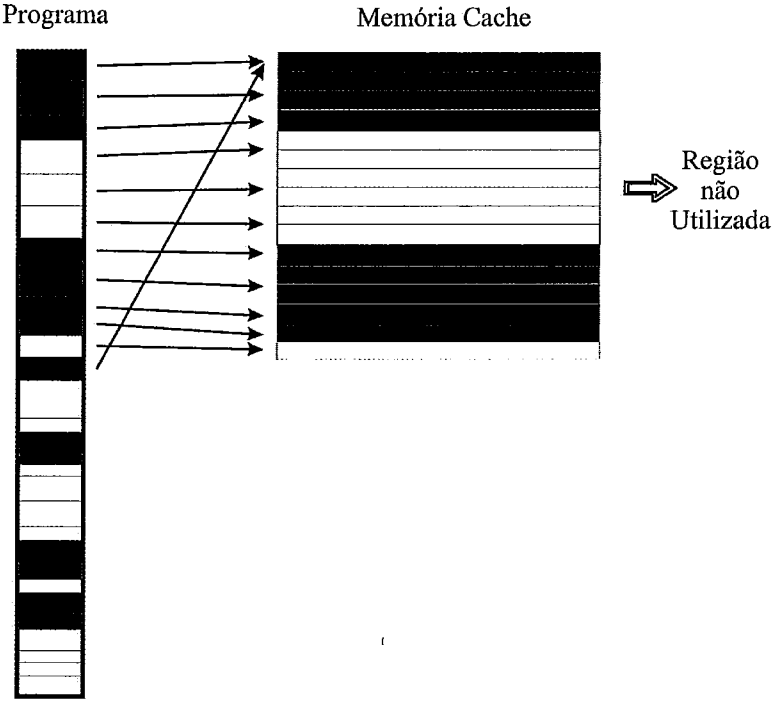


Figura 4.2: Mapeamento dos Blocos Básicos no Programa Original

A nossa proposta é de que ao reorganizar os Blocos Básicos do programa, fazendo com que os blocos que são executados fiquem contíguos na Memória Principal, preferencialmente a partir do primeiro endereço útil do programa, a Memória *Cache* será melhor utilizada, proporcionando um aumento no desempenho da execução. Como pode ser visto na Figura 4.3, após a reorganização dos blocos, provavelmete haverá uma diminuição dos conflitos, possibilitando, no caso em que todos os Blocos Básicos executados couberem na *cache*, a total ausência de *misses* por conflito, restando apenas os *misses* compulsórios.

A criação do perfil consiste em determinar a quantidade de vezes que um Bloco Básico é executado. Para isso, é feita uma simulação completa do programa onde esta contagem é realizada e gravada em um arquivo. Este arquivo de perfil (ver Figura 4.4) possui o número do bloco, o endereço inicial, o endereço final, o total de instruções e a sua freqüência de execução. A Tabela 4.1 mostra um resumo da execução dos programas em relação as instruções enquanto que a Tabela 4.2, mostra um resumo em relação aos Blocos Básicos. O percentual mostrado se refere ao total de instruções (ou blocos) que são executados em relação ao total existente.

No nosso experimento serão realizadas três reorganizações diferentes. Na primeira

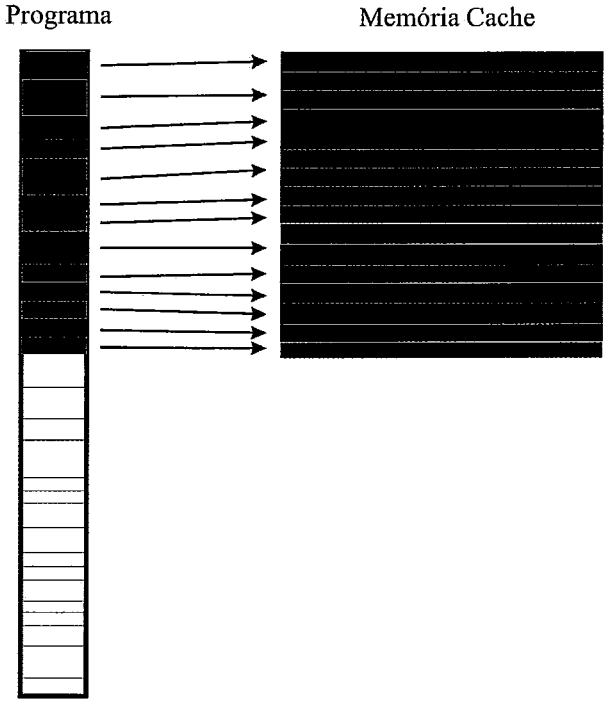


Figura 4.3: Mapeamento dos Blocos Básicos no Programa Reorganizado

# Bloco	Endereço Inicial do Bloco	Endereço Final do Bloco	Total de Instruções	Total de Acessos
0	4194304	4194616	40	0
1	4194624	4194728	14	1
2	4194736	4194768	5	1
3	4194776	4194784	2	1
⋮	⋮	⋮	⋮	⋮

Figura 4.4: Formato de um arquivo de perfil

faremos a transferência dos blocos que nunca foram executados para o final do programa. Na segunda reorganização iremos ordenar os blocos de forma decrescente em relação a sua freqüência de ativação. Na terceira e última reorganização, ordenaremos os blocos em relação a sua contribuição na execução, isto é, a sua freqüência de ativação multiplicada pelo total de instruções do bloco. Uma descrição detalhada dos algoritmos é mostrada no Capítulo 5.

A reorganização implica em verificar no arquivo de perfil a freqüência dos Blocos Básicos e então movimenta-los de acordo com a estratégia estabelecida. Ao movimentar um Bloco Básico é necessário fazer as correções nos desvios que tenham relação com

Programas	Instruções		
	Total	Executadas	Percentual
Compress	12.980	3.165	24,38%
Gcc	270.846	128.836	47,56%
Go	77.700	63.044	81,13%
Ijpeg	49.622	15.054	30,33%
Li	22.580	6.433	28,48%
M88ksim	35.858	8.283	23,09%
Perl	66.948	15.656	23,38%
Vortex	123.866	60.874	49,14%

Tabela 4.1: Estrutura dos Programas em Instruções

Programas	Blocos Básicos		
	Total	Executados	Percentual
Compress	3.499	775	22,14%
Gcc	78.950	34.363	43,52%
Go	14.575	10.843	74,39%
Ijpeg	10.662	2.880	27,01%
Li	6.162	1.641	26,63%
M88ksim	9.235	1.876	20,31%
Perl	19.205	4.013	20,89%
Vortex	28.382	12.598	44,38%

Tabela 4.2: Estrutura dos Programas em Blocos Básicos

ele. As instruções de desvio condicional tem um limite de alcance de 16383 intruções para frente e 16384 instruções para trás. Caso a correção de um destes desvios leve a uma distância superior ao do seu alcance, ela deverá ser substituída por um conjunto de instruções que tenham o mesmo objetivo.

Durante a execução, foi verificada a ocorrência de erros nas simulações. Após um minucioso rastreamento da execução, foi constatado que os programas que estão sendo usados para este trabalho (*BenchMarks CINT95 - Compress, Gcc, Go, Ijpeg, Li, M88ksim, Perl e Vortex*) guardam dados na área de código [16], provavelmente constantes e tabelas de desvios. Como não existe a possibilidade de sabermos o que é instrução e o que é dado, as áreas de dados foram demarcadas como sendo Blocos Básicos e como nenhuma referência de busca de instrução é feita a estes blocos, já que eles na verdade são dados, eles tiveram uma freqüência de ativação nula, levando então, a sua reorganização. Sendo assim, quando o programa vai buscar estes dados no endereço original, ele estará recebendo uma informação errada, levando a erros na simulação dos *benchmarks* reorganizados.

Para podermos realizar o nosso experimento, mudamos de estratégia. Ao invés de mo-

dificarmos realmente o programa original, criamos uma tabela de conversão de endereços entre o programa original e o seu respectivo programa reorganizado. Com esta tabela, a simulação poderá ser realizada utilizando o código original, mas em vez de passarmos para a Memória *Cache* o endereço original da instrução executada, é feita uma consulta à tabela de mapeamento e enviado o endereço modificado, que será o endereço correspondente ao código reorganizado. Desta forma, foi possível realizar o levantamento dos *hits* e *misses* sem que a existência de dados na área de código interferisse na execução, pois para o simulador, o código executado continua sendo o original.

Para a criação das tabelas de conversão de endereços será executado um programa que aplicará um dos algoritmos de reorganização em cima dos dados dos arquivos de perfil gerando um arquivo contendo toda a relação entre os endereços do código original e o reorganizado. A Figura 4.5 mostra o formato deste arquivo.

# Bloco	Endereço Inicial do Bloco Original	Endereço Final do Bloco Original	Endereço Inicial do Bloco Movimentado	Endereço Final do Bloco Movimentado
0	4194304	4194616	4219632	4219944
1	4194624	4194728	4194304	4194408
2	4194736	4194768	4194416	4194448
3	4194776	4194784	4194456	4194464
⋮	⋮	⋮	⋮	⋮

Figura 4.5: Formato de um arquivo de tabelas de conversão

Quando é dado início a execução da simulação no *sim-fast*, é feita a montagem de um vetor de mapeamento contendo todos os endereços das instruções convertidos. A montagem deste vetor é feita através da tabela de conversão de endereços criada anteriormente. Esta tabela esta organizada em Blocos Básicos, sendo necessário expandi-los nos endereços iniciais de cada instrução do programa. Durante a execução, o endereço original é aplicado a uma função que irá converte-lo no índice do vetor de mapeamento. Ao referenciar este vetor na posição indicada pelo índice, será devolvido o endereço referente ao código reorganizado, que será então enviado para a memória *cache*. A Figura 4.6 ilustra este processo.

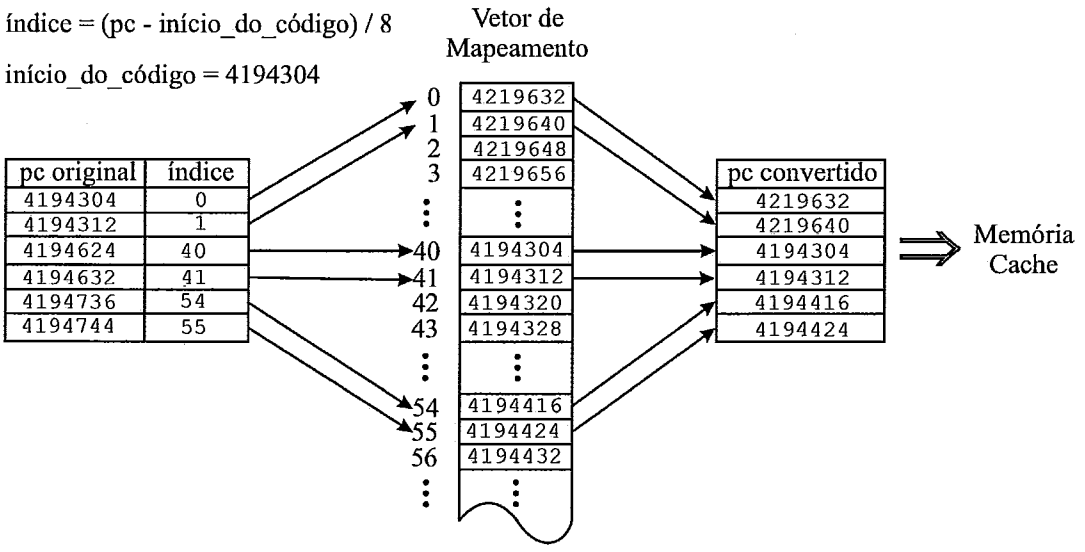


Figura 4.6: Formato do vetor de conversão e como é aplicado

4.1.1 Algoritmos Usados para a Reorganização dos Blocos Básicos

4.1.1.1 Ordenação Original dos Blocos Básicos

Nesta simulação, os programas foram executados sem que houvesse qualquer modificação na distribuição dos Blocos Básicos. A finalidade desta execução é obter as taxas de *misses/hits* produzidas pela ordenação originalmente gerada pelo compilador e compará-las com as correspondentes percentagens produzidas pela execução de cada programa, mas com Blocos Básicos ordenados pelos diferentes algoritmos aqui estudados.

4.1.1.2 Transferência dos Blocos Básicos que Nunca Foram Acessados - (Alg-1)

Este método consiste em transferir os Blocos Básicos que nunca são executados para o final do código. Já que eles nunca serão acessados, é como se estivéssemos reduzindo o tamanho do código binário e portanto, nossa expectativa é que a poluição da memória *cache* de instruções diminua.

Cada arquivo de perfil gerado previamente contém uma descrição das instruções e Blocos Básicos do programa. Ele indica o início, final, total de *bytes* e a freqüência de ativação de cada um dos Blocos Básicos. Empregando estas informações, o algoritmo realiza os seguintes passos para a realocação dos Blocos Básicos:

1. Carregar o arquivo de perfil em um vetor cuja estrutura receba todos os seus campos;

2. Efetuar uma cópia deste vetor. Temos um vetor que será o original e um outro vetor onde será feita a reorganização. Este vetor original será utilizado para gerar a tabela de conversão final relacionando os endereços originais com os produzidos pelo algoritmo;

3. Percorrer o vetor da reorganização até encontrar um Bloco Básico que possua freqüência de ativação igual a zero;

4. Ao encontrar, alterar o início deste bloco para o endereço final do código e o endereço final para representar a sua nova localização (endereço inicial + tamanho do Bloco Básico - 8). A subtração por oito relizada advém do fato das instruções possuirem 8 *bytes*;

5. Deslocar todos os Blocos Básicos a partir deste ponto o equivalente ao tamanho do bloco transferido. Esta alteração se resume em subtrair dos endereços iniciais e finais de todos os Blocos Básicos seguintes, incluindo o que foi transferido, o valor correspondente ao tamanho deste bloco;

6. Voltar a percorrer o vetor da reorganização a partir deste ponto até encontrar outro Bloco Básico que possua freqüência igual a zero e repetir os passos 4, 5 e 6 até chegar ao final do vetor;

7. Ao chegar no final do vetor original, teremos toda a relação entre a ordenação original e a nova organização, já que os Blocos Básicos não foram retirados do vetor e sim, mudamos o endereço de início e fim;

8. Por fim, teremos um vetor com os endereços originais e outro com os endereços reorganizados. É criado o arquivo de mapeamento no formato binário contendo o número do Bloco Básico, o endereço inicial original, o endereço final original, o endereço inicial reorganizado e o endereço final reorganizado.

O arquivo binário será lido pelo simulador na hora da execução para realizar o mapeamento entre o código original e o reorganizado.

4.1.1.3 Ordenação dos Blocos Básicos em Função da sua Freqüência de Ativação - (Alg-2)

Este método consiste em colocar os Blocos Básicos em ordem decrescente de freqüência de ativação que representa a quantidade vezes que eles são executados, sem levar em

consideração a execução individual de cada uma de suas instruções. O objetivo deste método é de colocar os blocos mais freqüêntes em posições adjacentes na Memória *Cache* tentando diminuir as suas substituições. O arquivo contendo a tabela de mapeamento tem o mesmo formato que o utilizado no método anterior.

Através do arquivo de perfil, o algoritmo realiza os seguintes passos para a criação da tabela de mapeamento.

1. Carregar o arquivo de perfil em um vetor cuja estrutura receba todos os seus campos;

2. Efetuar uma cópia deste vetor. Temos um vetor que será o original e um outro vetor onde será feita a reorganização. Tal como no método anterior, o vetor original será utilizado para gerar a tabela de conversão final relacionando os enderecos originais com os produzidos pelo algoritmo;

3. Percorrer o vetor da reorganização até encontrar um Bloco Básico que possua freqüência de ativação inferior ao do bloco que o sucede;

4. Ao encontrar, alterar os valores de início e fim destes blocos para que indiquem a inversão de suas posições;

5. Repetir os itens 3 e 4 até chegar ao final do vetor;

6. Caso haja alguma inversão durante a varredura do vetor, repetir os passos 3, 4 e 5 a partir do seu iníco.

7. Ao chegar no final do vetor sem que haja qualquer inversão de blocos, teremos toda a relação entre a ordenação original e a nova organização;

8. Por fim, teremos um vetor com os endereços originais e outro com os endereços reorganizados. É criado o arquivo de mapeamento no formato binário contendo o mesmo formato indicado no método anterior.

4.1.1.4 Ordenação dos Blocos Básicos em função da sua Contribuição - (Alg-3)

Este método é semelhante ao Alg-2 só que leva em consideração o total de instruções de cada Bloco na hora de calcular a contribuição, fazendo o produto da freqüência de ativação do bloco pelo seu total de instruções. O objetivo deste método é de colocar os Blocos com maior contribuição em posições adjacentes na Memória *Cache* tentando

diminuir as suas substituições. As etapas da reorganização são iguais as do Alg-2 sendo que na etapa 3, ao invés de ser utilizada a freqüência de ativação como referência, será utilizado o produto desta freqüência pelo total de instruções do bloco, gerando assim a freqüência de contribuição.

4.2 Ambiente Experimental

Nosso estudo foi realizado com o simulador *sim-fast* do pacote SimpleScalar Tools 3.0 [3] produzido por Doug Burger e Todd M. Austin na Universidade de *Wisconsin*. O *sim-fast* é um simulador funcional do processador MIPS IV e o código fonte está disponível na distribuição do SimpleScalar. Graças a esta disponibilização, foi possível fazer alterações em seu código de forma a realizar uma simulação personalizada. Foi utilizado um computador Pentium de 400 Mhz e o sistema operacional Linux para rodar os experimentos.

Para a realização da nossa pequisa, foi programada na linguagem C, uma Memória *Cache* de instruções com associatividade *n*. Se *n* for 1, então a *cache* funcionará como sendo de Mapeamento Direto. Se *n* for igual ao total de linhas existentes, ela se comportará como sendo Puramente Associativa, e valores intermediários a estes levará ao comportamento de uma *cache* Associativa por Conjunto com *n* linhas por conjunto.

A *cache* implementada tem a finalidade de verificar a ocorrência de *hits* e *misses*. É feita a contagem de cada um deles e a contagem total das instruções simuladas. Ao final, é gerado um relatório contendo todas estas informações.

O *sim-fast* também apresenta ao final da simulação um relatório contendo diversas informações onde uma delas é o total de instruções simuladas. Este valor é comparado ao contabilizado pela *cache* e usado para validar os resultados produzidos pelas nossas simulações.

Junto com o pacote do *sim-fast* também são distribuídos os programas inteiros do SPEC95 traduzidos para a representação interna do simulador (programas .ss). Estes serão os programas utilizados nas nossas simulações.

Abaixo temos as linhas de comando utilizadas em cada simulação.

- Compress:

```
sim-fast comp.ss test.in
```

- Gcc:

```
sim-fast gcc.ss -funroll-loops -fforce-mem -fcse-follow-jumps
-fcse-skip-blocks -fexpensive-optimizations -fstrength-reduce
-fpeephole -fschedule-insns -finline-functions -fschedule-insns2
-O cccp.i -o cccpi.s
```

- Go:

```
sim-fast go.ss 40 19 null.in
```

- Ijpeg:

```
sim-fast ijpeg.ss -image_file specmun.ppm -compression.quality 90
-compression.optimize_coding 0 -compression.smoothing_factor 90
-difference.image 1 -difference.x_stride 10 -difference.y_stride
10 -verbose 1 -GO.findoptcomp
```

- Li:

```
sim-fast li.ss test.lsp
```

- M88ksim:

```
sim-fast m88ksim.ss -r dhry.lit
```

- Perl:

```
sim-fast perl.ss jumble.pl jumble.in
```

- Vortex:

```
sim-fast vortex.ss vortex.raw
```

A seguir temos uma pequena descrição dos *BenchMarks* utilizados retirada do pacote que acompanha o SPEC95.

- *Compress*: Este programa reduz o tamanho dos arquivos usando a codificação adaptativa de Lempel-Ziv. Sempre que possível, cada arquivo é substituído por um com a extenção ".z". Se nenhum arquivo for especificado, uma entrada padrão é compactada para uma saída padrão. Arquivos compactados podem ser restaurados ao seu formato original usando *Un-compress*.

  O tamanho da compressão obtido depende do tamanho dos dados de entrada, do total de *bits* por caracter e da distribuição das *sub-strings comuns*. Normalmente, entradas do tipo texto são reduzidas em 50-60%.

- *Gcc*: Este programa é baseado no compilador C GNU versão 2.5.3 distribuido pela *Free Software Foundation*. O *benchmark* mede o tempo levado pelo compilador C GNU para converter uma quantidade de seus fontes pré-processados para a saída de arquivos ".s" da linguagem *assembly* otimizada do Sparc.

- *Go*: Este programa é uma versão especial do programa *Go, The Many Faces of Go*, desenvolvido para o uso como parte do pacote de *benchmark* do SPEC. Foi removido todo o código referente à interface com o usuário, as otimizações específicas do PC, o depurador e o verificador de faixas de código e todos os códigos protegidos por #ifdefs. Foram inseridos protótipos de funções para todas as funções e a compilação correu sem avisos tanto para o Compilador C da *Microsoft* como para o da HP. Ele é totalmente escrito em C e não possui dependências com "*endianness*".

  Após a compilação, o executável é totalmente independente. Não há a necessidade de arquivos de entrada. Ele é definido para jogar uma única partida do *Go* contra ele mesmo, com os resultados das jogadas enviados para a saída padrão (*stdout*). Opcionalmente pode-se especificar um arquivo de entrada na linha de comando. O formato deste arquivo é o mesmo que o formato de saída (uma lista de movimentos), para que se possa especificar as posições iniciais.

  Este programa possui a maior parte dos seus cálculos envolvendo números inteiros. Ele possui uma porção muito pequena de uso de números com ponto flutuante. Só no início da execução, onde é feita a inicialização de um vetor. Praticamente não se faz divisões e existem algumas multiplicações. A maioria dos dados é armazenada em vetores unidimensionais para evitar multiplicações.

  Este programa foi extensivamente otimizado usando o *gprof* para termos o máximo de performance. Alguns *loopings* foram desenrolados em C. Ele possui muitos

loopings pequenos e muitos controles de fluxo (*if/them/else*).

- *Ijpeg*: Faz compressões e descompressões na memória de imagens utilizando as facilidades do formato JPEG. Ele faz uma série de compressões em vários níveis de qualidade em cima de diversas imagens.

- *Li*: Este programa é um interpretador *Lisp* escrito em C.

- *M88ksim*: Este programa é um simulador escrito em C. Ele é capaz de medir o número de ciclos de *clock* que um microprocessador 88100 levaria para executar um programa. Ele é essencialmente um programa de cálculos com números inteiros, embora a mistura exata das instruções depende do programa que está sendo simulado.

- *Perl*: É um interpertador da linguagem *Perl*. Nesta simulação são feitos cálculos matemáticos básicos e procuras por palavras em *arrays* associativos.

- *Vortex*: Este programa é um sub-grupo de um programa de banco de dados completamente orientado a objeto chamado *VORTEx*. (*VORTEx* vem de "*Virtual Object Runtime EXpository*").

  As transações feitas do Banco de Dados e para o mesmo são traduzidas através de um esquema. O esquema provê as informações necessárias para gerar o mapeamento do armazenamento interno do bloco de dados para o modelo que possa ser visualizado no contexto da aplicação.

  O esquema vindo com o *benchmark* é pré-configurado para manipular três bancos de dados diferentes: lista de clientes, lista de partes e dados geométricos. São fornecidos tanto os binários *little-endian* como os *big-endian* dos esquemas.

  O *benchmark* monta e manipula três bancos de dados separados (porém inter-relacionados) baseados no esquema. O tamanho do banco de dados é escalável, e para o CINT95 as diretrizes foram restringidas para 40 *Mbytes*.

Capítulo 5

Experimentos

Diversas organizações de memória *cache* foram empregadas nos nossos experimentos. Foram feitas variações em três parâmetros: capacidade da *cache*, associatividade e capacidade da linha. Para a capacidade da *cache*, foram utilizados valores de 2k instruções, 10% do total de instruções do código e 5% do total de instruções executadas pelo menos 1 vez. A Tabela 5.2 contém estas capacidades para cada programa com exceção da capacidade de 2k instruções que é de tamanho fixo. A associatividade foi variada de 1 (Mapeamento Direto), 2 e 4. A capacidade das linhas foram variadas de 1, 2 e 4 instruções. Na *cache* com 5% do total de instruções executadas pelo menos 1 vez utilizamos apenas a política de carregamento Mapeamento Direto com linhas de capacidade para 1 e 4 instruções. Esta combinação de parâmetros nos proporciona 20 variações de *cache* onde cada programa será executa 4 vezes. Uma execução com a organização original dos Blocos Básicos e outras três com diferentes formas de reorganização do código.

O objetivo de usarmos uma *cache* com capacidade para 2k instruções, é para ficarmos mais próximos das *caches* existentes no mercado. Estamos nos referindo as *caches* internas que possuem por volta de 16k *bytes*. Como cada instrução do nosso simulador possui 64 *bits* (8 *bytes* - 2 de anotações e 6 de informações), logo estamos armazenando apenas 2k instruções. Alguns dos programas simulados, possuem uma quantidade pequena de instruções que são realmente executadas possibilitando que quase todas estas instruções permaneçam na *cache*. Sendo assim resolvemos utilizar uma *cache* onde a sua capacidade é proporcional a quantidade de instruções existentes no programa. No nosso caso utilizamos *caches* com capacidade para 10% destas instruções e *caches* com capacidade para 5% do total de instruções executadas pelo menos 1 vez. As Memórias *Caches* comerciais possuem capacidades que são potências de 2, porém, nós resolvemos utilizar na nossa simulação uma capacidade que fosse aproximadamente os 10% e os 5% mencionados

anteriormente, mesmo que não fossem uma potência de 2.

No nosso algoritmo foram contabilizadas as seguintes informações:

- Total de Instruções Executadas;
- Total de Instruções executadas Pelo Menos uma Vez (Primeiro *Miss*);
- Total de falhas na Memória *Cache* de instruções;
- Total de acertos na Memória *Cache* de instruções.

Programas	Total de Instruções Executadas
Compress	3.135.305
Gcc	1.263.333.472
Go	16.389.864.996
Ijpeg	555.447.887
Li	957.028.255
M88ksim	490.351.196
Perl	2.391.943.843
Vortex	9.051.643.607
Total	31.102.748.561

Tabela 5.1: Total de Instruções Executadas

Programas	Capacidade da *Cache* em Instruções	
	10%	5%
Compress	1.312	192
Gcc	27.088	6.444
Go	7.776	3.152
Ijpeg	4.976	752
Li	2.272	324
M88ksim	3.600	416
Perl	6.704	784
Vortex	12.400	3.044

Tabela 5.2: Capacidade da *Cache*: 10% do Total de Instruções do Programa e 5% do Total de Instruções Executadas pelo menos 1 vez

Capítulo 6

Discussão dos Resultados dos Experimentos

Os resultados dos experimentos estão anexados nos três apêndices. No Apêndice A são apresentados os resultados referente a uma *cache* com capacidade para 2k instruções. No Apêndice B os resultados obtidos com uma *cache* com capacidade para 10% do total de instruções do programa e no Apêndice C estão os resultados com uma *cache* capaz de armazenar 5% do total de instruções executadas pelo menos uma vez.

A coluna Ganho/Perda em cada tabela dos apêndices representa de forma percentual se houve uma diminuição na taxa de *misses* (Ganho) ou se houve aumento desta taxa (Perda).

6.1 *Caches* com Capacidade para 2k Instruções

6.1.1 Mapeamento Direto

Nas Tabelas A.1, A.2 e A.3, estão os resultados produzidos pelas simulações utilizando os algoritmos de reorganização Alg-1, Alg-2 e Alg-3 respectivamente com uma *cache* com capacidade para 1 instrução por linha. Para o algoritmo Alg-1, tivemos perda de desempenho nos programas M88ksim e no Vortex. O Gcc e o Perl tiveram ganhos relativamente modestos. Os demais programas tiveram ganhos que variam desde 12,24% até 55,32%. O algoritmo Alg-2 não apresentou perda exceto o Gcc que apresentou um ganho muito pequeno (0,44%). Com o algoritmo Alg-3, só o Gcc obteve perda de desempenho.

As Tabelas A.4, A.5 e A.6 apresentam os resultados obtidos pelas simulações considerando linhas com 2 instruções. O algoritmo Alg-1 continuou apresentando perda de desempenho, porém estas perdas foram menores que as apresentadas na Tabela A.1. O algoritmo Alg-2 desta vez levou a um aumento na perda de desempenho no Gcc. O algo-

ritmo Alg-3 continua proporcionando perda no Gcc.

As Tabelas A.7, A.8 e A.9 apresentam os resultados produzidos pelas simulações considerando linhas com 4 instruções. Neste caso, o algoritmo Alg-1 apresentou perda de desempenho apenas no M88ksim, porém esta perda é menor que as apresentadas nas tabelas referentes as linhas com 1 e 2 instruções. O Alg-2 continua apresentando perda apenas para o Gcc, sendo que esta perda vem sendo progressiva. Na simulação com linhas de 1 instrução não obtivemos perda, mas o ganho foi bem modesto (0,44%). Com linhas de 2 instruções, passamos para uma perda de (1,07%). Agora estamos com uma perda de (4,02%). O Alg-3 apresentou perdas para os programas Gcc, Go e Ijpeg. Estas perdas também vem ocorrendo de forma progressiva em relação ao aumento da capacidade da linha.

6.1.2 Associatividade 2 (*two-way set associative*)

Nas Tabelas A.10, A.11 e A.12, estão os resultados produzidos pelas simulações utilizando os algoritmos de reorganização Alg-1, Alg-2 e Alg-3 respectivamente com uma *cache* com capacidade para 1 instrução por linha. O algoritmo Alg-1 apresentou perda apenas no programa Vortex. O algoritmo Alg-2 apresentou ganho em todos os programas e o Alg-3 proporcionou perda apenas no Gcc.

As Tabelas A.13, A.14 e A.15 apresentam os resultados obtidos pelas simulações considerando linhas com 2 instruções. O algoritmo Alg-1 continuou apresentando perda no Vortex, porém esta perda foi menor que a registrada na simulação com 1 instrução por linha. O algoritmo Alg-2 passou a apresentar perda para o Gcc. O algoritmo Alg-3 apresentou perda para o Gcc, Go e Ijpeg.

As Tabelas A.16, A.17 e A.18 apresentam os resultados obtidos pelas simulações considerando linhas com 4 instruções. O algoritmo Alg-1 apresentou ganhos em todos os programas. O algoritmo Alg-2 apresentou perdas nos programas Gcc e Go. O algoritmo Alg-3 continua apresentando perda no Gcc, Go e Ijpeg.

6.1.3 Associatividade 4 (*four-way set associative*)

Nas Tabelas A.19, A.20 e A.21, estão os resultados produzidos pelas simulações utilizando os algoritmos de reorganização Alg-1, Alg-2 e Alg-3 respectivamente com uma *cache* com capacidade para 1 instrução por linha. O algoritmo Alg-1 apresentou perda nos programas Perl e Vortex. O algoritmo Alg-2 apresentou perda apenas no Ijpeg e o algoritmo Alg-3 apresentou perda no Gcc e no Ijpeg.

As Tabelas A.22, A.23 e A.24 apresentam os resultados obtidos pelas simulações considerando linhas com 2 instruções. O algoritmo Alg-1 apresentou perda apenas no programa Vortex. O algoritmo Alg-2 apresentou perda no Go e no Ijpeg e o algoritmo Alg-3 apresentou perda para os programas Gcc, Go e Ijpeg.

As Tabelas A.25, A.26 e A.27 apresentam os resultados obtidos pelas simulações considerando linhas com 4 instruções. O algoritmo Alg-1 apresentou perda apenas para o Vortex. O algoritmo Alg-2 teve perda no Gcc e no Go e o algoritmo Alg-3 continuou tendo perdas nos programas Gcc, Go e Ijpeg.

A Figura 6.1 mostra a média dos ganhos para cada algoritmo referente a cada um dos programas executados, que está representada na Tabela A.28. Como pode ser observado, existem situações em que tivemos perda no desempenho. No final da tabela é apresentado uma média geral para cada algoritmo em uma *cache* com 2k instruções que também podem ser vistos na Figura 6.2. Em média tivemos um ganho de 40,67% para o algoritmo Alg-1, 52,55% para o algoritmo Alg-2 e 41,33% para o algoritmo Alg-3, mostrando que apesar das perdas de algumas simulações, ainda obtivemos ganhos superiores a 40%.

O algoritmo Alg-1 obteve uma perda média apenas no programa Vortex. O algoritmo Alg-2 não apresentou uma perda média e o Algoritmo Alg-3 apresentou perdas médias nos programas Gcc, Go e Ijpeg.

6.2 *Caches* com Capacidade para 10% do Total de Instruções dos Programas

6.2.1 Mapeamento Direto

Nas Tabelas B.1, B.2 e B.3, estão os resultados produzidos pelas simulações utilizando os algoritmos de reorganização Alg-1, Alg-2 e Alg-3 respectivamente com *cache* com capacidade para 1 instrução por linha. Para o algoritmo Alg-1, tivemos perda de desempenho nos programas Compress, Li, M88ksim e no Vortex. Os algoritmos Alg-2 e Alg-3 não apresentaram perdas, contudo, o Alg-2 obteve maiores ganhos que o Alg-3.

As Tabelas B.4, B.5 e B.6 apresentam os resultados obtidos pelas simulações considerando linhas com 2 instruções. O algoritmo Alg-1 apresentou perda nos programas Li e Vortex, mostrando uma melhora em relação a simulação com linhas de apenas 1 instrução. O algoritmo Alg-2 não apresentou perda. O algoritmo Alg-3 apresentou perda no programa Gcc.

As Tabelas B.7, B.8 e B.9 apresentam os resultados obtidos pelas simulações consi-

derando linhas com 4 instruções. Neste caso, o algoritmo Alg-1 continuou apresentando perda para o Li e o Vortex, porém esta perda é menor que as apresentadas nas tabelas referentes as linhas com 1 e 2 instruções. O algoritmo Alg-2 continua não apresentando perda. O Alg-3 apresentou perda para os programas Gcc e Ijpeg.

6.2.2 Associatividade 2 (*two-way set associative*)

Nas Tabelas B.10, B.11 e B.12, estão os resultados produzidos pelas simulações utilizando os algoritmos de reorganização Alg-1, Alg-2 e Alg-3 respectivamente com uma *cache* com capacidade para 1 instrução por linha. O algoritmo Alg-1 apresentou uma perda expressiva nos programas M88ksim e Perl. Os algoritmo Alg-2 e Alg-3 apresentaram perdas no Gcc e Ijpeg.

As Tabelas B.13, B.14 e B.15 apresentam os resultados obtidos pelas simulações considerando linhas com 2 instruções. O algoritmo Alg-1 continuou apresentando uma perda expressiva do M88ksim e do Perl, porém esta perda foi menor que a registrado na simulação com 1 instrução por linha. O algoritmo Alg-2 apresentou perda para o Gcc e o Ijpeg. Já o algoritmo Alg-3 apresentou perda apenas para o Gcc.

As Tabelas B.16, B.17 e B.18 apresentam os resultados obtidos pelas simulações considerando linhas com 4 instruções. O algoritmo Alg-1 continua apresentando perda nos programas M88ksim e Perl, apesar desta perda ser ainda menor que com linhas de 2 instruções. O algoritmo Alg-2 apresentou perdas apenas no Gcc. O algoritmo Alg-3 passou a apresentandar perda no Gcc e no Go.

6.2.3 Associatividade 4 (*four-way set associative*)

Nas Tabelas B.19, B.20 e B.21, estão os resultados produzidos pelas simulações utilizando os algoritmos de reorganização Alg-1, Alg-2 e Alg-3 respectivamente com uma *cache* com capacidade para 1 instrução por linha. O algoritmo Alg-1 apresentou perda de desempenho nos programas Go, Ijpeg e Perl. Os algoritmos Alg-2 e Alg-3 apresentaram perdas nos programs Gcc, Go e Ijpeg.

As Tabelas B.22, B.23 e B.24 apresentam os resultados obtidos pelas simulações considerando linhas com 2 instruções. O algoritmo Alg-1 apresentou perda nos programas Go e Perl. O algoritmo Alg-2 apresentou perda apenas no Go e o algoritmo Alg-3 apresentou perda para os programas Gcc e Go.

As Tabelas B.25, B.26 e B.27 apresentam os resultados obtidos pelas simulações considerando linhas com 4 instruções. Os algoritmos Alg-1 e Alg-2 apresentaram perda

apenas para o Go. O algoritmo Alg-3 apresentou perda nos programas Gcc e Go.

A Figura 6.3 mostra a média dos ganhos para cada algoritmo referente a cada um dos programas executados, que está representado na Tabela B.28. Para esta *cache* também obtivemos situações de perda de desempenho. Analizando a média geral apresentada no final da tabela e representada na Figura 6.4, verificamos ganhos médios de 54,58% para o algoritmo Alg-2 e 46,95% para o algoritmo Alg-3. O algoritmo Alg-1 mostrou que em média obteve perda. No caso deste algoritmo específico, o programa M88ksim teve um comportamento atípico que resultou em uma perda média de 93,87%. Ele foi o responsável pelo baixo desempenho oferecido por este algoritmo que foi de -0,01%. Como este valor ficou muito próximo do zero, devido a escala do gráfico, não foi possível a representação do algoritmo Alg-1. Se não considerarmos a sua contribuição, teríamos um ganho médio de 13,19% como pode ser visto na Figura 6.5.

O algoritmo Alg-1 apresentou perda média nos programas M88ksim e Perl. O algoritmo Alg-2 continuou não apresentando perda média e o Algoritmo Alg-3 apresentou perda média apenas no programa Gcc.

6.3 *Caches* com Capacidade para 5% do Total de Instruções Executadas pelo menos 1 Vez

6.3.1 Mapeamento Direto

Nas Tabelas C.1, C.2 e C.3, estão os resultados produzidos pelas simulações utilizando os algoritmos de reorganização Alg-1, Alg-2 e Alg-3 respectivamente com uma *cache* com capacidade para 1 instrução por linha. Para o algoritmo Alg-1, tivemos perda apenas no programa Li. O algoritmo Alg-2 apresentou perda nos programas Compress e Ijpeg. O algoritmo Alg-3 apresentou perda nos programas Compress, Gcc e Ijpeg.

As Tabelas C.4, C.5 e C.6 apresentam os resultados obtidos pelas simulações considerando linhas com 4 instruções. Neste caso, o algoritmo Alg-1 continuou apresentando perda para o Li, porém esta perda é menor que a apresentada na tabela referente as linhas com 1 instrução. O algoritmo Alg-2 apresentou perda para o Ijpeg. O Alg-3 apresentou perda para os programas Compress, Gcc, Go, Ijpeg e o Li.

A Figura 6.6 mostra a média dos ganhos para cada algoritmo referente a cada um dos programas executados, que está representada na Tabela C.7. Para esta *cache* também obtivemos situações de perda de desempenho. Analizando a média geral apresentada no final da tabela e ilustrada pela Figura 6.7, verificamos ganhos médios de 15,82% para

o algoritmo Alg-1, 23,72% para o algoritmo Alg-2 e 0,43% para o algoritmo Alg-3. O algoritmo Alg-3 apresentou um ganho médio muito pequeno, já que 3 dos programas simulados levaram a perdas. Para esta configuração de *cache* temos um volume de dados muito menor do que o utilizado nos casos anteriores, sendo injusta a sua comparação.

O algoritmo Alg-1 apresentou perda média apenas no programa Li. O algoritmo Alg-2 apresentou perda média apenas no Compress e o Algoritmo Alg-3 apresentou perda média nos programas Compress, Gcc e Ijpeg.

A Figura 6.8 apresenta a média dos resultados obtidos por cada algoritmo em relação a todas as simulações realizadas. Como pode ser observado, tivemos em média ganhos em todos os algoritmos.

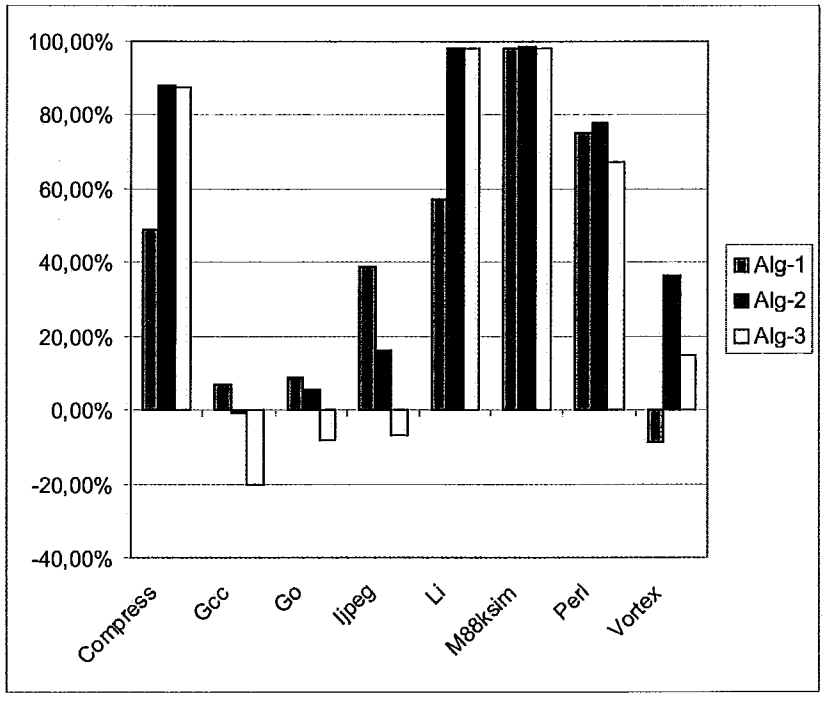


Figura 6.1: Médias das Simulações em uma Memória Cache com Capacidade para 2k Instruções

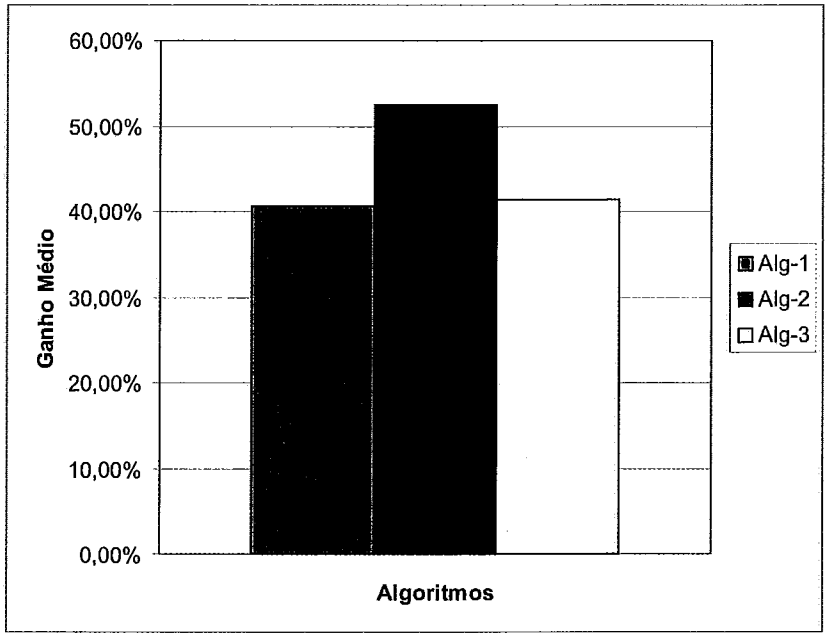


Figura 6.2: Ganho Médio dos Algoritmos em uma Cache com capacidade para 2k instruções

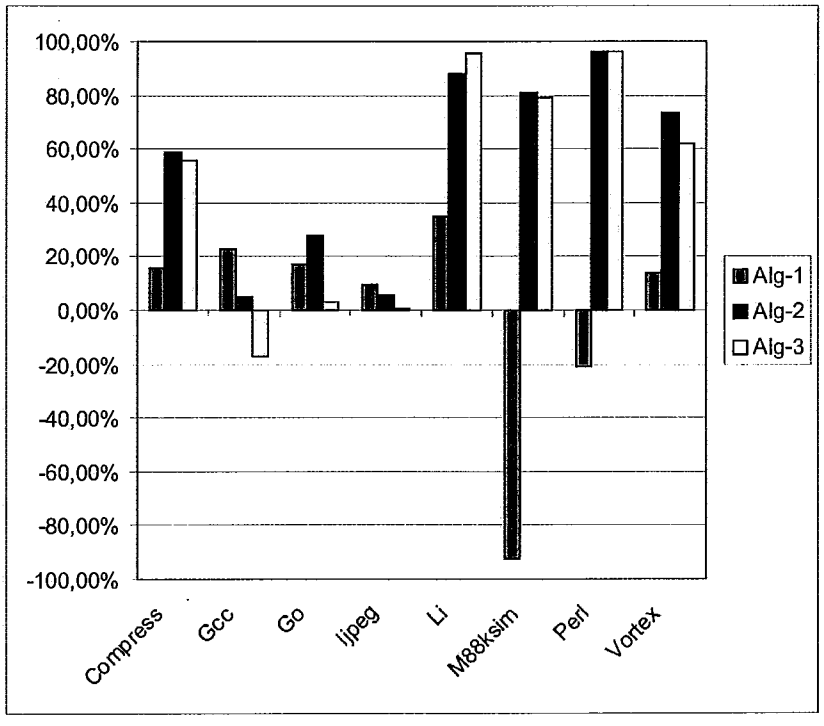


Figura 6.3: Médias das Simulações em uma Memória Cache com Capacidade para 10% do Total de Instruções do Programa

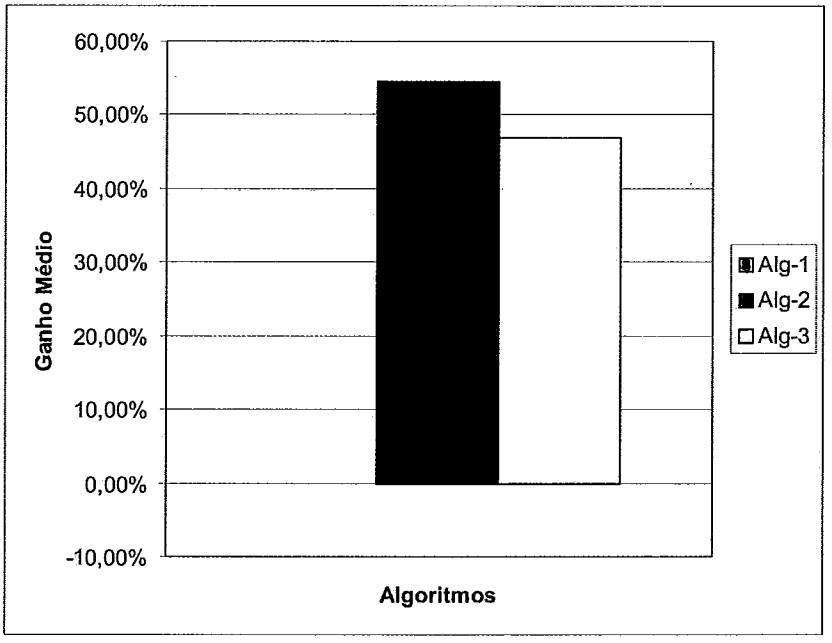


Figura 6.4: Ganho Médio dos Algoritmos em uma Cache com capacidade para 10% do total de instruções

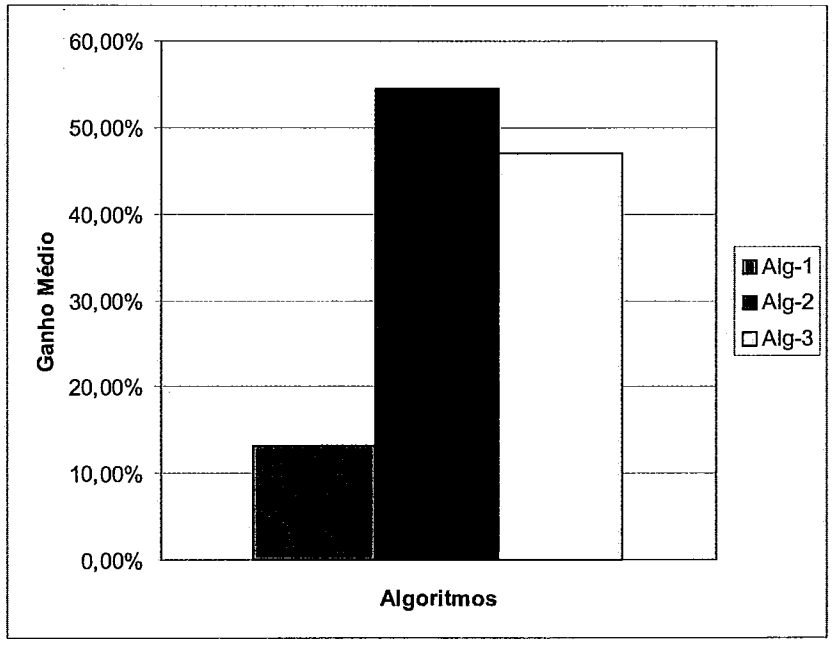


Figura 6.5: Ganho Médio dos Algoritmos com exceção do M88ksim em uma Cache com capacidade para 10% do total de instruções

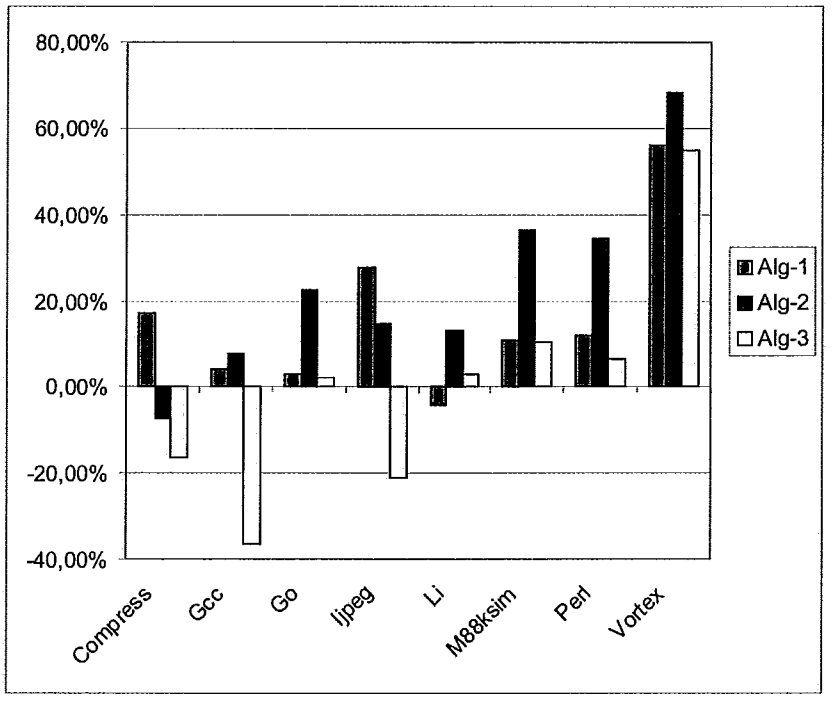


Figura 6.6: Médias das Simulações em uma Memória Cache com Capacidade para 5% do Total de Instruções Executadas pelo menos uma Vez

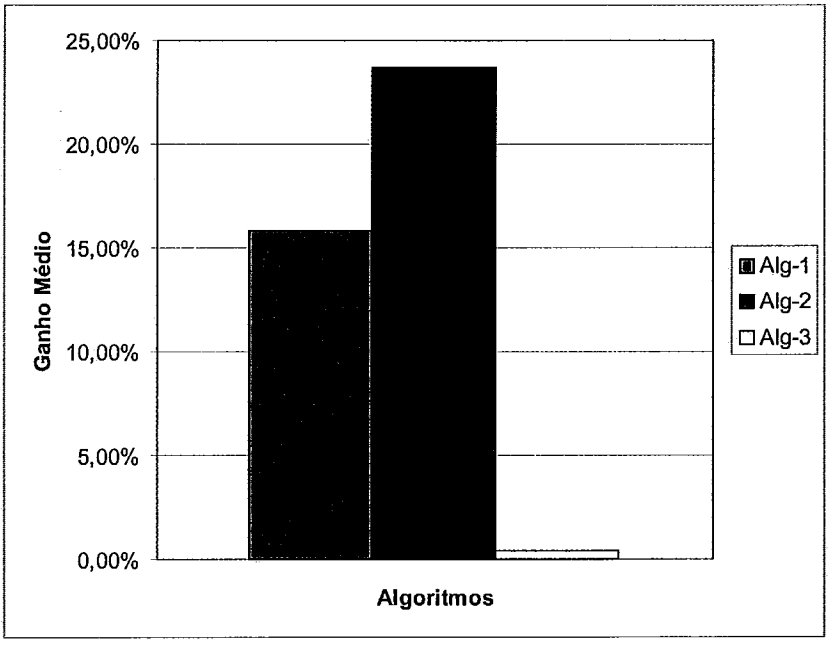


Figura 6.7: Ganho Médio dos Algoritmos em uma Cache com capacidade para 5% do total de instruções executadas pelo menos um vez

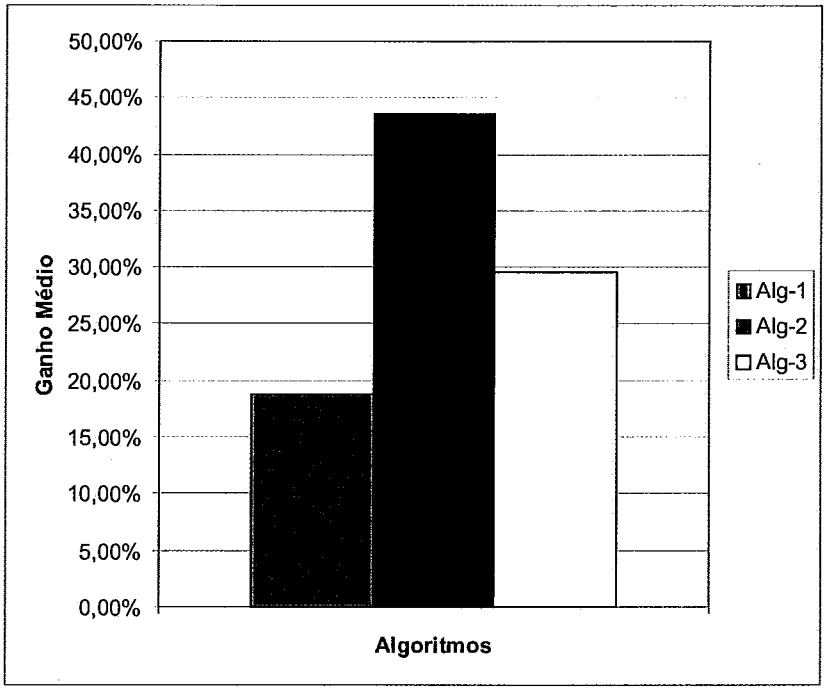


Figura 6.8: Ganho Médio Final dos Algoritmos

Capítulo 7

Conclusões

Nesta tese propusemos formas de reduzir as falhas na Memória *Cache* de instruções de arquiteturas do tipo RISC. Com o grande aumento de velocidade dos processadores, e a incapacidade das memórias em acompanhar este aumento, a ociosidade do processador ao aguardar uma informação da Memória Principal vem crescendo. As Memórias *caches* reduziram bastante esta ociosidade, porém na ocorrência de uma falha, exige a necessidade da consulta à Memória Principal resultando em ociosidade no processador.

Partindo do conhecimento de que grande parte de um programa nunca é executado, especificamos e implementamos três algoritmos que reorganizarão o código destes programas com o objetivo de reduzir a taxa de falhas na Memória *Cache* de instruções. Para esta reorganização é necessário verificar que trechos do código nunca são executados. Foi feita uma simulação inicial onde foram contadas a freqüência de acesso de cada Bloco Básico e armazenadas em um arquivo de perfil. Com base neste perfil, foram definidos os três algoritmos de reorganização. O primeiro algoritmo (Alg-1) faz a transferência dos blocos que nunca foram executados para o final do código. O segundo algoritmo (Alg-2) ordena os Blocos Básico em ordem decrescente em relação a sua freqüência de ativação. O terceiro algoritmo (Alg-3) classifica em ordem decrescente os Blocos Básicos segundo a contribuição que cada bloco forneceu ao total de instruções executadas (*Instruction Counts*). Foi implementada uma Memória *Cache* parametrizável para contabilizar o total de falhas. Este total foi avaliado na execução do código original e na simulação do código gerado por cada algoritmo de reorganização. Estas taxas foram comparadas para verificar se houve ganho ou perda.

A maior dificuldade encontrada na reorganização dos Blocos Básicos é a necessidade de ajustar o campo de endereços das instruções de desvio. Como agravante, havia o problema da existência de dados na área de código. Diante destas dificuldades utilizamos

uma tabela de mapeamento de endereços que converte o endereço original no seu respectivo endereço reordenado, possibilitando fazer o experimento sem a necessidade do ajuste dos endereços.

Os nossos experimentos mostraram ganhos médio de 18,83% para o algoritmo Alg-1, 43,62% para o algoritmo Alg-2 e 29,57% para o algoritmo Alg-3. Apesar de algumas simulações não obterem o ganho esperado, em média verifica-se que estas técnicas de reorganização do código aumentam o seu desempenho.

Como foi verificado nesta pesquisa, a simples reorganização do código binário de um programa pode trazer um alto rendimento no uso da memória *cache*. Os algoritmos aqui apresentados não levam em consideração a ordem de chamada dos Blocos Básicos, sugerindo que novas pesquisas sejam realizadas com outros algoritmos de reorganização visando descobrir alguns que ofereçam maiores ganhos. É provável que não haja um algoritmo que seja ótimo para todos os casos, como pode ser visto nos resultados desta pesquisa, porém deve existir um que melhor se adeqüa aquela aplicação específica.

Apêndice A

Resultados dos Experimentos para uma *Cache* com Capacidade para 2k instruções

Este apêndice apresenta os resultados das simulações realizadas considerando *caches* com capacidade para 2k instruções. A disposição destes resultados está na forma de tabelas. Cada tabela representa a comparação de um dos algoritmos propostos e a simulação com o código na sua organização original. Para cada *cache* utilizada temos três tabelas, uma para cada algoritmo. Para facilitar a visualização, foram dispostas três tabelas por página, cada uma de um algoritmo diferente, tendo em cada página todos os resultados para cada tipo de *cache*.

Programas	Miss Original	Miss Alg - 1	Ganho/Perda
Compress	49.270	33.062	32,90%
Gcc	264.480.587	246.702.891	6,72%
Go	4.279.649.872	3.755.658.191	12,24%
Ijpeg	8.332.210	3.722.756	55,32%
Li	30.725.154	21.440.264	30,22%
M88ksim	51.224.447	80.780.049	-57,70%
Perl	306.551.689	282.090.540	7,98%
Vortex	1.726.993.272	1.835.743.551	-6,30%

Tabela A.1: Desempenho de uma *Cache* com capacidade para 2k instruções, Mapeamento Direto e linhas com capacidade para 1 instrução utilizando o algoritmo de reorganização do código Alg-1.

Programas	Miss Original	Miss Alg - 2	Ganho/Perda
Compress	49.270	3.465	92,97%
Gcc	264.480.587	263.329.538	0,44%
Go	4.279.649.872	3.402.906.412	20,49%
Ijpeg	8.332.210	6.693.075	19,67%
Li	30.725.154	249.446	99,19%
M88ksim	51.224.447	70.472	99,86%
Perl	306.551.689	47.047.648	84,65%
Vortex	1.726.993.272	1.071.417.207	37,96%

Tabela A.2: Desempenho de uma *Cache* com capacidade para 2k instruções, Mapeamento Direto e linhas com capacidade para 1 instrução utilizando o algoritmo de reorganização do código Alg-2.

Programas	Miss Original	Miss Alg - 3	Ganho/Perda
Compress	49.270	3.539	92,82%
Gcc	264.480.587	277.207.195	-4,81%
Go	4.279.649.872	3.593.870.296	16,02%
Ijpeg	8.332.210	7.115.799	14,60%
Li	30.725.154	187.476	99,39%
M88ksim	51.224.447	78.138	99,85%
Perl	306.551.689	42.350.865	86,18%
Vortex	1.726.993.272	1.350.183.040	21,82%

Tabela A.3: Desempenho de uma *Cache* com capacidade para 2k instruções, Mapeamento Direto e linhas com capacidade para 1 instrução utilizando o algoritmo de reorganização do código Alg-3.

Programas	Miss Original	Miss Alg - 1	Ganho/Perda
Compress	28.235	18.345	35,03%
Gcc	150.371.950	137.906.004	8,29%
Go	2.283.253.236	1.972.516.782	13,61%
Ijpeg	4.481.090	1.910.363	57,37%
Li	16.867.297	11.514.093	31,74%
M88ksim	29.986.340	44.664.128	-48,95%
Perl	179.335.215	154.683.442	13,75%
Vortex	971.467.080	977.903.151	-0,66%

Tabela A.4: Desempenho de uma *Cache* com capacidade para 2k instruções, Mapeamento Direto e linhas com capacidade para 2 instruções utilizando o algoritmo de reorganização do código Alg-1.

Programas	Miss Original	Miss Alg - 2	Ganho/Perda
Compress	28.235	1.746	93,82%
Gcc	150.371.950	151.974.154	-1,07%
Go	2.283.253.236	1.859.243.306	18,57%
Ijpeg	4.481.090	3.495.855	21,99%
Li	16.867.297	131.840	99,22%
M88ksim	29.986.340	37.575	99,87%
Perl	179.335.215	24.685.198	86,24%
Vortex	971.467.080	555.055.943	42,86%

Tabela A.5: Desempenho de uma *Cache* com capacidade para 2k instruções, Mapeamento Direto e linhas com capacidade para 2 instruções utilizando o algoritmo de reorganização do código Alg-2.

Programas	Miss Original	Miss Alg - 3	Ganho/Perda
Compress	28.235	1.859	93,42%
Gcc	150.371.950	179.414.682	-19,31%
Go	2.283.253.236	2.153.462.016	5,68%
Ijpeg	4.481.090	4.172.757	6,88%
Li	16.867.297	117.868	99,30%
M88ksim	29.986.340	47.431	99,84%
Perl	179.335.215	24.860.828	86,14%
Vortex	971.467.080	775.078.297	20,22%

Tabela A.6: Desempenho de uma *Cache* com capacidade para 2k instruções, Mapeamento Direto e linhas com capacidade para 2 instruções utilizando o algoritmo de reorganização do código Alg-3.

Programas	Miss Original	Miss Alg - 1	Ganho/Perda
Compress	17.835	11.038	38,11%
Gcc	92.459.004	82.060.465	11,25%
Go	1.261.462.518	968.430.147	23,23%
Ijpeg	2.517.261	1.017.084	59,60%
Li	11.617.091	6.653.451	42,73%
M88ksim	19.218.725	24.909.749	-29,61%
Perl	119.830.521	92.739.718	22,61%
Vortex	583.352.511	538.719.875	7,65%

Tabela A.7: Desempenho de uma *Cache* com capacidade para 2k instruções, Mapeamento Direto e linhas com capacidade para 4 instruções utilizando o algoritmo de reorganização do código Alg-1.

Programas	Miss Original	Miss Alg - 2	Ganho/Perda
Compress	17.835	895	94,98%
Gcc	92.459.004	96.178.806	-4,02%
Go	1.261.462.518	1.052.825.337	16,54%
Ijpeg	2.517.261	1.911.085	24,08%
Li	11.617.091	75.571	99,35%
M88ksim	19.218.725	20.707	99,89%
Perl	119.830.521	14.073.963	88,26%
Vortex	583.352.511	297.467.353	49,01%

Tabela A.8: Desempenho de uma *Cache* com capacidade para 2k instruções, Mapeamento Direto e linhas com capacidade para 4 instruções utilizando o algoritmo de reorganização do código Alg-2.

Programas	Miss Original	Miss Alg - 3	Ganho/Perda
Compress	17.835	1.016	94,30%
Gcc	92.459.004	133.468.883	-44,35%
Go	1.261.462.518	1.472.990.345	-16,77%
Ijpeg	2.517.261	2.654.446	-5,45%
Li	11.617.091	84.128	99,28%
M88ksim	19.218.725	31.449	99,84%
Perl	119.830.521	16.171.221	86,50%
Vortex	583.352.511	497.617.848	14,70%

Tabela A.9: Desempenho de uma *Cache* com capacidade para 2k instruções, Mapeamento Direto e linhas com capacidade para 4 instruções utilizando o algoritmo de reorganização do código Alg-3.

Programas	Miss Original	Miss Alg - 1	Ganho/Perda
Compress	25.479	12.633	50,42%
Gcc	236.192.838	225.443.032	4,55%
Go	3.492.750.495	3.222.822.391	7,73%
Ijpeg	3.947.989	2.461.274	37,66%
Li	16.513.234	9.017.237	45,39%
M88ksim	36.960.117	28.614.746	22,58%
Perl	189.834.063	147.725.026	22,18%
Vortex	1.352.995.587	1.593.304.049	-17,76%

Tabela A.10: Desempenho de uma *Cache* com capacidade para 2k instruções, Associatividade 2 e linhas com capacidade para 1 instrução utilizando o algoritmo de reorganização do código Alg-1.

Programas	Miss Original	Miss Alg - 2	Ganho/Perda
Compress	25.479	3.556	86,04%
Gcc	236.192.838	234.805.812	0,59%
Go	3.492.750.495	3.301.144.399	5,49%
Ijpeg	3.947.989	3.000.804	23,99%
Li	16.513.234	349.655	97,88%
M88ksim	36.960.117	66.099	99,82%
Perl	189.834.063	34.279.997	81,94%
Vortex	1.352.995.587	932.989.452	31,04%

Tabela A.11: Desempenho de uma *Cache* com capacidade para 2k instruções, Associatividade 2 e linhas com capacidade para 1 instrução utilizando o algoritmo de reorganização do código Alg-2.

Programas	Miss Original	Miss Alg - 3	Ganho/Perda
Compress	25.479	3.533	86,13%
Gcc	236.192.838	238.501.780	-0,98%
Go	3.492.750.495	3.357.243.623	3,88%
Ijpeg	3.947.989	3.839.553	2,75%
Li	16.513.234	231.101	98,60%
M88ksim	36.960.117	74.683	99,80%
Perl	189.834.063	50.997.656	73,14%
Vortex	1.352.995.587	1.089.201.338	19,50%

Tabela A.12: Desempenho de uma *Cache* com capacidade para 2k instruções, Associatividade 2 e linhas com capacidade para 1 instrução utilizando o algoritmo de reorganização do código Alg-3.

Programas	Miss Original	Miss Alg - 1	Ganho/Perda
Compress	15.531	7.278	53,14%
Gcc	135.641.527	126.292.425	6,89%
Go	1.849.834.841	1.696.384.813	8,30%
Ijpeg	2.129.913	1.259.639	40,86%
Li	9.912.603	4.948.174	50,08%
M88ksim	24.922.391	16.618.348	33,32%
Perl	114.517.670	81.841.015	28,53%
Vortex	777.875.343	849.553.722	-9,21%

Tabela A.13: Desempenho de uma *Cache* com capacidade para 2k instruções, Associatividade 2 e linhas com capacidade para 2 instruções utilizando o algoritmo de reorganização do código Alg-1.

Programas	Miss Original	Miss Alg - 2	Ganho/Perda
Compress	15.531	1.794	88,45%
Gcc	135.641.527	135.846.026	-0,15%
Go	1.849.834.841	1.800.010.477	2,69%
Ijpeg	2.129.913	1.561.430	26,69%
Li	9.912.603	180.231	98,18%
M88ksim	24.922.391	34.861	99,86%
Perl	114.517.670	18.174.857	84,13%
Vortex	777.875.343	481.129.038	38,15%

Tabela A.14: Desempenho de uma *Cache* com capacidade para 2k instruções, Associatividade 2 e linhas com capacidade para 2 instruções utilizando o algoritmo de reorganização do código Alg-2.

Programas	Miss Original	Miss Alg - 3	Ganho/Perda
Compress	15.531	1.825	88,25%
Gcc	135.641.527	156.278.347	-15,21%
Go	1.849.834.841	1.991.253.885	-7,64%
Ijpeg	2.129.913	2.229.724	-4,69%
Li	9.912.603	146.589	98,52%
M88ksim	24.922.391	47.469	99,81%
Perl	114.517.670	28.606.231	75,02%
Vortex	777.875.343	628.143.756	19,25%

Tabela A.15: Desempenho de uma *Cache* com capacidade para 2k instruções, Associatividade 2 e linhas com capacidade para 2 instruções utilizando o algoritmo de reorganização do código Alg-3.

Programas	Miss Original	Miss Alg - 1	Ganho/Perda
Compress	10.384	4.719	54,56%
Gcc	84.346.050	75.456.098	10,54%
Go	1.018.653.571	918.076.521	9,87%
Ijpeg	1.245.835	658.264	47,16%
Li	7.693.724	2.964.131	61,47%
M88ksim	16.442.294	10.245.507	37,69%
Perl	87.755.739	48.260.307	45,01%
Vortex	481.692.142	465.894.214	3,28%

Tabela A.16: Desempenho de uma *Cache* com capacidade para 2k instruções, Associatividade 2 e linhas com capacidade para 4 instruções utilizando o algoritmo de reorganização do código Alg-1.

Programas	Miss Original	Miss Alg - 2	Ganho/Perda
Compress	10.384	912	91,22%
Gcc	84.346.050	85.702.628	-1,61%
Go	1.018.653.571	1.055.418.167	-3,61%
Ijpeg	1.245.835	857.954	31,13%
Li	7.693.724	93.002	98,79%
M88ksim	16.442.294	19.254	99,88%
Perl	87.755.739	10.269.578	88,30%
Vortex	481.692.142	256.499.283	46,75%

Tabela A.17: Desempenho de uma *Cache* com capacidade para 2k instruções, Associatividade 2 e linhas com capacidade para 4 instruções utilizando o algoritmo de reorganização do código Alg-2.

Programas	Miss Original	Miss Alg - 3	Ganho/Perda
Compress	10.384	972	90,64%
Gcc	84.346.050	118.317.481	-40,28%
Go	1.018.653.571	1.321.300.881	-29,71%
Ijpeg	1.245.835	1.405.345	-12,80%
Li	7.693.724	98.748	98,72%
M88ksim	16.442.294	32.772	99,80%
Perl	87.755.739	18.227.241	79,23%
Vortex	481.692.142	405.828.251	15,75%

Tabela A.18: Desempenho de uma *Cache* com capacidade para 2k instruções, Associatividade 2 e linhas com capacidade para 4 instruções utilizando o algoritmo de reorganização do código Alg-3.

Programas	Miss Original	Miss Alg - 1	Ganho/Perda
Compress	14.690	6.753	54,03%
Gcc	221.672.870	215.310.743	2,87%
Go	3.344.351.221	3.313.922.420	0,91%
Ijpeg	2.386.261	2.193.619	8,07%
Li	6.917.869	1.186.166	82,85%
M88ksim	1.005.290	818.740	18,56%
Perl	123.599.702	136.126.677	-10,14%
Vortex	1.110.409.422	1.420.881.634	-27,96%

Tabela A.19: Desempenho de uma *Cache* com capacidade para 2k instruções, Associatividade 4 e linhas com capacidade para 1 instrução utilizando o algoritmo de reorganização do código Alg-1.

Programas	Miss Original	Miss Alg - 2	Ganho/Perda
Compress	14.690	3.604	75,47%
Gcc	221.672.870	219.776.376	0,86%
Go	3.344.351.221	3.294.030.019	1,50%
Ijpeg	2.386.261	2.567.944	-7,61%
Li	6.917.869	295.811	95,72%
M88ksim	1.005.290	50.989	94,93%
Perl	123.599.702	55.853.320	54,81%
Vortex	1.110.409.422	919.208.172	17,22%

Tabela A.20: Desempenho de uma *Cache* com capacidade para 2k instruções, Associatividade 4 e linhas com capacidade para 1 instrução utilizando o algoritmo de reorganização do código Alg-2.

Programas	Miss Original	Miss Alg - 3	Ganho/Perda
Compress	14.690	3.588	75,58%
Gcc	221.672.870	224.387.628	-1,22%
Go	3.344.351.221	3.331.375.801	0,39%
Ijpeg	2.386.261	2.746.374	-15,09%
Li	6.917.869	293.388	95,76%
M88ksim	1.005.290	75.106	92,53%
Perl	123.599.702	51.765.297	58,12%
Vortex	1.110.409.422	1.027.719.661	7,45%

Tabela A.21: Desempenho de uma *Cache* com capacidade para 2k instruções, Associatividade 4 e linhas com capacidade para 1 instrução utilizando o algoritmo de reorganização do código Alg-3.

Programas	Miss Original	Miss Alg - 1	Ganho/Perda
Compress	9.358	3.795	59,45%
Gcc	128.466.976	122.075.950	4,97%
Go	1.765.836.131	1.746.732.405	1,08%
Ijpeg	1.320.277	1.115.908	15,48%
Li	4.284.829	703.449	83,58%
M88ksim	814.254	496.037	39,08%
Perl	81.126.114	78.264.588	3,53%
Vortex	630.370.463	767.499.049	-21,75%

Tabela A.22: Desempenho de uma *Cache* com capacidade para 2k instruções, Associatividade 4 e linhas com capacidade para 2 instruções utilizando o algoritmo de reorganização do código Alg-1.

Programas	Miss Original	Miss Alg - 2	Ganho/Perda
Compress	9.358	1.813	80,63%
Gcc	128.466.976	128.012.400	0,35%
Go	1.765.836.131	1.800.738.381	-1,98%
Ijpeg	1.320.277	1.336.554	-1,23%
Li	4.284.829	154.383	96,40%
M88ksim	814.254	27.742	96,59%
Perl	81.126.114	29.708.006	63,38%
Vortex	630.370.463	471.968.691	25,13%

Tabela A.23: Desempenho de uma *Cache* com capacidade para 2k instruções, Associatividade 4 e linhas com capacidade para 2 instruções utilizando o algoritmo de reorganização do código Alg-2.

Programas	Miss Original	Miss Alg - 3	Ganho/Perda
Compress	9.358	1.853	80,20%
Gcc	128.466.976	148.837.260	-15,86%
Go	1.765.836.131	1.965.061.182	-11,28%
Ijpeg	1.320.277	1.583.996	-19,97%
Li	4.284.829	166.373	96,12%
M88ksim	814.254	50.208	93,83%
Perl	81.126.114	29.326.169	63,85%
Vortex	630.370.463	583.357.349	7,46%

Tabela A.24: Desempenho de uma *Cache* com capacidade para 2k instruções, Associatividade 4 e linhas com capacidade para 2 instruções utilizando o algoritmo de reorganização do código Alg-3.

Programas	Miss Original	Miss Alg - 1	Ganho/Perda
Compress	7.139	2.674	62,54%
Gcc	80.862.235	73.922.419	8,58%
Go	967.416.626	947.730.910	2,03%
Ijpeg	780.231	575.500	26,24%
Li	3.619.681	457.206	87,37%
M88ksim	1.037.873	520.728	49,83%
Perl	57.712.290	48.081.631	16,69%
Vortex	401.549.049	419.015.079	-4,35%

Tabela A.25: Desempenho de uma *Cache* com capacidade para 2k instruções, Associatividade 4 e linhas com capacidade para 4 instruções utilizando o algoritmo de reorganização do código Alg-1.

Programas	Miss Original	Miss Alg - 2	Ganho/Perda
Compress	7.139	922	87,09%
Gcc	80.862.235	81.939.539	-1,33%
Go	967.416.626	1.047.949.074	-8,32%
Ijpeg	780.231	716.905	8,12%
Li	3.619.681	81.574	97,75%
M88ksim	1.037.873	16.189	98,44%
Perl	57.712.290	16.756.359	70,97%
Vortex	401.549.049	248.901.719	38,01%

Tabela A.26: Desempenho de uma *Cache* com capacidade para 2k instruções, Associatividade 4 e linhas com capacidade para 4 instruções utilizando o algoritmo de reorganização do código Alg-2.

Programas	Miss Original	Miss Alg - 3	Ganho/Perda
Compress	7.139	977	86,31%
Gcc	80.862.235	112.964.448	-39,70%
Go	967.416.626	1.295.890.989	-33,95%
Ijpeg	780.231	985.900	-26,36%
Li	3.619.681	111.843	96,91%
M88ksim	1.037.873	36.326	96,50%
Perl	57.712.290	18.216.582	68,44%
Vortex	401.549.049	369.455.516	7,99%

Tabela A.27: Desempenho de uma *Cache* com capacidade para 2k instruções, Associatividade 4 e linhas com capacidade para 4 instruções utilizando o algoritmo de reorganização do código Alg-3.

Programas	Média Alg - 1	Média Alg - 2	Média Alg - 3
Compress	48,91%	87,85%	87,52%
Gcc	7,18%	-0,66%	-20,19%
Go	8,78%	5,71%	-8,15%
Ijpeg	38,64%	16,31%	-6,68%
Li	57,27%	98,05%	98,07%
M88ksim	97,98%	98,79%	97,98%
Perl	75,18%	78,08%	67,23%
Vortex	-8,56%	36,24%	14,90%
Média	40,67%	52,55%	41,33%

Tabela A.28: Média dos desempenhos em uma *cache* com capacidade para 2k instruções

Apêndice B

Resultados dos Experimentos para uma *Cache* com Capacidade para 10% do Tamanho do Código

Este apêndice apresenta os resultados das simulações realizadas considerando *caches* com capacidade para 10% do total de instruções dos programas. A disposição destes resultados está na forma de tabelas. Cada tabela representa a comparação de um dos algoritmos propostos e a simulação com o código na sua organização original. Para cada *cache* utilizada temos três tabelas, uma para cada algoritmo. Para facilitar a visualização, foram dispostas três tabelas por página, cada uma de um algoritmo diferente, tendo em cada página todos os resultados para cada tipo de *cache*.

Programas	Miss Original	Miss Alg - 1	Ganho/Perda
Compress	66.922	73.683	-10,10%
Gcc	21.443.461	11.738.154	45,26%
Go	1.762.604.679	1.179.397.532	33,09%
Ijpeg	1.987.665	1.801.744	9,35%
Li	24.873.179	44.120.400	-77,38%
M88ksim	35.666.955	40.124.789	-12,50%
Perl	107.242.808	71.641.770	33,20%
Vortex	354.551.56	435.709.189	-22,89%

Tabela B.1: Desempenho de uma *Cache* com capacidade para 10% do tamanho do codigo, Mapeamento Direto e linhas com capacidade para 1 instrução utilizando o algoritmo de reorganização do código Alg-1.

Programas	Miss Original	Miss Alg - 2	Ganho/Perda
Compress	66.922	21.409	68,01%
Gcc	21.443.461	17.464.469	18,56%
Go	1.762.604.679	669.908.234	61,99%
Ijpeg	1.987.665	1.854.682	6,69%
Li	24.873.179	82.578	99,67%
M88ksim	35.666.955	25.178	99,93%
Perl	107.242.808	27.184	99,97%
Vortex	354.551.56	96.742.231	78,19%

Tabela B.2: Desempenho de uma *Cache* com capacidade para 10% do tamanho do codigo, Mapeamento Direto e linhas com capacidade para 1 instrução utilizando o algoritmo de reorganização do código Alg-2.

Programas	Miss Original	Miss Alg - 3	Ganho/Perda
Compress	66.922	21.630	67,68%
Gcc	21.443.461	18.734.308	12,63%
Go	1.762.604.679	859.832.639	51,22%
Ijpeg	1.987.665	1.984.626	0,15%
Li	24.873.179	110.070	99,56%
M88ksim	35.666.955	23.972	99,93%
Perl	107.242.808	31.400	99,97%
Vortex	354.551.56	96.742.231	72,71%

Tabela B.3: Desempenho de uma *Cache* com capacidade para 10% do tamanho do codigo, Mapeamento Direto e linhas com capacidade para 1 instrução utilizando o algoritmo de reorganização do código Alg-3.

Programas	Miss Original	Miss Alg - 1	Ganho/Perda
Compress	39.122	38.643	1,22%
Gcc	12.257.429	6.421.857	47,61%
Go	923.811.082	616.188.850	33,30%
Ijpeg	1.067.528	910.914	14,67%
Li	15.072.017	24.052.369	-59,58%
M88ksim	21.451.369	21.297.683	0,72%
Perl	65.493.203	39.314.014	39,97%
Vortex	204.370.001	230.071.159	-12,58%

Tabela B.4: Desempenho de uma *Cache* com capacidade para 10% do tamanho do codigo, Mapeamento Direto e linhas com capacidade para 2 instruções utilizando o algoritmo de reorganização do código Alg-1.

Programas	Miss Original	Miss Alg - 2	Ganho/Perda
Compress	39.122	10.960	71,99%
Gcc	12.257.429	9.995.338	18,45%
Go	923.811.082	370.169.813	59,93%
Ijpeg	1.067.528	946.521	11,34%
Li	15.072.017	43.652	99,71%
M88ksim	21.451.369	13.079	99,94%
Perl	65.493.203	14.280	99,98%
Vortex	204.370.001	40.069.230	80,39%

Tabela B.5: Desempenho de uma *Cache* com capacidade para 10% do tamanho do codigo, Mapeamento Direto e linhas com capacidade para 2 instruções utilizando o algoritmo de reorganização do código Alg-2.

Programas	Miss Original	Miss Alg - 3	Ganho/Perda
Compress	39.122	12.295	68,57%
Gcc	12.257.429	12.560.333	-2,47%
Go	923.811.082	539.883.265	41,56%
Ijpeg	1.067.528	1.062.815	0,44%
Li	15.072.017	70.661	99,53%
M88ksim	21.451.369	14.287	99,93%
Perl	65.493.203	17.856	99,97%
Vortex	204.370.001	58.459.815	71,40%

Tabela B.6: Desempenho de uma *Cache* com capacidade para 10% do tamanho do codigo, Mapeamento Direto e linhas com capacidade para 2 instruções utilizando o algoritmo de reorganização do código Alg-3.

Programas	Miss Original	Miss Alg - 1	Ganho/Perda
Compress	24.880	20.923	15,90%
Gcc	7.813.465	3.652.899	53,25%
Go	514.199.098	333.692.325	35,10%
Ijpeg	597.953	465.813	22,10%
Li	10.626.367	15.163.740	-42,70%
M88ksim	14.220.413	11.507.543	19,08%
Perl	42.294.424	22.217.642	47,47%
Vortex	126.323.033	129.018.723	-2,13%

Tabela B.7: Desempenho de uma *Cache* com capacidade para 10% do tamanho do codigo, Mapeamento Direto e linhas com capacidade para 4 instruções utilizando o algoritmo de reorganização do código Alg-1.

Programas	Miss Original	Miss Alg - 2	Ganho/Perda
Compress	24.880	5.737	76,94%
Gcc	7.813.465	6.286.925	19,54%
Go	514.199.098	222.987.731	56,63%
Ijpeg	597.953	494.467	17,31%
Li	10.626.367	24.061	99,77%
M88ksim	14.220.413	7.028	99,95%
Perl	42.294.424	7.786	99,98%
Vortex	126.323.033	21.520.578	82,96%

Tabela B.8: Desempenho de uma *Cache* com capacidade para 10% do tamanho do codigo, Mapeamento Direto e linhas com capacidade para 4 instruções utilizando o algoritmo de reorganização do código Alg-2.

Programas	Miss Original	Miss Alg - 3	Ganho/Perda
Compress	24.880	7.630	69,33%
Gcc	7.813.465	9.937.296	-27,18%
Go	514.199.098	391.955.772	23.77%
Ijpeg	597.953	620.223	-3,72%
Li	10.626.367	51.309	99,52%
M88ksim	14.220.413	9.122	99,94%
Perl	42.294.424	11.737	99,97%
Vortex	126.323.033	39.470.283	68,75%

Tabela B.9: Desempenho de uma *Cache* com capacidade para 10% do tamanho do codigo, Mapeamento Direto e linhas com capacidade para 4 instruções utilizando o algoritmo de reorganização do código Alg-3.

Programas	Miss Original	Miss Alg - 1	Ganho/Perda
Compress	63.416	60.195	5,08%
Gcc	8.936.866	8.358.830	6,47%
Go	907.481.512	727.062.026	19,88%
Ijpeg	1.816.919	1.810.161	0,37%
Li	5.318.993	2.459.124	53,77%
M88ksim	8.202.693	36.950.127	-350,46%
Perl	13.194.982	34.873.309	-164,29%
Vortex	167.523.439	128.628.852	23,22%

Tabela B.10: Desempenho de uma *Cache* com capacidade para 10% do tamanho do código, Associatividade 2 e linhas com capacidade para 1 instrução utilizando o algoritmo de reorganização do código Alg-1.

Programas	Miss Original	Miss Alg - 2	Ganho/Perda
Compress	63.416	28.894	54,44%
Gcc	8.936.866	9.148.104	-2,36%
Go	907.481.512	553.904.485	38,96%
Ijpeg	1.816.919	1.928.777	-6,16%
Li	5.318.993	75.161	98,59%
M88ksim	8.202.693	16.577	99,80%
Perl	13.194.982	25.413	99,81%
Vortex	167.523.439	47.996.886	71,35%

Tabela B.11: Desempenho de uma *Cache* com capacidade para 10% do tamanho do código, Associatividade 2 e linhas com capacidade para 1 instrução utilizando o algoritmo de reorganização do código Alg-2.

Programas	Miss Original	Miss Alg - 3	Ganho/Perda
Compress	63.416	29.508	53,47%
Gcc	8.936.866	9.505.155	-6,36%
Go	907.481.512	602.472.319	33,61%
Ijpeg	1.816.919	1.829.372	-0,69%
Li	5.318.993	119.854	97,75%
M88ksim	8.202.693	19.330	99,76%
Perl	13.194.982	27.516	99,79%
Vortex	167.523.439	63.083.094	62,34%

Tabela B.12: Desempenho de uma *Cache* com capacidade para 10% do tamanho do código, Associatividade 2 e linhas com capacidade para 1 instrução utilizando o algoritmo de reorganização do código Alg-3.

Programas	Miss Original	Miss Alg - 1	Ganho/Perda
Compress	38.011	32.347	14,90%
Gcc	5.022.559	4.456.096	11,28%
Go	479.811.453	385.364.326	19,68%
Ijpeg	980.379	912.338	6,94%
Li	3.271.991	1.399.873	57,22%
M88ksim	4.847.021	20.098.456	-314,66%
Perl	10.717.984	19.383.855	-80,85%
Vortex	96.435.137	67.642.708	29,86%

Tabela B.13: Desempenho de uma *Cache* com capacidade para 10% do tamanho do codigo, Associatividade 2 e linhas com capacidade para 2 instruções utilizando o algoritmo de reorganização do código Alg-1.

Programas	Miss Original	Miss Alg - 2	Ganho/Perda
Compress	38.011	14.812	61,03%
Gcc	5.022.559	5.153.197	-2,60%
Go	479.811.453	305.047.896	36,42%
Ijpeg	980.379	982.138	-0,18%
Li	3.271.991	38.517	98,82%
M88ksim	4.847.021	8.713	99,82%
Perl	10.717.984	13.228	99,88%
Vortex	96.435.137	24.743.789	74,34%

Tabela B.14: Desempenho de uma *Cache* com capacidade para 10% do tamanho do codigo, Associatividade 2 e linhas com capacidade para 2 instruções utilizando o algoritmo de reorganização do código Alg-2.

Programas	Miss Original	Miss Alg - 3	Ganho/Perda
Compress	38.011	16.195	57,39%
Gcc	5.022.559	6.099.233	-21,44%
Go	479.811.453	376.099.469	21,62%
Ijpeg	980.379	964.419	1,63%
Li	3.271.991	77.829	97,62%
M88ksim	4.847.021	11.255	99,77%
Perl	10.717.984	15.311	99,86%
Vortex	96.435.137	36.884.250	61,75%

Tabela B.15: Desempenho de uma *Cache* com capacidade para 10% do tamanho do codigo, Associatividade 2 e linhas com capacidade para 2 instruções utilizando o algoritmo de reorganização do código Alg-3.

Programas	Miss Original	Miss Alg - 1	Ganho/Perda
Compress	26.025	18.236	29,93%
Gcc	3.019.267	2.477.439	17,95%
Go	269.774.863	210.000.163	22,16%
Ijpeg	548.046	463.754	15,38%
Li	2.360.216	783.051	66,82%
M88ksim	2.902.426	10.849.170	-273,80%
Perl	9.322.697	11.617.673	-24,62%
Vortex	62.376.387	36.614.469	41,30%

Tabela B.16: Desempenho de uma *Cache* com capacidade para 10% do tamanho do codigo, Associatividade 2 e linhas com capacidade para 4 instruções utilizando o algoritmo de reorganização do código Alg-1.

Programas	Miss Original	Miss Alg - 2	Ganho/Perda
Compress	26.025	7.622	70,71%
Gcc	3.019.267	3.227.847	-6,91%
Go	269.774.863	183.577.212	31,95%
Ijpeg	548.046	510.457	6,86%
Li	2.360.216	20.301	99,14%
M88ksim	2.902.426	4.747	99,84%
Perl	9.322.697	7.153	99,92%
Vortex	62.376.387	13.226.564	78,80%

Tabela B.17: Desempenho de uma *Cache* com capacidade para 10% do tamanho do codigo, Associatividade 2 e linhas com capacidade para 4 instruções utilizando o algoritmo de reorganização do código Alg-2.

Programas	Miss Original	Miss Alg - 3	Ganho/Perda
Compress	26.025	9.633	62,99%
Gcc	3.019.267	4.794.628	-58,80%
Go	269.774.863	274.340.299	-1,69%
Ijpeg	548.046	535.499	2,29%
Li	2.360.216	53.991	97,71%
M88ksim	2.902.426	7.198	99,75%
Perl	9.322.697	9.848	99,89%
Vortex	62.376.387	24.576.842	60,60%

Tabela B.18: Desempenho de uma *Cache* com capacidade para 10% do tamanho do codigo, Associatividade 2 e linhas com capacidade para 4 instruções utilizando o algoritmo de reorganização do código Alg-3.

Programas	Miss Original	Miss Alg - 1	Ganho/Perda
Compress	55.271	46.036	16,71%
Gcc	7.173.533	7.075.248	1,37%
Go	413.924.138	428.743.088	-3,58%
Ijpeg	1.810.227	1.846.539	-2,01%
Li	1.255.717	239.587	80,92%
M88ksim	23.718	18.150	23,48%
Perl	176.125	234.394	-33,08%
Vortex	89.795.517	80.320.844	10,55%

Tabela B.19: Desempenho de uma *Cache* com capacidade para 10% do tamanho do codigo, Associatividade 4 e linhas com capacidade para 1 instrução utilizando o algoritmo de reorganização do código Alg-1.

Programas	Miss Original	Miss Alg - 2	Ganho/Perda
Compress	55.271	39.860	27,88%
Gcc	7.173.533	7.182.256	-0,12%
Go	413.924.138	428.395.062	-3,50%
Ijpeg	1.810.227	1.835.439	-1,39%
Li	1.255.717	116.347	90,73%
M88ksim	23.718	15.162	36,07%
Perl	176.125	24.501	86,09%
Vortex	89.795.517	37.421.287	58,33%

Tabela B.20: Desempenho de uma *Cache* com capacidade para 10% do tamanho do codigo, Associatividade 4 e linhas com capacidade para 1 instrução utilizando o algoritmo de reorganização do código Alg-2.

Programas	Miss Original	Miss Alg - 3	Ganho/Perda
Compress	55.271	38.712	29,96%
Gcc	7.173.533	7.285.890	-1,57%
Go	413.924.138	480.632.197	-16,12%
Ijpeg	1.810.227	1.818.882	-0,48%
Li	1.255.717	140.465	88,81%
M88ksim	23.718	15.610	34,19%
Perl	176.125	24.966	85,82%
Vortex	89.795.517	41.768.412	53,48%

Tabela B.21: Desempenho de uma *Cache* com capacidade para 10% do tamanho do codigo, Associatividade 4 e linhas com capacidade para 1 instrução utilizando o algoritmo de reorganização do código Alg-3.

Programas	Miss Original	Miss Alg - 1	Ganho/Perda
Compress	34.805	24.874	28,53%
Gcc	3.958.191	3.706.939	6,35%
Go	217.026.391	224.935.867	-3,64%
Ijpeg	971.780	930.932	4,20%
Li	798.075	135.900	82,97%
M88ksim	13.878	9.454	31,88%
Perl	123.325	129.997	-5,41%
Vortex	54.072.177	42.771.590	20,90%

Tabela B.22: Desempenho de uma *Cache* com capacidade para 10% do tamanho do codigo, Associatividade 4 e linhas com capacidade para 2 instruções utilizando o algoritmo de reorganização do código Alg-1.

Programas	Miss Original	Miss Alg - 2	Ganho/Perda
Compress	34.805	20.284	41,72%
Gcc	3.958.191	3.898.307	1,51%
Go	217.026.391	237.180.349	-9,29%
Ijpeg	971.780	931.683	4,13%
Li	798.075	58.997	92,61%
M88ksim	13.878	7.942	42,77%
Perl	123.325	12.710	89,69%
Vortex	54.072.177	19.403.330	64,12%

Tabela B.23: Desempenho de uma *Cache* com capacidade para 10% do tamanho do codigo, Associatividade 4 e linhas com capacidade para 2 instruções utilizando o algoritmo de reorganização do código Alg-2.

Programas	Miss Original	Miss Alg - 3	Ganho/Perda
Compress	34.805	20.409	41,36%
Gcc	3.958.191	4.378.419	-10,62%
Go	217.026.391	300.184.611	-38,32%
Ijpeg	971.780	945.698	2,68%
Li	798.075	84.848	89,37%
M88ksim	13.878	8.864	36,13%
Perl	123.325	13.807	88,80%
Vortex	54.072.177	24.597.750	54,51%

Tabela B.24: Desempenho de uma *Cache* com capacidade para 10% do tamanho do codigo, Associatividade 4 e linhas com capacidade para 2 instruções utilizando o algoritmo de reorganização do código Alg-3.

Programas	Miss Original	Miss Alg - 1	Ganho/Perda
Compress	25.153	14.991	40,40%
Gcc	2.333.919	2.009.706	13,89%
Go	119.111.874	121.559.828	-2,06%
Ijpeg	539.192	473.102	12,26%
Li	561.253	87.114	84,48%
M88ksim	9.324	5.161	44,65%
Perl	88.816	88.382	0,49%
Vortex	37.025.233	24.046.842	35,05%

Tabela B.25: Desempenho de uma *Cache* com capacidade para 10% do tamanho do codigo, Associatividade 4 e linhas com capacidade para 4 instruções utilizando o algoritmo de reorganização do código Alg-1.

Programas	Miss Original	Miss Alg - 2	Ganho/Perda
Compress	25.153	10.461	58,41%
Gcc	2.333.919	2.302.953	1,33%
Go	119.111.874	145.226.303	-21,92%
Ijpeg	539.192	480.383	10,91%
Li	561.253	30.793	94,51%
M88ksim	9.324	4.304	53,84%
Perl	88.816	6.830	92,31%
Vortex	37.025.233	10.548.415	71,51%

Tabela B.26: Desempenho de uma *Cache* com capacidade para 10% do tamanho do codigo, Associatividade 4 e linhas com capacidade para 4 instruções utilizando o algoritmo de reorganização do código Alg-2.

Programas	Miss Original	Miss Alg - 3	Ganho/Perda
Compress	25.153	12.187	51,55%
Gcc	2.333.919	3.178.011	-36,17%
Go	119.111.874	225.026.752	-88,92%
Ijpeg	539.192	511.305	5,17%
Li	561.253	46.973	91,63%
M88ksim	9.324	5.595	39,99%
Perl	88.816	8.562	90,36%
Vortex	37.025.233	16.665.827	54,99%

Tabela B.27: Desempenho de uma *Cache* com capacidade para 10% do tamanho do codigo, Associatividade 4 e linhas com capacidade para 4 instruções utilizando o algoritmo de reorganização do código Alg-3.

Programas	Média Alg - 1	Média Alg - 2	Média Alg - 3
Compress	15,84%	59,01%	55,81%
Gcc	22,60%	5,27%	-16,89%
Go	17,10%	27,91%	2,97%
Ijpeg	9,25%	5,50%	0,83%
Li	34,59%	87,90%	95,72%
M88ksim	-92,40%	81,33%	78,82%
Perl	-20,79%	96,40%	96,05%
Vortex	13,70%	73,33%	62,28%
Média	-0,01%	54,58%	46,95%

Tabela B.28: Média dos desempenhos em uma *cache* com capacidade para 10% do tamanho do codigo

Apêndice C

Resultados dos Experimentos para uma *Cache* com Capacidade para 5% do Total de Instruções Executadas pelo menos 1 Vez

Este apêndice apresenta os resultados das simulações realizadas considerando *caches* com capacidade para 5% do total de instruções executadas pelo menos 1 vez. A disposição destes resultados está na forma de tabelas. Cada tabela representa a comparação de um dos algoritmos propostos e a simulação com o código na sua organização original. Para cada *cache* utilizada temos três tabelas, uma para cada algoritmo. Para facilitar a visualização, foram dispostas três tabelas por página, cada uma de um algoritmo diferente, tendo em cada página todos os resultados para cada tipo de *cache*.

Programas	Miss Original	Miss Alg - 1	Ganho/Perda
Compress	184.118	169.893	7,73%
Gcc	105.603.090	105.416.377	0,18%
Go	3.196.620.731	3.111.304.407	2,67%
Ijpeg	18.892.632	14.169.248	25,00%
Li	302.127.086	315.636.339	-4,47%
M88ksim	223.933.474	198.596.925	11,31%
Perl	525.685.468	481.619.729	8,38%
Vortex	2.137.582.725	1.011.182.409	52,70%

Tabela C.1: Desempenho de uma *Cache* com capacidade para 5% do total de instruções executadas pelo menos 1 vez, Mapeamento Direto e linhas com capacidade para 1 instrução utilizando o algoritmo de reorganização do código Alg-1.

Programas	Miss Original	Miss Alg - 2	Ganho/Perda
Compress	184.118	219.775	-19,37%
Gcc	105.603.090	95.919.950	9,17%
Go	3.196.620.731	2.375.514.036	25,69%
Ijpeg	18.892.632	25.791.697	-36,52%
Li	302.127.086	265.466.988	12,13%
M88ksim	223.933.474	150.012.095	33,01%
Perl	525.685.468	389.675.696	25,87%
Vortex	2.137.582.725	743.849.076	65,20%

Tabela C.2: Desempenho de uma *Cache* com capacidade para 5% do total de instruções executadas pelo menos 1 vez, Mapeamento Direto e linhas com capacidade para 1 instrução utilizando o algoritmo de reorganização do código Alg-2.

Programas	Miss Original	Miss Alg - 3	Ganho/Perda
Compress	184.118	203.782	-10,68%
Gcc	105.603.090	119.375.236	-13,04%
Go	3.196.620.731	2.630.206.383	17,72%
Ijpeg	18.892.632	25.789.905	-36,51%
Li	302.127.086	266.524.615	11,78%
M88ksim	223.933.474	194.943.415	12,95%
Perl	525.685.468	464.018.111	11,73%
Vortex	2.137.582.725	920.849.039	56,92%

Tabela C.3: Desempenho de uma *Cache* com capacidade para 5% do total de instruções executadas pelo menos 1 vez, Mapeamento Direto e linhas com capacidade para 1 instrução utilizando o algoritmo de reorganização do código Alg-3.

Programas	Miss Original	Miss Alg - 1	Ganho/Perda
Compress	69.688	51.169	26,57%
Gcc	39.067.564	35.954.003	7,97%
Go	953.235.734	921.107.559	3,37%
Ijpeg	5.699.152	3.988.920	30,01%
Li	104.607.992	108.762.675	-3,97%
M88ksim	73.276.096	65.667.686	10,38%
Perl	188.892.470	159.150.602	15,75%
Vortex	718.657.585	290.783.157	59,54%

Tabela C.4: Desempenho de uma *Cache* com capacidade para 5% do total de instruções executadas pelo menos 1 vez, Mapeamento Direto e linhas com capacidade para 4 instruções utilizando o algoritmo de reorganização do código Alg-1.

Programas	Miss Original	Miss Alg - 2	Ganho/Perda
Compress	69.688	66.218	4,98%
Gcc	39.067.564	36.590.814	6,34%
Go	953.235.734	768.019.065	19,43%
Ijpeg	5.699.152	6.109.301	-7,20%
Li	104.607.992	90.220.861	13,75%
M88ksim	73.276.096	44.493.949	39,28%
Perl	188.892.470	107.245.752	43,22%
Vortex	718.657.585	205.149.773	71,45%

Tabela C.5: Desempenho de uma *Cache* com capacidade para 5% do total de instruções executadas pelo menos 1 vez, Mapeamento Direto e linhas com capacidade para 4 instruções utilizando o algoritmo de reorganização do código Alg-2.

Programas	Miss Original	Miss Alg - 3	Ganho/Perda
Compress	69.688	84.958	-21,91%
Gcc	39.067.564	62.340.350	-59,57%
Go	953.235.734	1.079.048.615	-13,20%
Ijpeg	5.699.152	10.197.156	-78,92%
Li	104.607.992	110.630.101	-5,76%
M88ksim	73.276.096	67.312.750	8,14%
Perl	188.892.470	186.125.987	1,46%
Vortex	718.657.585	339.629.996	52,74%

Tabela C.6: Desempenho de uma *Cache* com capacidade para 5% do total de instruções executadas pelo menos 1 vez, Mapeamento Direto e linhas com capacidade para 4 instruções utilizando o algoritmo de reorganização do código Alg-3.

Programas	Média Alg - 1	Média Alg - 2	Média Alg - 3
Compress	17,15%	-7,20%	-16,30%
Gcc	4,08%	7,76%	-36,31%
Go	3,02%	22,56%	2,26%
Ijpeg	27,51%	14,66%	-21,21%
Li	-4,22%	12,94%	3,01%
M88ksim	10,85%	36,15%	10,55%
Perl	12,07%	34,55%	6,60%
Vortex	56,12%	68,33%	54,83%
Média	15,82%	23,72%	0,43%

Tabela C.7: Média dos desempenhos em uma *cache* com capacidade para 5% do total de instruções executadas pelo menos 1 vez

Referências Bibliográficas

[1] Anant Agarwal and Steven D. Pudar. "Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches". In *13th Annual International Symposium on Computer Architecture*, June 1986.

[2] Thomas Ball and James R. Larus. "Efficient path Profiling". In *Proceedings of the 29th annual ACM/IEEE International Symposium on Microarchitecture*, pages 46–57, 1996.

[3] Doug Burger and Todd M. Austin. "The Simplescalar Tool Set, version 2.0". *ACM SIGARCH Computer Architecture News*, 25(3):13–25, 1997.

[4] Peter J. Denning. "Virtual Memory". *ACM Computing Surveys (CSUR)*, 2(3):153–189, 1970.

[5] Peter J. Denning and Stuart C. Schwartz. "Properties of the Working-Set Model". *Communications of the ACM*, 15(3):191–198, 1972.

[6] Edil S. T. Fernandes e Gabriel Pereira da Silva. "BBM - Um Processador de Blocos Básicos". *Anais do I Workshop em Sistemas Computacionais de Alto Desempenho - WSCAD'00, São Pedro - SP*, 25 a 27 de Outubro de 2000, pp. 3-8.

[7] Edil S. T. Fernandes e Valmir C. Barbosa. "Monitoring the Structure and Behavior of Programs". Proceedings of the 4th International Conference on Massively Parallel Computing Systems MPCS'02, Ischia, Italy ISBN: 0-9669530-0-2 Published by the National Technological University Press, 10-12/April/2002, 6 pages.

[8] Domenicó Ferrari, "Improving Locality by Critical Working Sets", *Communications of the ACM*, Vol 17. No. 11, November 1974, pp. 614–620.

[9] A. Hashemi, D. Kaeli, and B. Calder. "Procedure Mapping Using Static Call Graph Estimation". In *Proceedings of the Workshop on Interaction between Compiler and Computer Architecture*, February 1997.

[10] John L. Hennessy and David a Patterson. "*Computer Architecture a Quantitative Approach*". Morgan Kaufmann, San Francisco, California, second edition, 1996.

[11] Norman P. Jouppi. "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers". In *Proceedings of the 17th annual International Symposium on Computer Architecture*, pages 364–373. ACM Press, 1990.

[12] John Kalamatianos and David R. Kaeli. "Temporal-Based Procedure Reordering for Improved Instruction Cache Performance". *HPCA-4*, pages 1–10, Feb 1998.

[13] D. E. Knuth. "An Empirical Study of FORTRAN programs". *Software — Practice & Experience*, Vol. 1, 1971, pp. 105–133.

[14] Karl Pettis and Robert C. Hansen. "Profile Guided Code Positioning". pages 16–27. ACM Press, June 1990.

[15] Alex Ramirez, Luiz André Barroso, Kourosh Gharachorloo, Robert Cohn, Josep Larriba-Pey, P. Geoffrey Lowney, and Mateo Valero. "Code Layout Optimizations for Transaction Processing Workloads". In *Proceedings of the 28th annual International Symposium on Computer Architecture*, pages 155–164. ACM Press, 2001.

[16] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. "Disassembly of Executable Code Revisited". *Proceedings 2002 Working Conferece on Reverse Engineering*, OCT 2002.

[17] Maurice V. Wilkes. "The Memory Gap and the Future of High Performance Memories". *ACM SIGARCH Computer Architecture News*, 29(1):2–7, 2001.