Estudo de Arquiteturas de Memórias Cache Distribuídas para o Sistema MULTIPLUS

Alexandre Malheiros Meslin

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. Ageu Cavalcanti Pacheco Junior, PhD
PRESIDENTE

Prof. Edil Severiano Tavares Fernandes, PhD

Prof. Júlio Salek Aude, PhD

Prof Adriano Joaquim de Oliveira Cruz, PhD

RIO DE JANEIRO, RJ - BRASIL

AGOSTO DE 1991

MESLIN, ALEXANDRE MALHEIROS

Estudo de Arquiteturas de Memórias *Cache* para o Sistema MULTIPLUS [Rio de Janeiro] 1991.

ix, 100 pp, 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1991)

Tese - Universidade Federal do Rio de Janeiro, COPPE.

1 - Memória *Cache* e Arquitetura de Computadores

I. COPPE/UFRJ II. Título (série)

Para minha mãe Sônia, minha avó Ruth e minha esposa Denise

AGRADECIMENTOS

Agradeço ao Professor Ageu Pacheco Cavalvanti Junior o apoio e orientação recebidos.

Ao professor Júlio Salek Aude pelas valiosas sugestões em toda a tese, e, principalmente, durante as simulações.

Ao NCE/UFRJ pela utilização dos seus laboratórios.

Ao CNPQ e FINEP pelo apoio financeiro ao Projeto MULTIPLUS que originou esta tese.

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

ESTUDO DE ARQUITETURAS DE MEMÓRIAS *CACHE* DISTRIBUIDAS PARA O SISTEMA MULTIPLUS

Alexandre Malheiros Meslin

Abril, 1991

Orientador: Ageu Cavalcante Pacheco Júnior
Programa: Engenharia de Sistemas e Computação

RESUMO

Este trabalho apresenta as considerações iniciais, os problemas e as limitações de um projeto de memórias *cache* para um sistema do tipo multiprocessador conectado através de uma rede de interconexão, onde as terminações são barramentos paralelos com diversas *CPU*'s. São também apresentados um pequeno histórico de memórias *cache*, a análise de esquemas de manutenção de coerência e ainda algumas arquiteturas de *cache* possíveis de serem implementadas. A seguir, são discutidos esquemas de atualização da memória principal e de manutenção da coerência aplicáveis ao Sistema MULTIPLUS. Por fim, as diversas arquiteturas propostas são simuladas para uma efetiva comparação dos seus desempenhos.

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

STUDY OF ARCHITECTURES OF DISTRIBUTED CACHE MEMORIES FOR THE MULTIPLUS SYSTEM

Alexandre Malheiros Meslin

April, 1991

Chairman: Ageu Cavalcante Pacheco Júnior
Department: Engenharia de Sistemas e Computação

ABSTRACT

This work presents some initial considerations, problems and limitations of a multicache memory project for a multiprocessor system coupled by an interconnection network and parallel buses. A brief review of cache memories, an analysis of data coherence schemes and some possible cache architectures are also presented. Main memory update and coherence maintenance schemes applied to the MULTIPLUS system are discussed. Finally, some architectures are simulated for performance comparisons.

ÍNDICE

CAPÍTULO I

INTRODUÇÃO

Uma memória *cache* é uma unidade de armazenamento geralmente pequena e muito rápida situada entre o processador e a memória principal que contém cópias de certas posições específicas desta. Cada entrada na *cache* contém um dado e o seu endereço (ou parte dele) associado, além de um campo de estado deste dado armazenado (*tag*). O campo de endereço e de estado é chamado de rótulo (*tag*). Cada acesso à memória é mapeado em uma entrada da *cache*. A comparação do endereço armazenado na *cache* com o do acesso corrente determinará se existe um acerto (*hit*) ou falha (miss). Todas as memórias *caches* necessitam de um *bit* por bloco sinalizando se o dado presente é válido ou não. Geralmente, invalidações na memória *cache* ocorrem na troca de processos ou após a carga inicial do sistema. O endereço que chega à memória *cache* pode ser tanto o endereço virtual quanto o real, dependendo da forma de implementação.

Visando aumentar a capacidade de processamento dos computadores (ainda monoprocessados), observou-se que, mesmo com a evolução da tecnologia, as unidades de processamento (*PE - processing element*) necessitavam informações a uma taxa (ciclo de *CPU*) que crescia mais rapidamente do que a das memórias convencionais em fornecer estas informações (ciclo de memória), o que deixava a unidade de processamento muito tempo parada (*idle*). A memória do sistema ganhou, então,a conotação de gargalo de "von Newmann" [STON87]. Observou-se também que a maioria dos programas passava grande parte do seu tempo de execução em trechos de tamanhos reduzidos (quando comparados ao tamanho do programa principal)[1].

A primeira idéia de memória *cache* foi a da colocação de uma memória, entre a *CPU* e a memória principal, de menor capacidade de

---

[1]Princípio da localidade de referência

armazenamento, porém bem mais rápida (5 a 10 vezes) do que a principal. Esta memória forneceria as informações (código e dados) à unidade de processamento a uma taxa que minimizaria o seu tempo *idle.*

A introdução de outras unidades que poderiam acessar a memória principal e o compartilhamento do barramento fez com que o "gargalo", que antes consistia apenas do tempo de acesso à memória principal, passasse a incluir também a disputa pelo barramento entre a unidade de processamento e os controladores de entrada e saída. Uma nova justificativa para as memórias *caches* (que tenderiam a desaparecer, uma vez que o tempo de acesso à memória diminuia com a evolução da tecnologia) foi a redução da taxa de acessos ao barramento.

Com o aparecimento dos computadores multiprocessados, o uso de *caches* tornou-se indispensável. Sua importância maior não reside mais apenas no fato de tentar manter a unidade de processamento com taxa de ocupação total, e sim em possibilitar que, com uma taxa de acerto (*hit rate*) a mais próxima possível de 100%, um maior número de *PE*'s possam compartilhar um mesmo barramento. Modernamente, mesmo uma memória *cache* com tempo de acesso comparável ao da memória principal seria justificável, pois funcionaria como uma espécie de *buffer*, reduzindo o tráfego entre os processadores e a memória principal [KAPL73].

Para melhor situar o problema, seja por exemplo, um sistema multiprocessado, onde os processadores demandam um acesso à memória a cada ciclo, mas não possuem memória *cache*. Supondo um barramento cujo esquema de prioridade de atendimento de acessos seja fixo (pelo número da placa, por exemplo), este sistema comportará idealmente apenas um *PE*, no máximo dois[2], já que um terceiro jamais ganharia o barramento conforme mostrado na figura 1.1.

---

[2]sendo que a cada instante de tempo haverá sempre um *PE* parado (*idle*), não contribuindo para a melhoria do desempenho do sistema

BR*n*- pedido de acesso do processador *n*

BG - reconhecimento ao pedido de acesso de qualquer PE

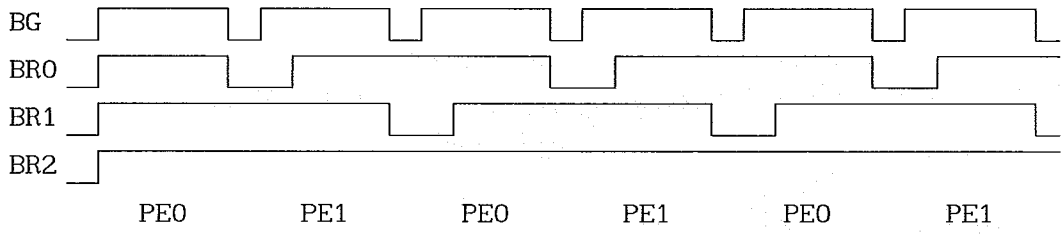


figura 1.1: diagrama de tempos

Este mesmo sistema, com *cache*, com *hit rate* de 95% (que atualmente pode ser considerada baixa) e uma relação entre tempo de acesso à *cache* e à memória principal de um fator de 10, comportaria perfeitamente 4 *PE*'s. O número máximo de *PE*'s em um mesmo barramento pode ser aproximadamente calculado por[3]:

$$N_{PE} \cong \frac{T_{CACHE} * P_{HIT}}{(1 - P_{HIT})\ T_{BUS}} + 1$$

onde: N_{PE} = número de processadores

P_{HIT} = taxa de acerto (*hit rate*)

T_{BUS} = tempo total de acesso ao barramento

T_{CACHE} = tempo total de acesso à *cache*

Para uma taxa de acerto igual para todas as *caches* dos *PE*'s, de P_{HIT} = 98% e um barramento com 100% de ocupação, um sistema com barramento é viável com até oito *PE*'s.

---

[3]desenvolvimento no apêndice D.

I.1 - Motivação

Este trabalho analisa diversas opções de arquitetura de memórias caches possíveis para o Sistema MULTIPLUS [AUDE90] e também suas implicações quanto ao tipo de protocolo de barramento e outros aspectos de arquitetura e sistema operacional. A escolha da arquitetura de cache pode influenciar de forma significativa o desempenho geral do sistema, principalmente considerando o esquema proposto de barramento duplo, como será visto na seção 3.

Sistemas de memória *cache* já foram discutidos em diversos trabalhos, sendo, portanto, a sua influência no desempenho de sistemas de computadores monoprocessados bem conhecidas.

A necessidade de uma simulação para o Sistema MULTIPLUS decorre de diversas inovações propostas para a arquitetura desta máquina, em particular, a co-existência de barramento e rede de interconexão ligando os diversos processadores e a sua arquitetura de memória com uma forte e bem definida hierarquia, onde cada processador possui uma memória local pertencente ao espaço de endereçamento global, além de permitir que variáveis compartilhadas sejam armazenadas em mais de uma *cache* privada.

Outro fator preponderante é a raridade de simulações de sistemas baseados em processadores de arquitetura RISC.

I.2 - Descrição da Arquitetura do MULTIPLUS

A necessidade de processar informações cada vez mais rápidamente, leva os projetistas de computadores a buscar novas soluções de arquitetura para os sistemas. O desenvolvimento da microeletrônica, que tem propiciado uma maior integração dos circuitos e, principalmente, o avanço da tecnologia CMOS, que possibilita a construção de transistores mais rápidos, tem fornecido componentes a baixo custo com velocidades superiores.

Observa-se que a demanda de processamento cresce a uma razão maior

do que a de aumento de velocidade dos computadores, assim como também é esperado um limite físico para o aumento da velocidade dos circuitos. Novas soluções foram investigadas de forma a tentar solucionar o problema. Primeiro houve a introdução de outros processadores para tarefas mais simples como as de E/S, depois vieram os computadores multiprocessados, onde vários processadores compartilham os mesmos recursos. Neste caso, o meio originalmente utilizado de acesso aos recursos era o próprio barramento paralelo já existente. Entretanto, o aumento do número de processadores acarreta um correspondente aumento do tráfego no barramento, não permitindo que este número seja muito grande. Assim, os projetistas procuraram outras formas de interconectar os processadores, as memórias e os periféricos. Soluções foram encontradas na especialização de barramentos e na inclusão de redes das mais variadas topologias.

Em continuidade ao desenvolvimento dos componentes, duas correntes em relação à filosofia de arquitetura do processador central se estabeleceram:

CISC (*Complex Instruction Set Computer*): esta corrente aposta em processadores com instruções mais complexas e por este motivo, microprogramados, o que leva a ciclos de instruções mais lentos, mas que ao mesmo tempo necessitam de poucas dessas instruções para executarem as tarefas.

RISC (*Reduced Instruction Set Computer*): a outra corrente acredita que os computadores possam ser baseados em processadores com instruções muito simples que podem ser executadas em apenas um ciclo de máquina e não necessitam ser microprogramados. Por outro lado, precisam de muito mais instruções do que os processadores CISC para realizarem as mesmas tarefas, aumentando muito a importância do projeto de memória *cache* para reduzir o tráfego no barramento e diminuir o tempo de espera dos processadores.

O MULTIPLUS é um computador multiprocessado de alto desempenho com arquitetura modular que está sendo desenvolvido no Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro. Capaz de suportar até 2048 nós de processamento, o MULTIPLUS é baseado em microprocessadores RISC

de 32 *bits* de arquitetura SPARC.

Em uma mesma placa (figura 1.2), denominada Nó de Processamento (NP) estão o processador, o co-processador de ponto flutuante, 64 K*bytes* de memória *cache* para dado e 64 K*bytes* de memória *cache* para instruções, até 32 M*bytes* de memória do tipo RAM e interface para barramentos externos caracterizados a seguir.

Até oito NP's podem ser interligados por dois barramentos de 64 *bits* de largura para dado com 36 *bits* de endereço formando um *cluster*. Os barramentos são especializados em transferência de instruções e transferência de dados, possuindo uma vazão máxima (de pico) de 320 M*bytes* por segundo [CYPR90b].

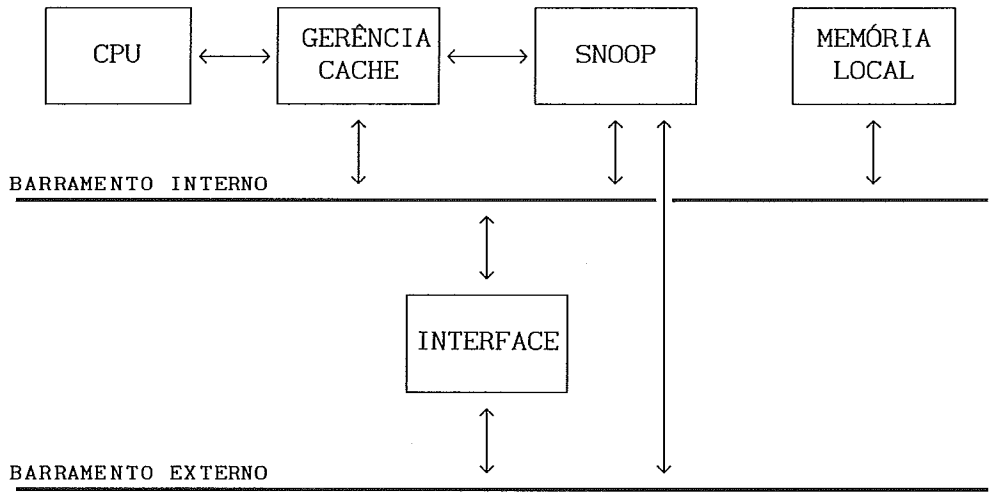


figura 1.2: diagrama em blocos do NP do **MULTIPLUS**

Os barramentos conectam os NP's às interfaces da rede de interconexão multiestágio do tipo N-cubo invertido (RI) [CRUZ90]. A RI pode suportar até 256 *clusters*.

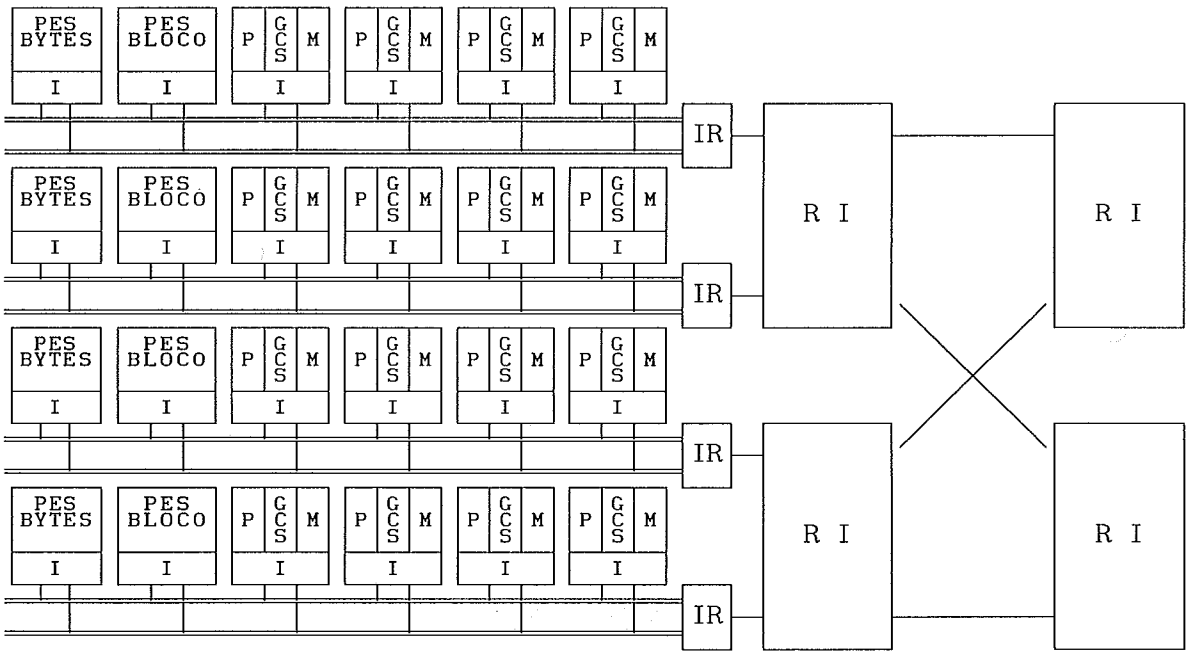
A memória local de cada NP pertence ao espaço de endereçamento global do sistema, podendo ser acessada por qualquer outro NP. Para o *software*, a memória do sistema é composta por um único grande bloco de até 32 G*bytes*.

A arquitetura de E/S do MULTIPLUS é distribuída. Para cada *cluster*

existem dois processadores de E/S (PES): um deles é orientado a blocos, controlando operações com discos e fitas; e o outro é orientado a *byte*, realizando operações com terminais e impressoras.

A figura 1.3 apresenta a arquitetura total do Sistema **MULTIPLUS** com 4 NP's por *cluster* e 4 *clusters*.

O esquema de manutenção de coerência de *cache* no MULTIPLUS é discutido na seção 4. Nas seções 2 e 3 serão apresentados alguns conceitos básicos de *cache* utilizados neste trabalho.



Legenda:
- P - Processador
- C - Cache
- G - Gerência
- S - Snoop
- M - Memória
- I - Interface
- RI - Rede de Interconexão
- IR - Interface de Rede
- ═ - Barramento Externo

figura 1.3: diagrama em blocos do Sistema **MULTIPLUS**.

CAPÍTULO II

REVISÃO DE MEMÓRIA "CACHE"

Dada a importância assumida pela utilização de memórias *cache* em sistemas multiprocessados, é lícito hoje afirmar que o desempenho de tais sistemas está intimamente ligado ao bom projeto do sistema de memória *cache*. Idealmente, o projeto deve considerar o tipo de aplicação à qual a máquina se destina para que possa ser identificada a sua *workload* (carga de trabalho) típica. Por outro lado, não é solução uma *cache* de tamanho muito grande que eleva demais o custo do projeto além de aumentar a sua complexidade ao ponto de torná-lo comercialmente inviável.

Variações mínimas da taxa de acerto da *cache* podem provocar uma diminuição muito grande no desempenho do sistema. Seja por exemplo, um sistema cujo tempo de acesso à memória principal é 10 vezes maior do que o de acesso à *cache*. Considerando-se apenas ciclos de leitura, o tempo médio de ciclo de *CPU* pode ser expresso pela fórmula:

$$t_{eff} = h * t_{cache} + (1 - h) * t_{main}$$

onde: t_{eff} = tempo eficaz médio de ciclo de *CPU*,
h = taxa de acerto na *cache*,
t_{cache} = tempo médio de acesso à *cache*,
t_{main} = tempo médio de acesso à memória principal.

Assim, para uma variação da taxa de acerto de 99% para 98% (variação relativa menor do que 1%), o tempo médio de ciclo aumenta em quase 10%.

Existem dois tipos de configurações básicas de memória *cache* para sistemas multiprocessados: a *cache* compartilhada (*shared cache*) e a *cache* privada (*private cache*) [DUBO82]. A *cache* compartilhada pode ser acessada por mais de um processador (fig. 2.1.a). É geralmente um *buffer* de alta

velocidade de uma determinada parte da memória. A memória *cache* privada é dedicada a um único processador e armazena blocos de diversas unidades de memória (fig. 2.1.b).

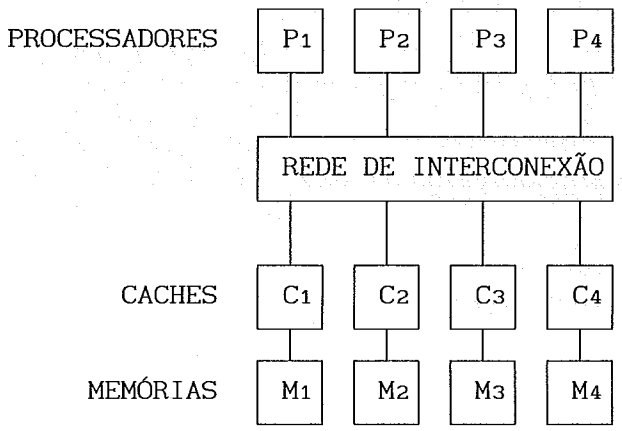


figura 2.1.a: sistema com *cache* compartilhada

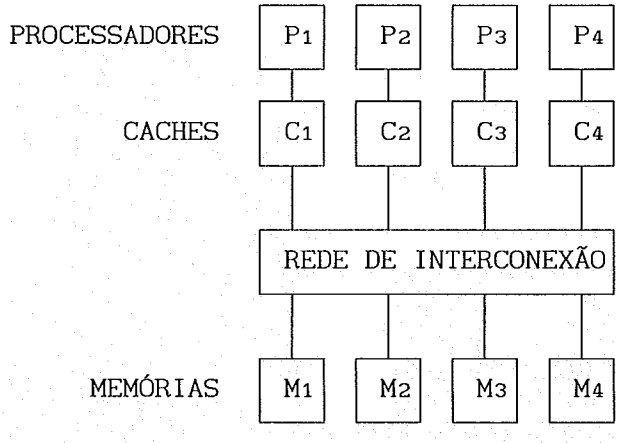


figura 2.1.b: sistema com *cache* privada

A desvantagem da *cache* compartilhada em relação à privada é que a primeira não reduz o tráfego nas vias de comunicação entre os processadores e os módulos de memória [KAPL73]. A *cache* privada é a opção mais utilizada em sistemas computacionais, sendo que o sistema Alliant FX/8 [TEST86] utiliza *cache* compartilhada.

II.1 - Alternativas de Arquiteturas "Cache"

Neste item serão apresentadas algumas características e parâmetros de arquiteturas de controladores e memórias *cache*.

Dependendo do esquema de gerência de memória[1] utilizado, são possíveis duas alternativas de arquitetura de *cache* em relação ao endereço: *cache* de endereço virtual e *cache* de endereço físico.

No *cache* virtual, o endereço que o controlador de *cache* recebe é o endereço virtual gerado pelo processador (figura 2.2). Neste caso, não existe o atraso correspondente ao cálculo de conversão do endereço virtual em real pela gerência de memória, possibilitando ao controlador de *cache* respostas mais rápidas. Por outro lado, a cada troca de contexto, quando as tabelas de gerência de memória são alteradas, há a necessidade de invalidar a *cache* toda, uma vez que o novo processo poderá gerar endereços virtuais que estejam presentes na *cache* mas não correspondam ao mesmo endereço físico.

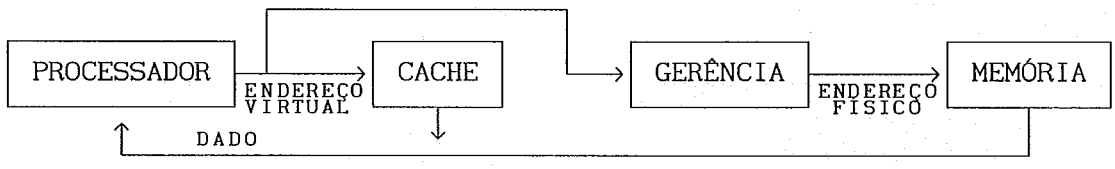


figura 2.2: diagrama em blocos de um sistema com *cache* de endereco virtual.

Alguns controladores de *cache* virtual (figura 2.3) armazenam também o endereço físico juntamente com os dados para evitar a necessidade de invalidar a *cache* a cada troca de processo. Para este tipo de controlador, é interessante que a gerência processe o endereço virtual em paralelo com a busca na *cache* e forneça o endereço físico em tempo menor do que o ciclo de *cache* para comparação com o endereço armazenado.

---

[1] entende-se como gerência de memória, neste caso, como sendo a unidade que, de alguma forma, remapeia os endereços gerados pelo processador, independentemente do método utilizado.

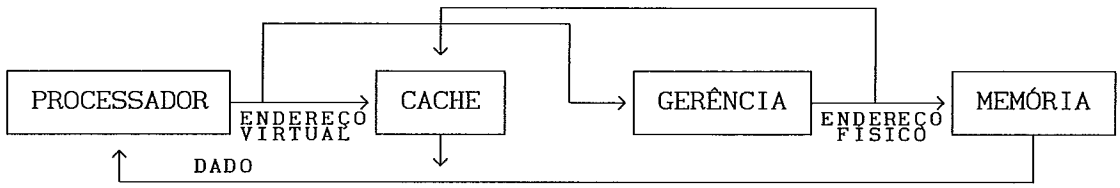


figura 2.3: diagrama em blocos de um sistema *cache* de endereço virtual e armazenamento de endereço físico

Cache Físico: a memória *cache* de endereço físico ou simplesmente *cache* físico, recebe o endereço já processado pela gerência de memória (figura 2.4).

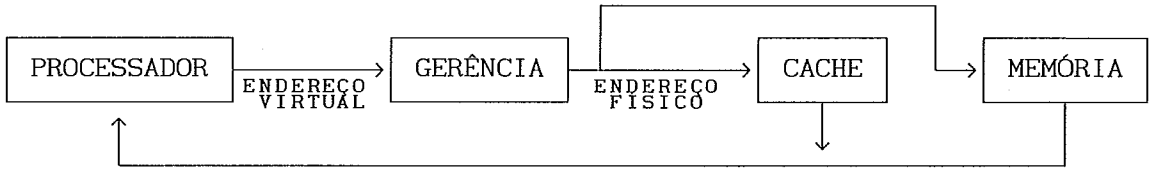


figura 2.4: diagrama em blocos de um sistema de *cache* de endereço físico

Para minimizar os efeitos da gerência entre a *cache* e o processador, geralmente aproveita-se o fato da gerência de memória não utilizar os *bits* de menor ordem da palavra de endereço e utiliza-se estes para endereçar a *cache*. Em paralelo, a gerência de memória converte o endereço virtual em real em um tempo compatível com o de acesso à *tag* de endereço da *cache*, ficando ambos os endereços disponíveis, quase que simultaneamente, para comparação pelo controlador de *cache* [HENN86]. O principal problema deste esquema está na limitação do número de entradas na *cache* pelo tamanho da página de memória virtual porque somente os *bits* não mapeados podem ser utilizados para endereçar a *cache* (figura 2.5).

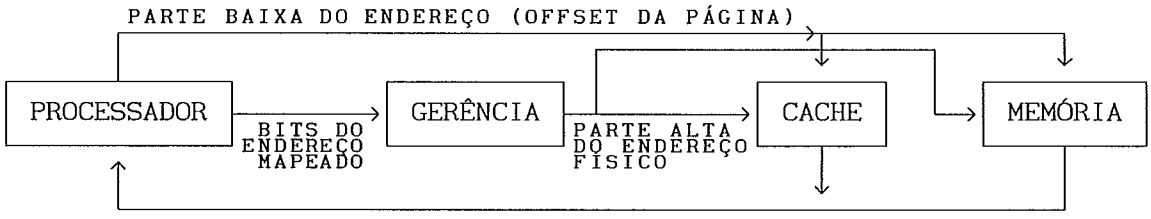


figura 2.5: diagrama em blocos de um sistema de *cache* de endereço físico e busca na *cache* em paralelo

A figura 2.6 fornece um exemplo de como a operação é realizada. Supondo um processador de 32 *bits* de endereço, páginas de 64K *bytes* e uma

cache com 2K entradas.

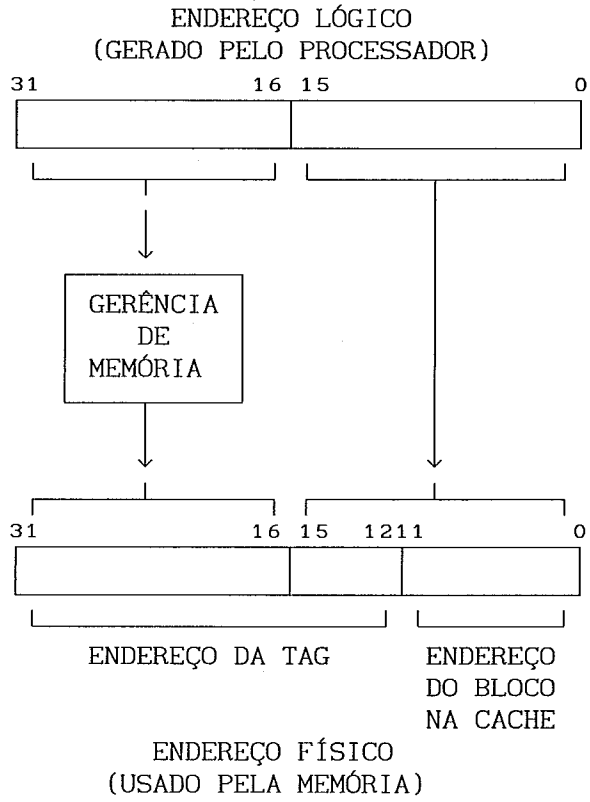


figura 2.6: formação de endereço físico e acesso à *cache* em paralelo

Uma memória *cache* pode ser vista como uma matriz de três dimensões que são as palavras, os blocos, e os conjuntos associativos (figura 2.7). Uma *cache* é formada por **s** conjunto associativos, cada um contendo **b** blocos de **p** palavras

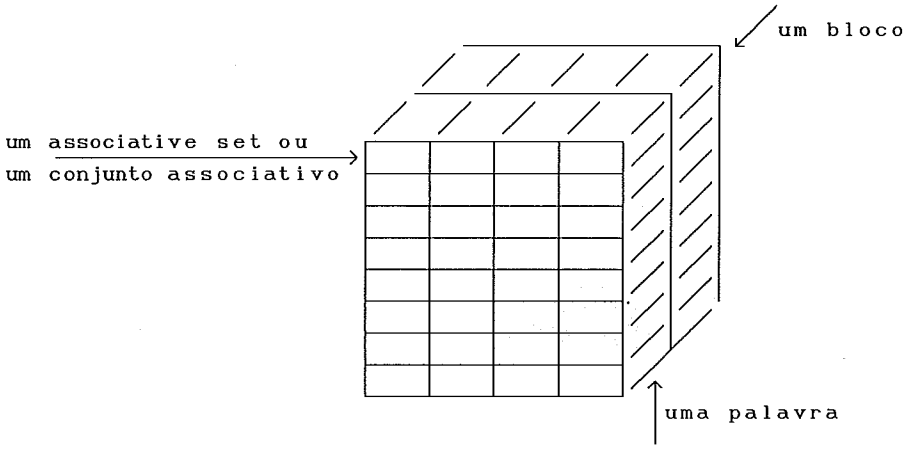


figura 2.7: representação de uma memória *cache* com:
- 64 palavras,
- bloco de 2 palavras,
- 8 conjuntos associativos de 4 blocos cada.

Um conjunto associativo (*associative set*) consiste de blocos que possuem em comum a mesma parte baixa do endereço do dado armazenado e podem mapear os mesmos blocos congruentes da memória principal. Quando é necessário ler um novo bloco da memória principal para a *cache*, um algoritmo de reposição deve selecionar qual bloco, dentro do conjunto, que deverá ser escolhido para ceder sua posição. Em geral, são algoritmos do tipo LRU, FIFO ou de base aleatória.

No exemplo da figura 2.7, os endereços dos conjuntos associativos seriam os três *bits* menos significativos da palavra de endereço, descontando-se os *bits* utilizados para selecionar o elemento acessado dentro do bloco[2], ou seja, os $\log_2 n_s$, onde n_s é o número de conjuntos associativos. O gráfico da figura 2.8 ilustra a relação entre o grau de associatividade e a taxa de falha na *cache*. Os blocos congruentes a um mesmo conjunto diferem entre si pelo rótulo associado (parte alta do endereço).

---

[2]Para um processador de 32 *bits* (4 *bytes*) e um bloco de *cache* de 256 *bits* (32 *bytes*), é necessário descontar 5 *bits* para seleção do elemento endereçado dentro do bloco.

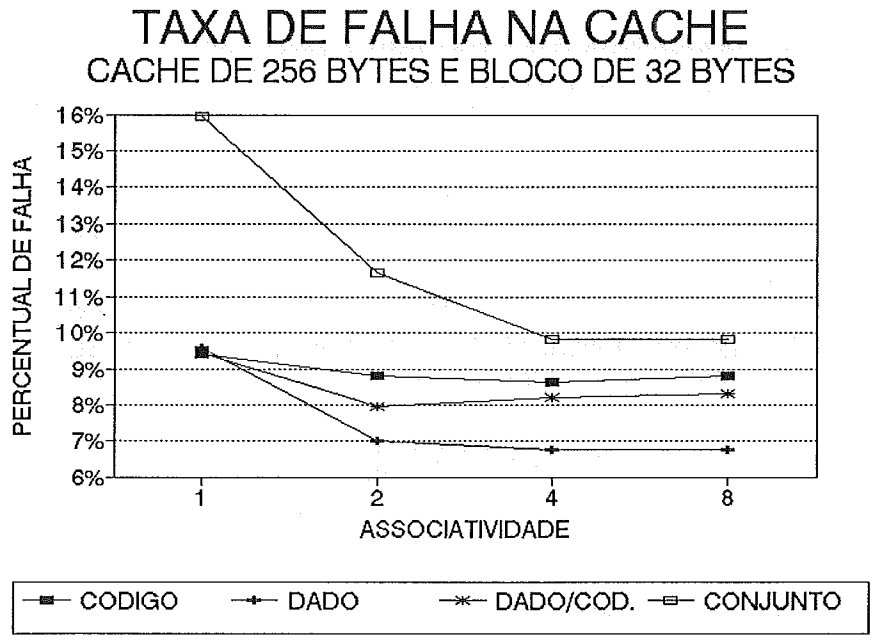


figura 2.8: grau de associatividade X taxa de falha na *cache* [3]

Definição de Bloco: é a unidade de informação nas operações de leitura/escrita entre o controlador de *cache* e a memória principal. Geralmente possui número de *bits* múltiplo da largura do barramento de dados, e normalmente é composto por "potência de 2" palavras do processador. Alguns autores preferem dar ao bloco o nome de linha, devido à forma de organização dos dados na memória *cache*.

O tamanho do bloco é um importante parâmetro para o desempenho da *cache*, visto que estudos sobre taxas de acerto de programas monoprocessados têm mostrado que, para uma *cache* de tamanho fixo, a taxa de falhas inicialmente diminui com o aumento do tamanho do bloco [EGGE89] porque os programas tendem a referenciar instruções e dados na vizinhança dos recentemente referenciados [KAPL73]. A medida que o tamanho do bloco cresce e se aproxima do tamanho da *cache*, a taxa de falhas começa crescer. Tal

---

[3] os gráficos 2.8 e 2.9 foram obtidos através de um pequeno simulador do tipo *trace driven* executando um programa que simula uma aplicação tipicamente científica.

fenômeno é explicado pelo fato de que a medida que a distância de um certo objeto pertinente ao bloco aumenta com relação ao objeto originariamente referenciado, sua chance de vir a ser referenciado diminui (*memory pollution* [SMIT87]). O gráfico 2.9 ilustra a relação entre o tamanho do bloco da *cache* e a sua taxa de falha.

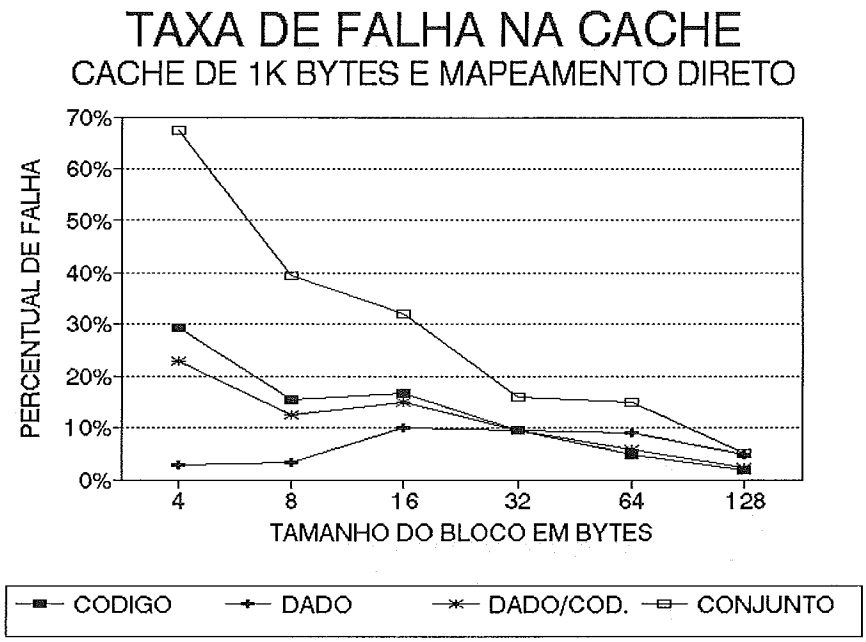


figura 2.9: tamanho do bloco X taxa de falha

Os comportamentos da demanda de dados de instruções de um programa não são idênticos, o que leva a uma necessidade de políticas diferentes para a gerência de *cache* [SMIT83].

Enquanto que a demanda de instruções de um programa é essencialmente sequêncial, com repetições destas sequências através de *loops*, dados são, em sua maioria, acessados por uma complexa lei de formação ou de forma puramente sequêncial quando se trata de vetores.

Sendo o código de um programa de apenas leitura, as invalidações na *cache* somente ocorrerão ao término de um programa, o que torna a coerência por *software* fácil de ser implementada. Em sistemas em que a *cache* seja acessada através de endereços virtuais, há necessidade de invalidação também quando houver troca de contexto, favorecendo o controle por *software*

da coerência dos dados armazenados em *cache*.

Neste mesmo esquema estão os dados apenas de leitura (*R/O*) ou utilizados por somente um processador. A partir desta separação, problema de manutenção de coerência se resume aos dados compartilhados por mais de um processador e que sejam de leitura e/ou escrita.

Weber [WEBE89] e Agarwal [AGAR88] dividem os objetos em diversos conjuntos dependendo do tipo de invalidação por eles provocados:

1 - código e dados de apenas leitura,
2 - objetos migratórios,
3 - objetos de sincronização,
4 - objetos muito lidos[4],
5 - objetos frequentemente lidos e escritos[5]

Uma outra divisão de dados que melhor se adapta às necessidades do sistema em discussão (figura 2.10) seria:

1 - código,
2 - dados de sincronização,
3 - dados locais,
4 - dados remotos no mesmo barramento,
5 - dados compartilhados através da rede de apenas leitura,
6 - dados compartilhados através da rede de leitura e escrita.

---

[4]objetos muito lidos são objetos que são lidos muitas vezes antes de serem escritos.

[5]objetos frequentemente lidos e escritos são objetos que são frequentemente lidos e frequentemente escritos simultaneamente.

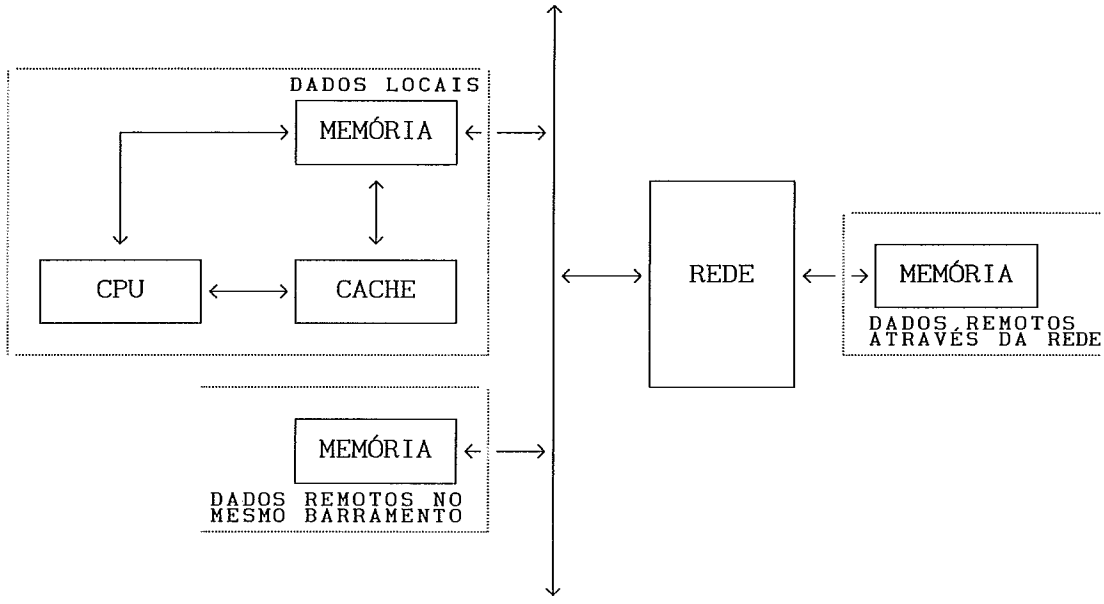


figura 2.10: distribuição dos dados em relação ao processador.

II.2 - Métodos de Atualização da Memória Principal

Esquemas de coerência de memória *cache* para sistemas multiprocessados interligados por um barramento comum a todas as unidades que podem acessar a memória principal já foram analisados por diversos autores ao longo do tempo [WILK65] [LEE69] [GOOD83]. A característica básica de um esquema de coerência consiste na monitoração do barramento (*snoop*) ou na troca de informações entre cada unidade que possua *cache* para detectar mudanças de conteúdo de dados que estejam mapeados em sua *cache* de forma a permitir que esta possa ser atualizada ou invalidada. Durante ciclos de leitura na memória principal, a monitoração é importante somente em alguns esquemas de *cache* que não atualizam a memória principal a cada ciclo de escrita da *CPU*. Neste caso, o valor contido em alguma *cache* será o atualizado, necessitando ser passado para a unidade que o requisitou.

Em sistemas *multicache* conectados por redes que não interliguem todas as unidades, não há um esquema de *snoop* implementável que não degrade consideravelmente o desempenho do sistema para monitorar a ação de unidades distantes (que não estão conectadas diretamente).

A seguir serão descritos alguns esquemas de atualização da memória principal comumente utilizados.

II.2.1 - "Write Trough"

O método *write through* destaca-se pela sua simplicidade e reduzido número de estados (dois - *Valid* e *Invalid*). Outra grande vantagem está no fato de que praticamente qualquer tipo de protocolo de barramento pode suportar esta política de controle de *cache*.

Neste esquema, toda escrita realizada pelo processador passa automaticamente para a memória principal. Em caso de *hit* (na *cache* privada local) na escrita, o dado da *cache* também é atualizado. Se em uma operação de leitura da *CPU* houver ausência na *cache* (*miss* ou *invalid*), o bloco inteiro que contém o dado faltoso é lido da memória principal e armazenado

na *cache* como válido. Subsequentes leituras deste bloco se processarão apenas entre processador e *cache*. Variações deste esquema, que assumem que localidades de referência também são geradas por escritas do processador, podem alocar um bloco na *cache* também em ciclos de escrita realizando um ciclo de leitura de bloco seguido da escrita da palavra. O bloco alocado nestas condições poderá desalocar algum outro bloco alocado por demanda de leitura.

A figura 2.11 ilustra o diagrama de estados de um bloco genérico na memória *cache* para esquemas do tipo *write-through*.

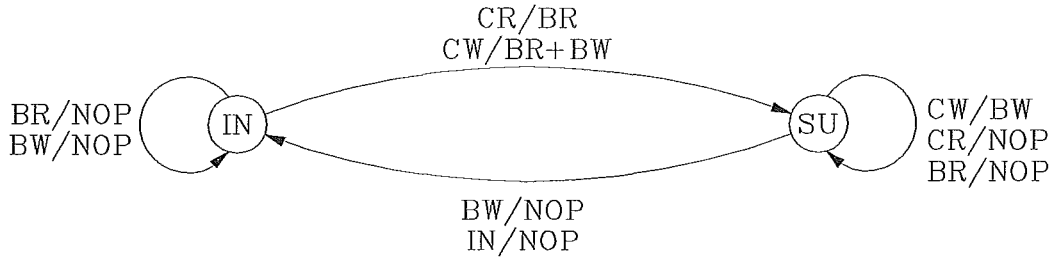


figura 2.11: esquema *write-through* - diagrama de estados de um bloco na *cache*

Ação/Reação
- BR - leitura de/no barramento
- BW - escrita de/no barramento
- CR - leitura executada pela CPU
- CW - escrita executada pela CPU
- IN - invalidação de entrada na *cache*
- NOP - nenhuma operação realizada ou necessária

Estado (dentro do círculo)
- IN - inválido (*INvalid*)
- SU - válido (*Shared Unmodified*)

Este método é o mais popular, sendo utilizado em sistemas como o Pégasus 32X do NCE-UFRJ. O método também é disponível em controladores de *cache* integrados fabricados comercialmente. Como exemplos, pode-se citar o Cypress CY7C604 e CY7C605 [CYPR89a], o Motorola 88200 [MOTO88], ambos para suas respectivas linhas de processadores RISC, e o Austek Microsystems A38152 [AUST87] para sistemas baseados em microprocessadores 80386 [INTE87] da Intel.

O esquema *write-through* de atualização da memória principal acaba gerando tráfego desnecessário no barramento quando uma determinada posição de memória é seguidamente acessada em escritas (intercaladas ou não por leituras) sem que nenhum outro dispositivo necessite acessá-la. Neste caso, várias escritas de um mesmo dispositivo são feitas sobre valores que não foram utilizados (lidos).

II.2.2 - "Write Back"

Um esquema alternativo de atualização, chamado de *copy back* ou *write back*, não escreve necessariamente na memória principal a cada escrita do processador. A *cache* possui, não mais apenas dois estados, como no esquema de *write-through*, e sim quatro que são:

Invalid (inválido):	posição ausente ou não atualizada nesta *cache* (*cache miss*),
Private (privado):	conteúdo válido e presente apenas nesta *cache*. A memória principal está atualizada,
Modified (modificado):	o conteúdo da *cache* está atualizado, presente somente nesta *cache* e a memória principal não está atualizada,
Shared (compartilhado):	o conteúdo da *cache* está válido assim como a memória principal. Pode existir uma ou mais cópias em outras *caches*.

A figura 2.12 ilustra o diagrama de estados de um bloco genérico na memória *cache* para esquemas do tipo *write-back*.

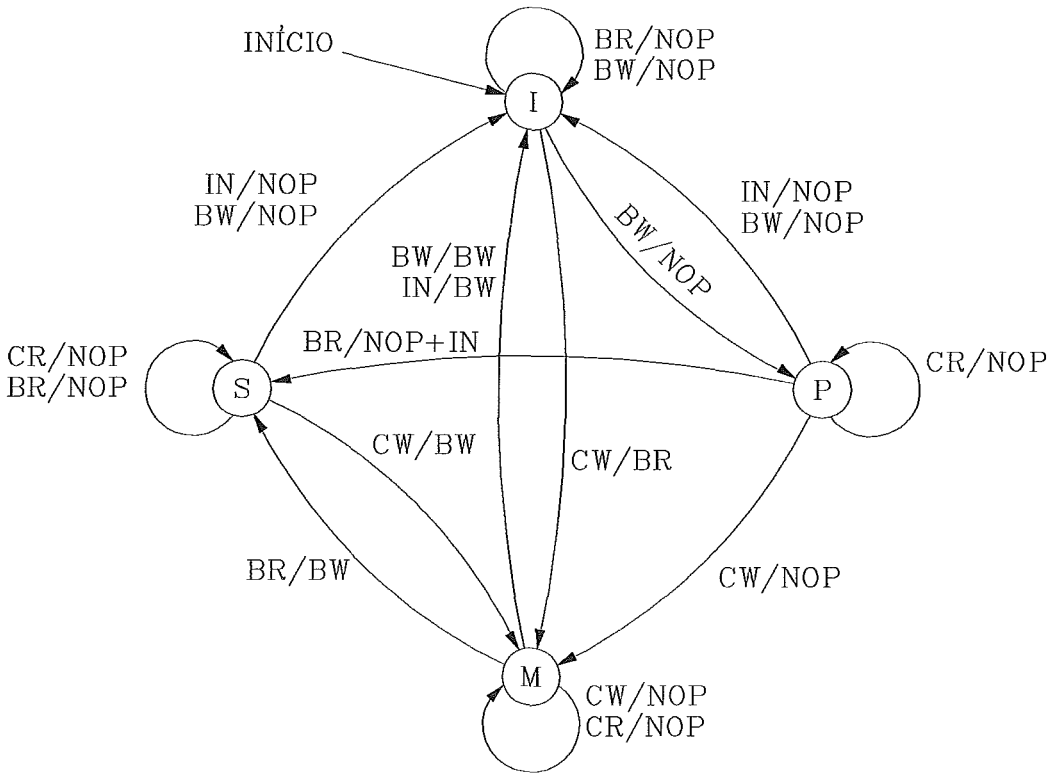


figura 2.12: esquema *write-back* - diagrama de estados de um bloco na *cache*.

Ação/Reação
BR - leitura de/no barramento
BW - escrita de/no barramento
CR - leitura executada pela CPU
CW - escrita executada pela CPU
IN - invalidação de entrada na *cache* (ação)
IN - invalidação de entrada nas outras *caches* (reação)
NOP - nenhuma operação realizada ou necessária

Quando um determinado processador **i** requisita um dado que não está presente na sua memória *cache*, este será lido da memória principal. Se houver outra cópia deste mesmo dado em outra *cache* **j** (**j** ≠ **i**), o dado será armazenado como *shared*. Caso não exista nenhuma outra cópia, o dado será armazenado como *private*.

O meio utilizado para interligar as diversas unidades, que compartilham a memória, deve prever, no seu protocolo, a possibilidade de

interrupção de ciclo de barramento em leituras e em escritas.

Em qualquer dos casos anteriores, um acesso do processador **i** a este mesmo dado já armazenado na *cache* **i** ocorrerá sem que o barramento externo seja utilizado, ou seja, um acesso a um dado presente na *cache* não gera acesso ao barramento, com exceção de uma escrita a um dado armazenado em *cache* no estado compartilhado (*shared*). Este acesso é necessário para que os outros controladores de *cache* saibam que o dado mudou de estado (de *shared* para *modified*).

Operações de escrita em bloco marcado como *private* provoca mudança no seu estado para *modified*, enquanto que em blocos no estado *shared*, causam um acesso ao barramento para que os outros controladores de *cache* possam invalidar o bloco, como já foi dito anteriormente.

Quando um dado presente na *cache* **i** é acessado na memória principal por um outro processador **j**, o seu estado muda para *invalid*, se o acesso for de escrita, ou para *shared*, se for de leitura. Caso o estado inicial seja *modified*, o ciclo corrente é interrompido, o controlador **i** atualiza a memória principal e o processador **j** reinicia o ciclo antes interrompido.

O esquema de *write back* é utilizado no sistema SEQUENT SYMMETRY [MANU87] e também pela CYPRESS que possui um controlador de *cache* (CY7C604 e CY7C605) [CYPR89a] para sua linha SPARC.

II.2.3 - "Write Once"

O esquema de *write-once* é uma modificação, proposta por Goodman [GOOD83], do esquema de *write-back* convencional. O reduzido tráfego no barramento é o grande destaque deste método de *cache* de quatro estados.

A figura 2.13 ilustra o diagrama de estados de um controlador de *cache* para esquemas do tipo *write-once*.

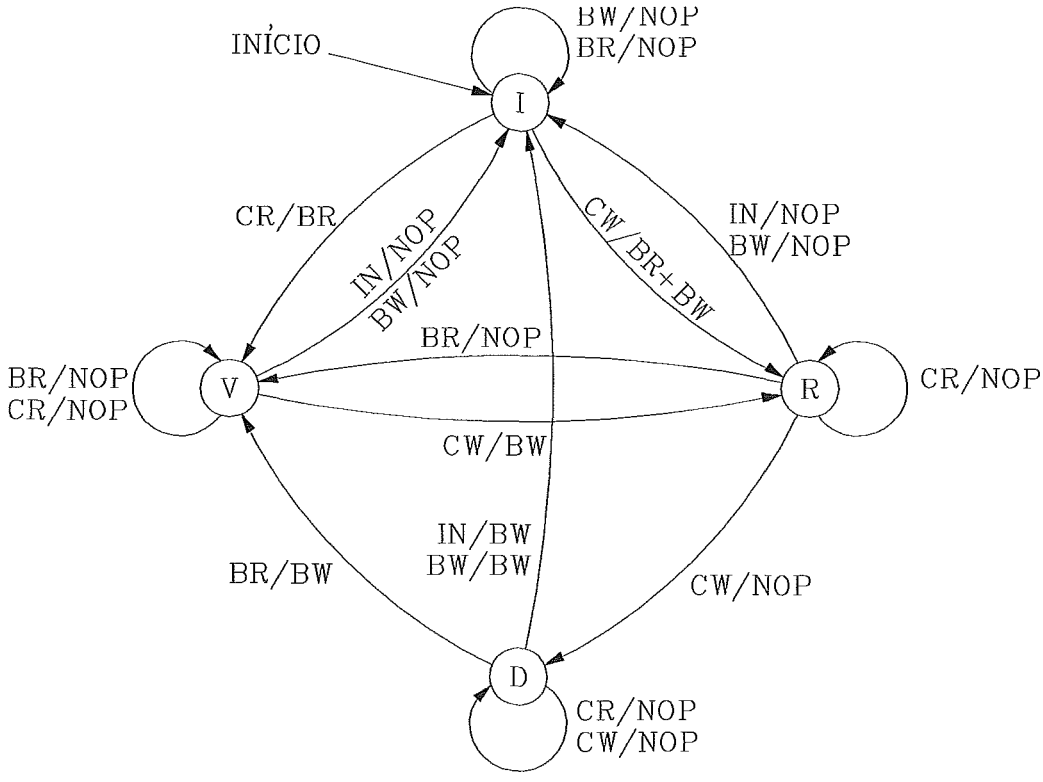


figura 2.13: esquema *write-once* - diagrama de estados de um bloco na *cache*.

Ação/Reação
BR - leitura de/no barramento
BW - escrita de/no barramento
CR - leitura executada pela CPU
CW - escrita executada pela CPU
IN - invalidação de entrada na *cache*
NOP - nenhuma operação realizada ou necessária

Invalid (inválido): posição ausente ou não atualizada nesta *cache* (gera um *cache miss*),

Reserved (reservado): conteúdo válido e presente apenas nesta *cache*. A memória principal está atualizada,

Dirty (sujo): o conteúdo da *cache* é a única cópia atualizada em todo o sistema,

Valid (válido): o conteúdo da *cache* está válida e pode haver mais alguma cópia em outra unidade de *cache*. A memória principal está atualizada.

Inicialmente o método de *write-through* é utilizado em uma escrita de um bloco específico, todas as outras *caches* invalidam este bloco de forma tal que a única cópia do bloco garantidamente pertence a uma única *cache* que é marcada como *reserved*. Subsequentes escritas neste bloco não necessitam atualização imediata da memória principal; o bloco é marcado como *dirty* e o método de *write-back* é então utilizado. O *snoop* verifica, a cada operação no barramento, se o endereço referenciado está na *cache* local. Se houver coincidência em uma operação de escrita, o bloco é marcado como *invalid*. Se coincidir em uma operação de leitura, nada será feito, a menos que o bloco já tenha sido modificado antes (*dirty* ou *reserved*). Se o estado for apenas *reserved*, este será trocado para *valid*. Se for *dirty*, a memória principal será inibida pela interface de *cache* que atualizará o dado nela. No mesmo ciclo de barramento, ou imediatamente após, o dado poderá ser lido da memória principal. O estado do dado na *cache* local será trocado para *valid*.

Este esquema é utilizado pela MOTOROLA em sua linha de RISC 88000 [MOTO88].

As figuras 2.14.a e 2.14.b fornecem os fluxos de ações para duas situações de escrita. Em **negrito** estão os estados dos blocos ou transições destes. Neste caso, é suposto que são necessários quatro acessos para carregar um bloco na *cache*.

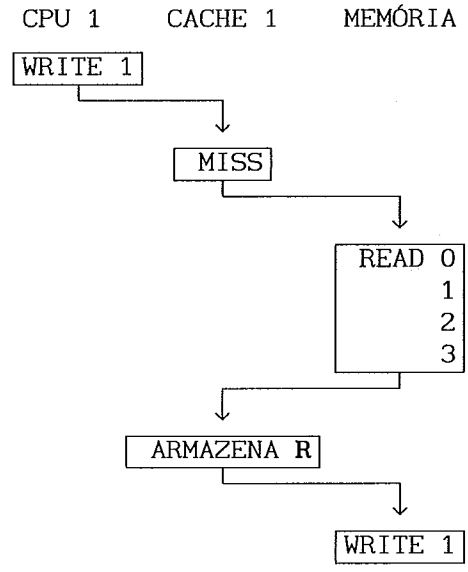


figura 2.14.a: esquema *write-once - write-miss* sem *hit* nas outras *caches*

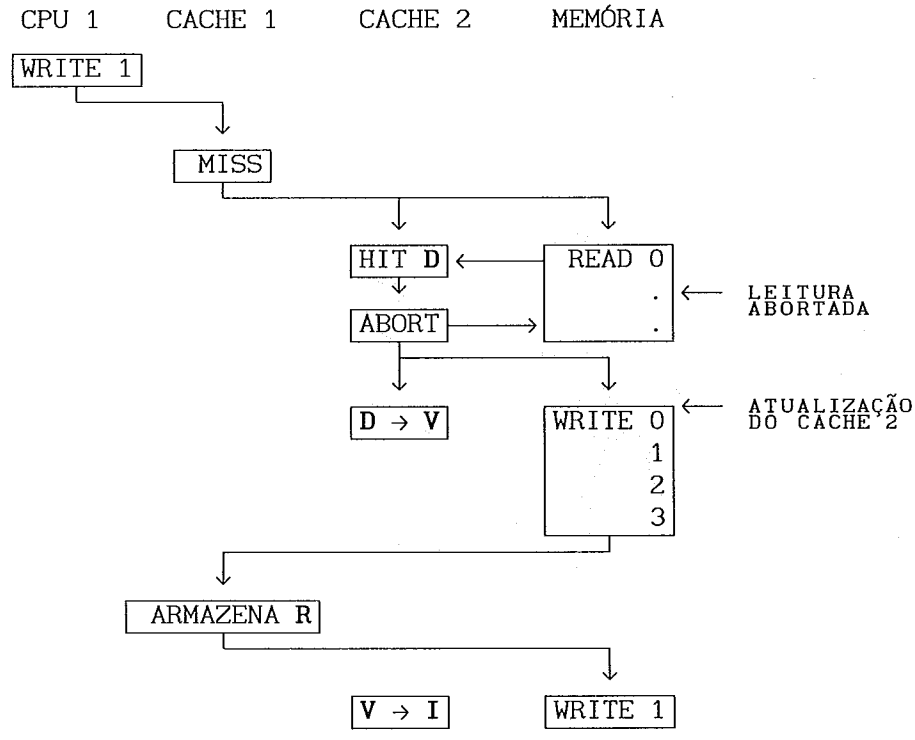


figura 2.14.b: esquema *write-once write-miss* com *hit* na *cache* 2

II.3 - Manutenção da Consistência

O problema de consistência em memórias *cache* aparece em multiprocessadores com *caches* privados [ARCH84]. Nestes sistemas, cada processador tem associado a ele uma *cache* ou memória rápida para armazenar o conteúdo das localizações de memória acessadas mais recentemente (ver figura 2.1). Se mais de uma *cache* pode ter uma cópia da mesma localização de memória e não existe nenhum meio de evitar que os processadores modifiquem simultaneamente suas respectivas cópias, poderá haver inconsistência entre as diversas cópias, isto é, a **base de dados** pode se tornar incoerente. Nestas condições há necessidade de um mecanismo que:

1. mantenha a memória principal sempre com a informação atualizada e as *caches* operem apenas para leitura (esquema tipo *write-through*); ou
2. possibilite múltiplas cópias mas tenha como restrição que todas as cópias sejam dados privados, código ou dados não modificáveis (as *caches* precisam ser invalidadas somente para troca de contexto);
3. permita dados compartilhados que também possam ser escritos (requerendo invalidações da *cache* e atualizações da memória principal).

Independentemente do método utilizado, um sistema multiprocessado tem uma *cache* coerente se um acesso de leitura a qualquer bloco sempre retorna o valor escrito mais atualizado.

A maior dificuldade de manter a coerência em sistemas "fracamente acoplados" está no custo da troca de informações entre controladores de *cache* quando ocorre mudança nos estados dos dados armazenados nelas.

Existe uma alternativa proposta por Stenström [STEN89] para, mesmo através da rede de interconexão, manter a coerência das *caches*. Este método permite que existam quantas cópias o sistema necessitar de cada dado. O método propõe a inclusão de um campo de *flags* em cada *cache* local para cada *cache* que possa conter uma cópia dos dados. No caso do MULTIPLUS, este campo seria de 2048 *bits*, o que o torna inviável.

Modificação mais viável deste método, para a arquitetura do

MULTIPLUS, transfere as *flags* das *caches* locais para a memória principal com a inclusão de um *bit* de validade. Neste caso, quando uma *CPU* acessar um dado, este passa a pertencer ao seu barramento. O número do barramento é escrito no lugar do dado na memória principal e o *bit* de validade deste bloco é apagado. Outras *CPU*'s do mesmo barramento podem armazenar este dado em sua *cache* local e a coerência é mantida dentro do *cluster* através de esquema baseado em *snoop*. Quando outra *CPU* de outro barramento tentar acessar este dado na memória principal (este dado, por definição, estará inválido em sua *cache* local), encontrará o *bit* de validade apagado e a própria chave re-roteará o acesso para o barramento **DONO DO DADO**. Este último, por sua vez, invalidará o dado, caso seja uma escrita, ou apenas o transmitirá, em caso de leitura (ação que poderá variar dependendo da forma de implementação do protocolo). Neste esquema, somente uma única cópia de um dado será permitida por barramento.

CAPÍTULO III

VARIAÇÕES DA ARQUITETURA DO MULTIPLUS

Nesta seção serão discutidas algumas variações possíveis para a arquitetura do sistema MULTIPLUS, principalmente a questão dos barramentos e sistemas *cache*.

III.1 - Barramento

Um dos grandes "gargalos" de sistemas multiprocessados está no barramento. Diversas soluções já foram apresentadas para minimizar o seu efeito como por exemplo: a inclusão de memórias *cache* [LIPT68] [TANG76] [GOOD83], a utilização de redes de interconexão [GOTT83], a utilização de memórias privadas [SEIT85] ou novos protocolos de barramento mais rápidos [IEEE87].

No projeto do Sistema MULTIPLUS procurou-se diminuir o "gargalo" através da utilização de um barramento duplo, uma vez que simulações mostraram que, mesmo utilizando-se diversos recursos como os já citados anteriormente, o número de processadores ficaria muito limitado.

O barramento único possui a grande vantagem da simplicidade de projeto e do baixo custo de material aliada à pouca área necessária para sua implementação.

O número máximo de Unidades de Processamento que podem compartilhar um mesmo barramento é dado por:

$$NPE = \frac{TCACHE * PHIT}{(1 - PHIT)TBUS} + 1$$

onde: PHIT - taxa de acerto na *cache*
TBUS - tempo total de acesso ao barramento
TCACHE - tempo total de acesso à *cache*
NPE - número de processadores

que para uma taxa de acerto na *cache* de 97% levaria a um máximo de 4

processadores[1].

A existência de memória local de grande capacidade de armazenamento (16M bytes) tende a diminuir o tráfego no barramento aumentando o limite teórico para 5 ou 6 processadores.

Teoricamente, a inclusão de um segundo barramento deveria dobrar o número máximo de processadores, o que de fato não ocorre devido ao aparecimento de outros problemas e *overheads*, assim como dificuldades de gerenciamento dos barramentos, de manutenção de coerência, etc.

Esta arquitetura de duplo barramento aumenta a complexidade do projeto do circuito de controle de *cache*, principalmente com relação ao *snoop*, que deverá monitorar ambos os barramentos.

A primeira idéia foi a de se utilizar os dois barramentos sem especialização, ou seja, cada um deles operando com instruções e dados e com escrita e leitura. Ou seja, quando um processador necessitasse acessar um objeto através do barramento, ele poderia utilizar qualquer um dos dois, preferencialmente aquele que se encontrasse desocupado, ou no caso de ambos estarem ocupados, entrar em uma fila de espera (ou em ambas, de acordo com o protocolo utilizado).

A não especialização dos barramentos externos traz problemas de compatibilidade com o controlador de *cache* utilizado (CY7C605 [CYPR90a]). Este controlador possui capacidade de monitorar (*snoop*) apenas um barramento por vez, e no caso de se utilizar o esquema de *write back*, este necessita que tanto os acessos de leitura de quanto os de escrita sejam monitorados, seria necessário "serializar" os acessos para o *snoop* do controlador pudesse monitorar os dois barramentos. Tal fato levaria o sistema com barramento duplo a um desempenho semelhante ao de um barramento simples.

---

[1] este máximo é assintótico, ou seja, o ótimo estaria perto da metade calculada.

Alguns resultados do trabalho de [MAMJ88] e [TAMI81] sugerem que uma aproximação muito boa (ver apêndice A) para tipos de acessos em sistemas baseados em processadores de arquitetura RISC, sugere que 80% dos acessos são de busca de instrução e que destes, somente 4% necessitam do barramento. Nos acessos de leitura de dados, (20% do total) 6% utilizam o barramento, enquanto que as escritas (7% do total de acessos) necessitam quase que sempre do barramento (chegando a necessitar sempre no caso de utilização de política de *write-through*).

A partir destes resultados pode-se chegar à conclusão que uma especialização de barramentos utilizando-se um deles para escrita e o outro para leitura iria dividir bem a sua demanda. O controle de *snoop* para tal poderia ser resolvido utilizando-se uma das facilidades oferecidas pelo controlador de *cache* que é a possibilidade de escolha entre dois esquemas de manutenção de coerência em *cache*: *copy-back* ou *write-through*. Este segundo esquema necessita de *snoop* somente nos ciclos de escrita. O problema é que a especialização dos barramentos nestes termos limita a utilização de esquemas de manutenção de coerência impossibilitando o uso de *copy-back*[2] que poderia diminuir a taxa de ocupação no barramento. A limitação do uso da política de *copy-back* reside no fato desta política necessitar de monitoração tanto em operações de escrita como em leitura, o que obrigaria a existência de dois monitores.

Uma opção mais natural seria a utilização de um barramento somente para leitura de instruções. Neste barramento não haveria necessidade de *snoop* uma vez que, por definição do Sistema, as intruções não são modificáveis, deixando todas as transações de dado, que requerem *snoop*, para o segundo barramento. O controlador de *snoop* ficaria, assim, dedicado a "bisbilhotar" apenas o barramento de dados.

De nada adiantaria em termos de desempenho utilizar no sistema

---

[2]esta política de atualização da memória principal requer monitoração do barramento em operações de leitura e escrita, enquanto que a política de *write-through*, apenas em escritas.

MULTIPLUS um protocolo de barramento que levasse a grandes *overheads* ou a uma baixa taxa de transferência. Além disso, o barramento necessita ser também um padrão muito utilizado para facilitar o interfaceamento com unidades que sejam encontradas comercialmente. É necessário que o barramento escolhido tenha um fator de confiabilidade elétrica muito grande e que tenha sido desenvolvido para ser instalado em *backplanes*, além de ser compatível com alguns dos esquemas de *cache* mais populares. Por tudo isto, foi escolhida a arquitetura com barramentos duplos, pois é a que melhor se adapta as necessidades de *througput* e de manutenção de coerência dos dados armazenados em *cache* como poderá ser visto no item 3.2. A opção pelo protocolo Mbus [SPAR89] é a solução natural, uma vez que este se interfaceia diretamente com o controlador de *cache* utilizado.

III.2 - Memória "Cache"

O controlador de *cache* comercial utilizado integra em um único encapsulamento de 244 pinos uma Unidade de Gerência de Memória (MMU - *Memory Management Unit*) e um Controlador de Cache (CC - *Cache Controller*) além de possuir internamente todas as *tags* e *flags* necessárias para manutenção de coerência e reposição de linhas, sendo totalmente compatível com a arquitetura SPARC.

A gerência possui uma TLB (*Translation Lookaside Buffer*) de 64 entradas com procura automática por *hardware* na memória principal em caso de falha; suporta 4096 contextos e possui proteção a nível de página que podem ser de 4K, 256K, 16M ou 4G *bytes*.

O controlador de *cache* suporta políticas de manutenção de coerência do tipo *write-through* sem alocação por escritas (*without write allocate*) ou *copy-back* com alocação por escritas (*with write allocate*), com possibilidade de troca direta de dados entre *caches* (*direct data intervention*), atualizando ou não a memória principal (*with or without reflectivity*). Cada controlador pode armazenar (utilizando duas pastilhas de memória do tipo CY7C157) até 64K bytes distribuidos por 2048 blocos de 32 bytes cada, mapeados diretamente.

Podem ser agrupadas até 4 (8 como poderá ser visto a seguir) destas unidades que, somando suas características, configuram uma memória *cache* de 256K (512K) bytes com mapeamento direto.

Cada unidade de memória *cache* engloba um controlador de *cache* e duas unidades de armazenamento. Um *cache* único diminui o custo e a área necessária para sua implementação. Outra grande vantagem está na simplicidade de interfaceamento entre processador e *cache*.

Por outro lado, um *cache* único para instrução e dados pode gerar conflito de endereços entre instruções e dados, principalmente no caso do controlador da CYPRESS CY7C605 que só possui modo de mapeamento direto, o que pode ocasionar um número grande de invalidações.

Mais particularmente em relação a arquitetura do MULTIPLUS, embora a conexão entre processador e *cache* seja mais simples porque só existe um único caminho, a interface entre *cache* e a memória local é mais complexa devido a existência de dois barramentos.

A existência de um único controlador de *cache* para instrução e dado também obriga a uma serialização das ações dos dois barramentos para fins de *snoop*, assim como no caso de barramentos não especializados (ver seção 3.1).

Também como no caso dos barramentos, a duplicação e especialização dos *caches* facilitam o interfaceamento com os barramentos Mbus.

A separação dos *caches* diminui também o número de invalidações devido a conflitos entre endereços de código e dado entre tarefas, possivelmente amenizando o efeito do *cache* ser de mapeamento direto.

Embora alguns autores [CYPR89b] afirmem que a *cache* organizada com instruções e dados separados tenha desempenho inferior (com relação à taxa de acerto) à *cache* única, é observável que esta melhoria no segundo caso é devido principalmente à taxa de acerto das instruções, ou seja, um sistema

com apenas *cache* de instruções teria um desempenho ainda comparável a um com *cache* de instruções e dados.

É intuitivo que a taxa de acerto da *cache* de instruções seja maior do que a de dados porque enquanto que o padrão de acesso de dados, com exceção de matrizes e vetores, é pseudo-aleatório, o que leva a *cache* a acertar quase que somente pela re-utilização das variáveis; o padrão de endereços gerado pela busca de instruções é basicamente sequencial, facilitando o acerto através da busca antecipada. Além disso, os programas são constituídos por diversos *loops*, o que leva a acertos também por re-utilização do código.

O conjunto de circuitos integrados (*chipset*) utilizado para formar a unidade de processamento do MULTIPLUS é composto por um processador RISC de 32 *bits*, comumente chamado de Unidade Inteira (IU - *Integer Unit*), um co-processador de ponto flutuante (FPU - *Floating Point Unit*), e um controlador de *cache* com gerência de memória (CC/MMU - *Cache Controller/ Memory Management Unit*).

A arquitetura da unidade de processamento da Cypress (IU + FPP + CC/MMU) não prevê a separação de *cache* em dado e instrução. Um artifício que pode ser utilizado consiste em anular ciclos de busca de instrução para o *cache* de dados e vice-versa através do sinal de **SNULL** do controlador de *cache* e do sinal **DXFER** da IU. Um cuidado especial deve ser tomado para não anular o ciclo de dado para o controlador de *cache* de instrução quando for ser realizada a sua programação e anulá-lo para o controlador de *cache* de dados para que sua programação não seja alterada pela programação do primeiro.

A figura 3.1 apresenta o diagrama em blocos do nó de processamento (NP) do Múltiplus e as tabelas 3.2.a e 3.2.b o controle dos *buffers*.

Esta arquitetura permite expandir a memória *cache* dos 64K *bytes* iniciais para até 512K *bytes* (256K de instruções e 256K de dados).

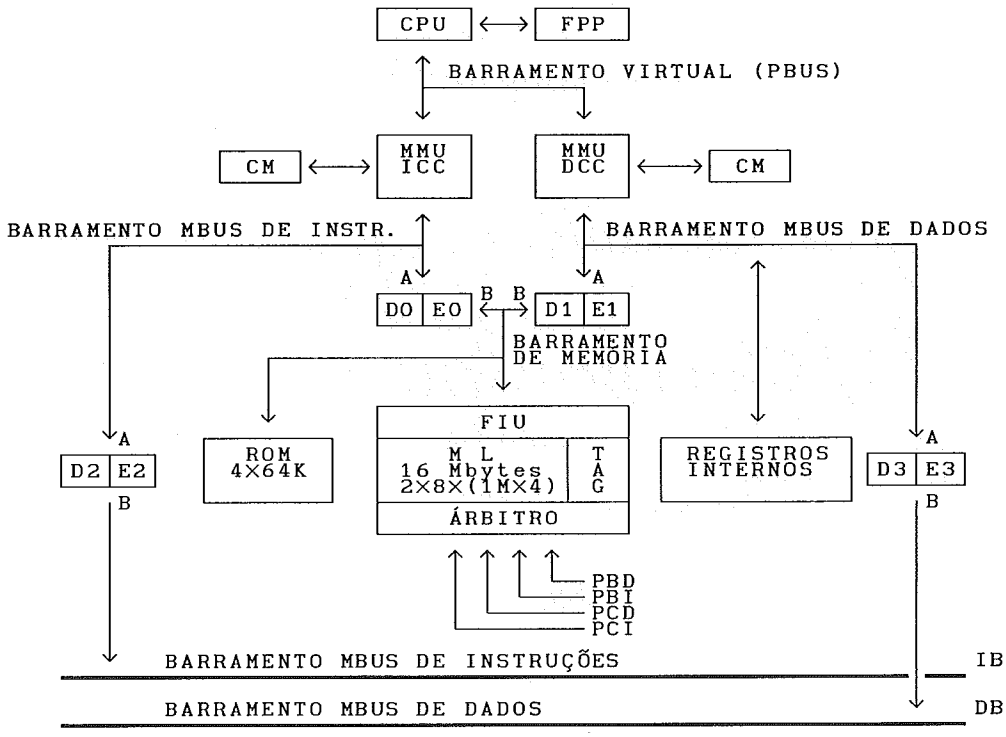


figura 3.1: diagrama em blocos de um NP genérico do MULTIPLUS

legenda:

CPU	- unidade de processamento central
FPP	- processador de ponto flutuante
MMU	- unidade de gerência de memória
[I/D]CC	- controlador de *cache* de instrução/dado
CM	- memória *cache*
FIU	- unidade de busca e soma
ML	- memória local
Bn	- *buffer* número n
IB	- barramento de instruções
DB	- barramento de dados
P[B/C][D/I]	- pedido de barramento/*cache* de dados/instrução
TAG	- marcador de memória compartilhada
REGISTROS	- interface serial
INTERNOS	- registro de LED's
	- registro de chaves
	- registros endereços do nó de processamento
	- chaveador de EPROM
	- registro de *time-out*

A C E S S O D E I N S T R U Ç Ã O

BUFFER	D0	D1	D2	D3	E0	E1	E2	E3
AÇÃO	D I R E Ç Ã O D O B U F F E R							
CRI	BA	O	O	X	AB	O	O	X
CWI	N Ã O E X I S T E E S T E C A S O							
BRI	BA	O	AB	X	AB	O	BA	X
BWI	AB	O	BA	X	AB	O	BA	X
CII	N Ã O E X I S T E E S T E C A S O							
BII	N Ã O E X I S T E E S T E C A S O							

figura 3.2.a: estados dos *buffer*'s no caso de acessos de instrução.

legenda: D_n - *buffer* de dado número n
E_n - *buffer* de endereço número n
AB - direção do fluxo do *buffer* n de A para B
BA - direção do fluxo do *buffer* n de B para A
CRI - leitura de instrução na *cache* local
CWI - escrita de instrução na *cache* local
BRI - leitura de instrução de uma unidade remota
BWI - escrita de instrução de uma unidade remota (provavelmente uma transferência de E/S)
CII - invalidação de instrução na *cache* local
BII - invalidação de instrução de uma unidade remota

A C E S S O D E D A D O

BUFFER	D0	D1	D2	D3	E0	E1	E2	E3
AÇÃO	D I R E Ç Ã O D O B U F F E R							
CRD	O	BA*1	X	$\frac{S}{\overline{S}}$ BA*1 / O	O	AB	X	$\frac{S}{\overline{S}}$ AB*1 / O
CWD	O	AB	X	$\frac{S}{\overline{S}}$ AB*2 / O	O	AB	X	$\frac{S}{\overline{S}}$ AB*2 / O
BRD	O	BA	X	AB	O	AB	X	BA
BWD	O	AB	X	BA	O	AB	X	BA
CID	X	O	X	X	X	O	X	AB
BID	X	O	X	X	X	O	X	BA

figura 3.2.b: estados dos *buffer*'s no caso de acessos de dado.

legenda: D_n - *buffer* de dado número n
E_n - *buffer* de endereço número n
CRD - leitura de dado na *cache* local

CWD - escrita de dado na *cache* local.
BRD - leitura de dado de uma unidade remota
BWD - escrita de dado de uma unidade remota
CID - invalidação de dado na *cache* local
BID - invalidação de dado de uma unidade remota

*1 O *buffer* 1 será desabilitado caso o dado esteja no estado *shared* e algum outro controlador de *cache* ative o sinal MHI. Neste caso o *buffer* 3 também será habilitado.

*2 O *buffer* 3 somente será habilitado se o dado estiver no estado *shared*.

CAPÍTULO IV

COERÊNCIA DE "CACHE" NO MULTIPLUS

Com o aparecimento de sistemas multiprocessados e *caches* privados distribuídos, houve um aumento significativo na dificuldade de se manter a consistência dos objetos armazenados nas *caches*. A política de manutenção de coerência cresceu, assim, em importância em relação a outros parâmetros como o algoritmo de reposição, associatividade, etc.

O desempenho do sistema multiprocessado está intimamente relacionado ao esquema de coerência implementado porque todos os controladores de *cache* que podem compartilhar objetos devem poder se comunicar para que mudanças no estado de objetos compartilhados sejam sinalizadas. Se esta comunicação for trazer um *overhead* muito grande ao sistema, talvez fosse melhor que este não tivesse *cache*.

IV.1 - Manutenção da Coerência por "Hardware"

O controlador de *cache* utilizado possui um circuito de *snoop* que monitora continuamente as ações no barramento de dados interno (ver figura 1.2), Mbus (de endereços físicos), mantendo o estado dos objetos contidos na *cache* sempre atualizados. Para que os controladores de *cache* possam monitorar os seus respectivos barramentos Mbus, os *buffers* da interface devem sempre estar habilitados levando o endereço para dentro do NP.

A arquitetura de memória do MULTIPLUS possibilita que acessos do processador à sua memória local sejam realizados sem a necessidade de utilização do barramento externo. Como esta memória local, logicamente pertence ao espaço de endereçamento global, outros processadores também podem acessá-la, levando dados desta memória para suas *caches* locais.

A base de dados pode se tornar inconsistente quando um processador escrever na sua memória local um dado que esteja presente em um bloco armazenado em uma outra *cache* de um outro processador. Este tipo de inconsistência tanto pode surgir quando se utiliza políticas de *cache* do tipo *write-through* ou do tipo *write-back*. Outro fator de inconsistência ocorre, em sistemas que utilizem políticas que não mantenham a memória local sempre atualizadas como no caso das políticas de *write-back e write-once*, quando um processador lê um dado armazenado na sua memória local cujo bloco tenha sido levado para outra *cache* de outro processador e este já tenha escrito neste bloco. Neste caso o processador que possui o dado em sua memória local ao buscá-lo estará trazendo um dado antigo não atualizado.

Alguns sistemas multiprocessados resolvem este tipo de problema de consistência de dados não permitindo que as memórias locais dos processadores sejam acessadas por outros processadores, como no caso do New York Ultracomputer (NYU) da New York University [GOTT83]. Em outros sistemas, como o Research Parallel Processor (RP3) da IBM [PFIS85] [PFIS86], os dados pertencentes à memória local de um processador que devem ser acessados por outros processadores não são armazenáveis em *cache*.

No caso do MULTIPLUS, optou-se por permitir que a memória local

seja compartilhada por todos os processadores do sistema e por maximizar as possibilidades de armazenamento em *cache* de dados, mesmo os compartilhados, e ao mesmo tempo diminuir ao máximo a utilização dos barramentos externos.

As operações entre um processador e sua memória local não são normalmente monitoradas por outros controladores de *cache* relativos a outros processadores, a não ser que estas sejam realizadas via barramento Mbus externo, que é comum a todos os Nós de Processamento. Por outro lado, o esquema de *write-back* obriga que as operações sejam monitoradas, pois todas as operações de memória podem mudar o estado dos dados relativos a esta operação que por ventura estejam armazenados em outras *caches* neste determinado momento. Felizmente existe um artifício que pode separar os acessos à memória que certamente não irão mudar o estado de nenhum dado armazenado em outra *cache* dos que talvez mudem o estado ainda internamente ao MASTER (aquele que gerou, ou ainda irá gerar o acesso). Este artifício, descrito a seguir, possibilita que alguns acessos sejam realizados internamente ao NP, sem tornar incoerente a base de dados e sem a necessidade de interferência de *software*.

Para poder separar os acessos que certamente não irão mudar o estado das outras *caches* daqueles que podem, foi incluído um sinalizador de compartilhamento em cada linha de *cache* na memória principal local indicando se aquela linha já foi utilizada por algum outro *cache* no mesmo *cluster*. Acessos não armazenáveis na *cache* não mudam o estado deste sinalizador.

Inicialmente, o estado do sinalizador está desligado, simbolizando que a linha não está presente em nenhuma outra *cache* do mesmo *cluster*. Leituras ou escritas no modo coerente (do tipo *COHERENT READ* e *COHERENT WRITE*) originadas por um controlador de *cache* remoto e do mesmo *cluster* levam o estado do sinalizador da linha em questão para ligado. O estado do sinalizador volta a desligado quando o controlador de *cache* local endereçar esta linha com o comando de invalidação (*COHERENT INVALIDATE*) em conjunto ou não com leitura ou escrita.

Todos os endereçamentos dirigidos à memória local poderão ser executados sem a necessidade de se utilizar o barramento Mbus externo, com

exceção de acessos no modo coerente e cuja linha, na memória local, esteja com o sinalizador de compartilhamento ligado.

IV.2 - Manutenção da Coerência por "Software"

Como a memória principal do sistema (*backing memory*) é global e acha-se distribuida pelos diversos *clusters* nos seus nós de processamento e pode ainda ser acessada por qualquer processador homogeneamente, com custo variável, um esquema para manter a coerência baseada na técnica de *snooping* torna-se inviável, pois para cada transação, haveria a necessidade de ser enviada uma mensagem do tipo *broadcast* por toda a rede de interconexão, o que acarretaria um *overhead* muito grande.

Uma outra alternativa é o método proposto por Stentrön [STEN89] mas que envolve muito *hardware*, o que se traduz em aumento de custo, área de ciruito impresso e dificuldade de projeto, de implementação e de depuração. Estudos preliminares indicaram que o ganho em desempenho não compensaria o seu custo de implementação.

Uma solução que simplifica o problema em questão seria a não inclusão na *cache* de dados dos dados compartilhados de leitura e escrita que são acessados via rede de interconexão. A perda de eficiência resultante desta simplificação é minimizada com a exclusão de: todas as instruções, das variáveis de apenas leitura e, sob controle do *software*, de variáveis compartilhadas de leitura e escrita momentaneamente privadas (variáveis migratórias) deste conjunto de objetos não armazenáveis em *cache*, porque transações com estes tipos de objetos não precisam de *snoop*.

Fica a cargo do núcleo do sistema operacional programar corretamente as tabelas de gerência de memória, avisando ao controlador de *cache* quais as páginas que podem ser armazenadas em *cache* ou não, e manter a exclusão mútua das variáveis migratórias.

IV.3 - Manutenção da Coerência Durante E/S

Os esquemas de manutenção de coerência discutidos anteriormente só garantem a consistência dos dados armazenados em *cache* para transações entre NP's. Será discutido agora a manutenção de coerência em relação à E/S e à rede de interconexão.

A E/S no sistema MULTIPLUS, por definição de sua arquitetura, será distribuida pelos seus diversos *clusters*, cada um contendo uma interface de E/S conectada a discos, terminais, impressoras e rede do tipo Ethernet.

A dificuldade de manter a coerência durante E/S reside na possibilidade da substituição de uma página de memória por outra de outro tipo, como, por exemplo, uma página de código de uma tarefa que já foi terminada ser substituida por outra página de dados de outra tarefa em execução.

Uma outra substituição problemática ocorre quando uma página de dados de leitura e escrita é trocada por uma página de instruções utilizando o barramento de instruções para carga do novo bloco de instruções. O *snoop* do controlador de *cache* de dados de um NP remoto, mas do mesmo *cluster*, não perceberia a transação e não invalidaria linhas da sua *cache* que por ventura ainda contivessem algum dado referente àquela página. Embora o processador deste controlador de *cache* não fosse mais se referir a esta página como dado, uma ação de *write-back* poderia ocorrer para a substituição de uma linha com um dado na *cache* no estado *dirty*, por uma linha de alguma outra página válida de dado. Este *write-back* escreveria dados antigos por cima de uma área de instruções atualmente válida na memória principal.

Para contornar este tipo de problema, o Núcleo do Sistema Operacional deverá forçar um *write-back* através de um comando de *flush* de página para cada página de dados ou instruções liberada. Após esta operação, toda transferência de E/S nunca levará a um acerto do *snoop* do *cache*, possibilitando que as transferências de E/S sejam realizadas através de qualquer um dos barramentos, independentemente de serem blocos de dados ou instruções.

Com esta liberdade, o Sistema Operacional e a interface de E/S (PES - Processador de Entrada e Saída) podem ser simplificados, tratando toda E/S como sendo transferência de blocos de dados, eliminando assim a necessidade, para o Sistema Operacional, de discriminar se o bloco é de dado ou instrução.

Em resumo, toda E/S será feita pelo barramento de instrução Mbus por ser o barramento menos requisitado (ver simulações). A cada liberação de uma página pelo Sistema Operacional, o seu núcleo se encarregará de enviar um comando de *flush* para o controlador de *cache* invalidando todas as posições na *cache* ocupadas por objetos das páginas liberadas.

IV.4 - Manutenção da Coerência com a Rede de Interconexão

Como foi mostrado no item 4.2, variáveis que podem ser acessadas pela rede de interconexão não podem, em primeira análise, ser armazenadas em *cache*; mesmo nas *caches* do mesmo *cluster* destas variáveis.

O problema da coerência aparece quando se utiliza o método de *write-back* e armazena-se na *cache* somente variáveis locais ao *cluster* mas permitindo-se que estas possam também ser acessadas através da rede por outros NP's de outros clusters. É possível que haja perda da coerência da base de dados quando um bloco armazenado em alguma *cache* do *cluster* **i** pertencente a uma memória deste mesmo *cluster* estiver em algum dos estados **DIRTY**, e um NP de outro *cluster* **j**, **i** ≠ **j**, necessitar realizar uma operação de escrita em algum segmento deste bloco. Por ser uma operação de escrita de apenas um segmento de um bloco, esta não poderá ser realizada no modo **COERENTE**[1], logo não será monitorada pelo controlador de *cache* pela definição do protocolo implementado no controlador de *cache* utilizado.

Exitem três alternativas possíveis para contornar este tipo de problema:

1º - Não inclusão na *cache* de todas as variáveis acessáveis pela rede. Esta não inclusão elimina a possibilidade de armazenar na *cache* grande parte das variáveis.

2º - Utilização do método de *write-through*: neste método as operações de escrita no modo COERENTE não necessitam endereçar um bloco de *cache* inteiro, podendo se limitar a segmentos de bloco de um a oito *bytes*.

3º - Operações de leitura, montagem e escrita (*read/modify/write*): neste caso, a IR deverá realizar uma operação de leitura no modo COERENTE com

---

[1]No modo coerente, todas as operações são necessariamente de blocos inteiros de *cache* para o esquema de *write-back*. No esquema de *write-through* esta obrigação se refere apenas a leituras.

invalidação, montar o novo bloco a ser escrito colocando o novo dado em sua posição correta dentro do bloco, para, imediatamente depois, sem liberar o barramento, em operação atômica, escrever o segmento na memória. As operações de leitura deverão ser realizadas no modo COERENTE pela IR para que, se o bloco referenciado estiver no estado **DIRTY** em alguma *cache*, o seu controlador possa tomar ciência da transferência através do seu *snoop* e fornecer o dado atualizado.

A maior parte dos sistemas que utlizam rede de interconexão, como o sistema BBN e o RP3, não permitem que dados compartilhados sejam armazenados em *cache*. Esta restrição facilita a manutenção da coerência, uma vez que a memória principal é mantida sempre atualizada e não existem cópias dos dados em diversas *caches* distribuídas pelo sistema.

As alternativas disponíveis serão simuladas para que a escolha possa ser realizada com base em comparações mais quantitativas.

IV.5 - Operações Indivisíveis

A Unidade Inteira (IU CY7C601) é capaz de realizar quatro tipos de operações indivisíveis, que são: **LDSTUB**, **LDSTUBA**, **SWAP** e **SWAPA**. As duas primeiras lêem um *byte* da memória e o escrevem com todos os *bits* ligados, e as duas últimas leêm uma palavra e escrevem o conteúdo de um registro previamente selecionado.

Este tipo de instrução é particularmente utilizada em sicronização de processos e também para garantir exclusão mútua em recursos que não podem ser compartilhados simultaneamente.

Por definição da arquitetura do Sistema Operacional do MULTIPLUS, as sincronizações serão feitas por uma fila de processos e um ponteiro para a próxima posição vaga. Cada processo que necessitar um determinado recurso deverá ler o valor atual do ponteiro da fila e incrementá-lo para que este possa apontar para a próxima posição vaga. Na fila, o processo apenas escreverá o seu número na posição que foi lida do apontador. Quando o processo que estiver utilizando atualmente o recurso for liberá-lo, este deverá consultar a fila para descobrir qual é o próximo processo e avisá-lo.

Este método elimina o chamado *hot-spot* que se formaria na variável de sincronização e o distribui para as memórias locais dos processadores.

Uma restrição a este método estaria na instrução de ler e incrementar o ponteiro para a fila que deverá ser atomicamente executada, que não é disponível no conjunto de instruções do CY7C601. A alternativa que deverá ser utilizada será a inclusão de um somador que armazenará o dado lido nos ciclos atômicos (chamados de ciclos de *load-store*) que somará a este uma constante para que o dado a ser escrito possa ser substituído pelo resultado. Este esquema tornará, em contra-partida, as instruções de **LDSTUB**, **SWAP**, **LDSTUBA** e **SWAPA** equivalentes.

As operações atômicas, além da indivisibilidade que é garantida pelo processador (este não aceita interrupções e pedidos de barramento durante suas execuções), deve ser garantida também na memória principal

através do seu árbitro que deve assegurar a posse do barramento ao processador até o fim do ciclo. Se dois processadores tentarem executar ao mesmo tempo instruções indivisíveis na mesma palavra de memória, o árbitro deve providenciar para que tudo se passe como se um processador executasse a instrução e somente depois de concluí-la, o outro a iniciasse, ou seja, o resultado deverá ser o mesmo para execuções em "paralelo" ou puramente sequenciais. Este tipo de tarefa não é de dificil realização, uma vez que todas estas ações irão necessitar de pelo menos um caminho em comum que será mutuamente exclusivo.

No caso em que os processadores possuem *cache* local, como no MULTIPLUS, esta última afirmação não é sempre verdade. Há a possibilidade de que a palavra na qual a operação vá se realizar esteja presente em pelo menos uma das *caches* de um dos processadores. Neste caso, enquanto um estiver lendo a palavra de sua própria *cache*, o outro a estaria lendo da memória principal (ou da sua *cache* também). Ambos iriam incrementar o mesmo valor e escrevê-lo de volta. Embora os controladores de *cache* garantam a manutenção da coerência do dado, a exclusão mútua não foi mantida.

No esquema de manutenção de coerênca do tipo *write-through*, implementado neste controlador de *cache*, somente existe previsão para garantir a exclusão mútua através da execução serial de instruções atômicas caso seja utilizado o controlador de *cache* CY7C604, que é exclusivo para sistemas monoprocessados. Uma solução viável poderia ser a alocação dos dados de sincronismo em páginas mapeadas como *not caching* (não passíveis de armazenamento em *cache*).

O método *write-back* nestes controladores de *cache* não prevê nenhum esquema para garantir a exclusão mútua. A solução mais imediata é a mesma do caso de *write through*.

Embora a alocação das variáveis de sincronização em páginas *not caching* seja viável, é de difícil implementação inicial porque na primeira fase do projeto deve ser utilizado o compilador de alguma máquina comercial baseada na arquitetura *SPARC* e muito provavelmente este compilador não deverá fornecer a facilidade de declarar variáveis como sendo do tipo *not*

caching.

Uma facilidade da IU poderá ser utilizada para garantir a "sequencialidade" dos ciclos. A IU possui dois sinais para sinalizar ciclos atômicos (**LOCK** e **LDSTO**). Estes sinais são ativados um ciclo de relógio antes do ciclo atômico iniciar e podem ser utilizados para colocar a IU no estado de espera (**HOLD**) enquanto um pedido de aquisição de barramento de dados é gerado. Como o estado de **HOLD** foi gerado externamente, os controladores de *cache* e outros dispositivos conectados diretamente ao barramento virtual devem ser sinalizados através da ativação do sinal **SNULL**.

A IU continuará no estado de **HOLD** até que a posse do barramento esteja garantida. O barramento será mantido preso (*locked*) até o fim da escrita do ciclo atômico corrente.

A natureza do algoritmo de sincronização não favorece a um acerto da variável de sicronização na *cache*, necessitando que esta seja trazida da memória principal na maioria das vezes. A baixa re-utilização e o alto custo da inclusão de um bloco na *cache* leva a optar por separar as variáveis de sincronização como não armazenáveis em *cache*, mesmo quando utilizadas somente dentro do mesmo *cluster*.

O apêndice C mostra o custo, em ciclos de relógio, de diversos casos de sincronização com instruções atômicas. Pode-se observar que a ação de trazer uma linha para a *cache* requer quatro acessos à memória e que mesmo utilizando-se o modo de troca de dados entre *caches* sem utilizar a memória principal, o tempo gasto ainda é muito grande não compensando armazenar a variável em *cache*.

O somador para computar o resultado da operação indivisível deverá estar o mais próximo possível da memória para que possa ser utilizado tanto localmente quanto pela rede.

As operações indivisíveis podem ser separadas em duas partes que são: leitura e escrita.

Não é necessário fazer o processador esperar que o dado seja escrito na segunda parte da operação, principalmente quando esta for realizada pela rede, o que significaria um gasto de tempo inútil. O próprio circuito controlador de memória poderia, ao ser informado que uma leitura (primeira parte) de uma operação indivisível está sendo realizada, incrementar este valor e escrevê-lo na memória. A escrita da IU seria perdida e executada apenas a nível do NP.

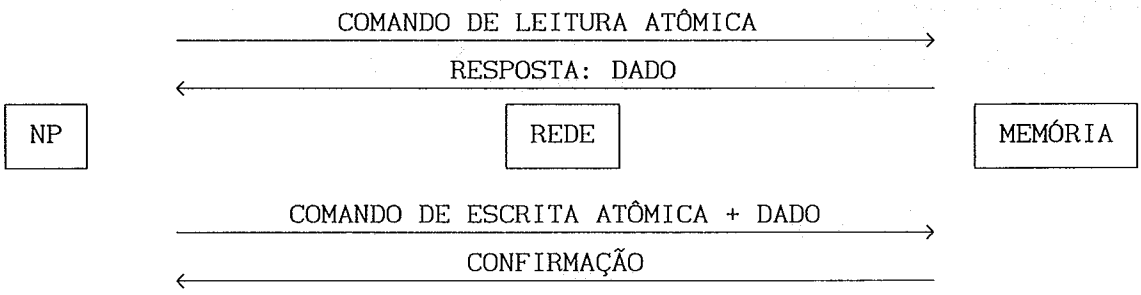


figura 4.1.a: instrução atômica executada tradicionalmente pela rede

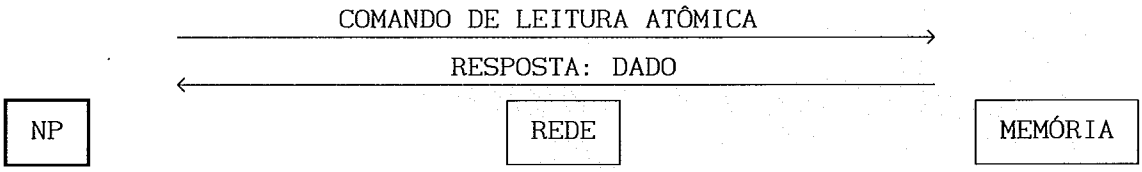


figura 4.1.b: instrução atômica com execução modificada pela rede

Conforme as figuras 4.1.a e 4.1.b demonstram, este método diminui para a metade o tempo de uma instrução atômica convencional.

CAPÍTULO V

SIMULAÇÕES

Para avaliação de diferentes alternativas de implementação da arquitetura do MULTIPLUS, e mais especificamente, da arquitetura de memória *cache*, foi desenvolvido um simulador funcional capaz de analisar o desempenho do MULTIPLUS variando-se o número de barramentos externos (1 ou 2), a política de atualização da memória principal (sem *cache*, *write through*, e *copy back*) e o uso ou não de *write buffer* para tentar igualar o sistema *write through* ao *copy back*, de desempenho reconhecidamente superior em sistemas monoprocessados com processadores de arquitetura CISC.

O simulador foi escrito em Pascal e roda em microcomputadores do tipo IBM PC/XT, AT ou 386. A versão final do simulador possui 200 K*bytes* de fonte e 60 K*bytes* de código.

Para cada configuração serão extraídos diversos parâmetros de desempenho que são:

■ **Tempo Médio de Ciclo de Processador:** mede o tempo médio de acesso à memória de cada processador. O resultado é apresentado como a média dos tempos médios e obtido, por simplificação, pela seguinte equação:

$$\overline{TM} = \frac{\#NP}{\sum_{i=1}^{\#NP} \frac{\sum_{j=1}^{\#CICL} C(i;j)}{\#CICL}} \cong \frac{\#NP \,.\, \#CICL}{\sum_{i=1}^{\#NP} \sum_{j=1}^{\#CICL} C(i;j)}$$

onde: $\overline{TM}$ = tempo médio de ciclo

#NP = número de nós de processamento

#CICL = número de ciclos de relógio da simulação

C(i;j) = 1, se o processador i terminou um ciclo no tempo j

C(i;j) = 0, se o processador i não terminou um ciclo no tempo j

Foi utilizada a segunda equação uma vez que o simulador prevê uma máquina homogênea; logo, o número de ciclos executados por cada processador tende a ser igual.

Com este resultado, espera-se poder conhecer o grau de interferência de um processador em outro do seu *cluster* e no resto do Sistema.

■ **Total de Ciclos Executados:** mede o total de ciclos executados por todos os NP's. É obtido pela seguinte equação:

$$TC = \sum_{i=1}^{\#NP} \sum_{j=1}^{\#CICL} C(i;j)$$

onde: TC = total de ciclos
#NP = número de nós de processamento
#CICL = duração, em ciclos de relógio, da simulação
C(i;j) = 1, se o processador i terminou um ciclo no tempo j
C(i;j) = 0, se o processador i não terminou um ciclo no tempo j

Sem perda da generalidade, o resultado foi normalizado dividindo-se o valor obtido pelo número de ciclos de simulação (tempo de simulação) para compatibilizar simulações com tempos totais diferentes.

$$TC = \frac{\sum_{i=1}^{\#NP} \sum_{j=1}^{\#CICL} C(i;j)}{\#CICL}$$

Idealmente, o total de ciclos executados dentro de um determinado intervalo de observação deve dobrar com o número de NP's.

■ **"Throughput" no Barramento de Dados/Instruções:** mede a vazão, em número de transações no barramento que interliga os NP's e a IR. Tem por finalidade obter o número máximo de NP's que podem ser incluídos em um *cluster* sem

saturar este canal.

É esperado que o número de acessos ao barramento aumente proporcionalmente ao número de processadores. O *throughput* obtido na simulação de um processador por *cluster* não deve ser considerado, uma vez que não são realizados acessos de manutenção de coerência de *cache*, de compartilhamento de dados e leitura de dados/instruções em outros NP's do mesmo *cluster*.

O resultado é apresentado como a média do número de acessos pelos diversos barramentos do sistema, conforme equação:

$$\overline{TB} = \frac{\sum_{i=1}^{\#CLUSTER} \frac{\sum_{j=1}^{\#CICL} C(i;j)}{\#CICL}}{\#CLUSTER} \cong \frac{\sum_{i=1}^{\#CLUSTER} \sum_{j=1}^{\#CICL} C(i;j)}{\#CLUSTER \;.\; \#CICL}$$

onde: $\overline{TB}$ = *throughput* médio do barramento

#CLUSTER = número de *clusters*

#CICL = número de ciclos de relógio da simulação

C(i;j) = 1, se iniciou-se um ciclo no barramento i no tempo j

C(i;j) = 0, se não iniciou-se um ciclo no barramento i no tempo j

■ **Taxa de Ocupação da Memória Local, Barramento de Dados e Barramento de Instruções:** medem o tempo que o recurso esteve ocupado durante o tempo de simulação. Também é apresentado como a média de todos os recursos do mesmo tipo.

■ **Tempo de Espera por Recursos:** será medido o tempo médio de espera do processador (no caso, o controlador de *cache*) pelo barramento de instruções e pelo barramento de dados. O resultado é expresso pela seguinte fórmula:

$$\overline{Tw} = \frac{\sum_{i=1}^{\#NP} \frac{\sum_{j=1}^{\#CICL} Ew\ (i,\ j)}{\sum_{j=1}^{\#CICL} C\ (i,\ j)}}{\#NP} = \frac{\sum_{i=1}^{\#NP} \sum_{j=1}^{\#CICL} Ew\ (i,\ j)}{\#NP * \sum_{i=1}^{\#NP} \sum_{j=1}^{\#CICL} C\ (i,\ j)}$$

onde: $\overline{Tw}$ = tempo médio de espera

#CICL = duração, em ciclos de relógio, da simulação

#NP = número de nós de processamento

Ew (i, j) = 1 se o processador i estava esperando pelo recurso no tempo j

Ew (i, j) = 0 se o processador i não estava esperando pelo recurso no tempo j

C (i, j) = 1 se o processador i ganhou o recurso no tempo j

C (i, j) = 0 se o processador i não ganhou o recurso no tempo j

A taxa de ocupação dos barramentos, aliada ao *throughput*, é um importante parâmetro para se obter a configuração máxima do sistema suportada pelos barramentos.

V.1 - Descrição do Simulador

Esta seção descreve o programa utilizado para simular os sistemas em questão.

Primeiramente é necessário que se conheçam algumas definições utilizadas neste texto:

- **Recurso:** são todos os objetos necessários para que um acesso de um processador se complete. Os elementos que sempre são alocados simultaneamente foram agrupados sob o mesmo nome, como por exemplo, o *snoop* e o barramento imediatamente conectado a este, sendo, então, o conjunto chamado apenas pelo nome de **SNOOP**.

- **Processador:** é o elemento com capacidade de processamento próprio do sistema. No caso do **MULTIPLUS**, a Unidade de processamento Inteira (IU).

- **Estado:** é um atributo relativo aos processadores e recursos. Os estados são divididos em três grandes grupos, a saber: **FREE** (livre), **W** (relativo somente aos processadores, composto por diversos estados, simbolizando que o processador está parado (*wait*), esperando pela alocação dos recursos necessários para que este realize o acesso correspondente) e **X** (também composto por diversos estados, tem relação com o estado **W** que o antecedeu, e é relativo a processadores e recursos em geral que estejam correntemente executando um acesso). Para maiores detalhes, ver apêndice B.

V.1.1 - Alocação de Processadores

Inicialmente, quando a máquina é ligada, todos os processadores estão no estado disponível (FREE), simulando um estado inicial do tipo *reset*. Cada processador é então alocado para executar um tipo de tarefa que consiste basicamente de um acesso de dado ou instrução. Embora, inicialmente, em um sistema real, a *cache* estivesse vazia e o primeiro acesso do processador fosse necessariamente de busca de instrução, para fins de simulação, é considerado que a *cache* já está com taxa de acerto

compatível com o seu estado de regime, eliminando-se assim problemas de estado transitório para que o número de iterações possa diminuir. Da mesma forma, o sinalizador de compartilhamento e o tipo de acesso a ser executado também iniciam a simulação no tempo como se já estivessem em regime. A alocação de tarefas (acessos) aos processadores é feita levando-se em conta apenas os percentuais dos tipos disponíveis de acessos conforme definição do usuário através dos parâmetros de entrada (a ser explicado detalhadamente na seção 5.2).

V.1.2 - Alocação de Recursos

Os diversos recursos disponíveis a cada processador são: o *cache*, o *snoop* e a memória local de seu NP; o barramento de dados, o barramento de instrução, as memórias locais e o *snoop* de NP's de seu *cluster*; e o barramento de dados, o barramento de instruções, a memória local e o *snoop* de NP's de outros *clusters*.

No caso de sistemas sem *cache*, não há necessidade de se alocar nenhum *cache* e *snoop*. E, em simulações de sistemas com apenas um barramento, o barramento de dados passa a ser também o de instruções, sendo, portanto, chamado de barramento único ou, somente, barramento.

Durante o acesso, em um sistema real, de um processador à memória de um outro NP no seu *cluster*, são alocados primeiramente o *cache* e o *snoop* deste processador e somente no ciclo de relógio seguinte é que o barramento externo é alocado para então alocar-se a memória e o *snoop* do outro NP, com uma defasagem de dois ciclos de relógio. Por simplificação, a alocação de recursos é feita no instante em que o processador passa do seu estado W (esperando) para o estado X (executando) sendo mantido os seus respectivos tempos que passam alocados conforme exemplo a seguir (figura 5.1)[1].

---

[1] O leitor pode notar que estes tempos se referem apenas a tempos de propagação e de protocolos. Não confundir com os tempos de arbitração que foram simulados.

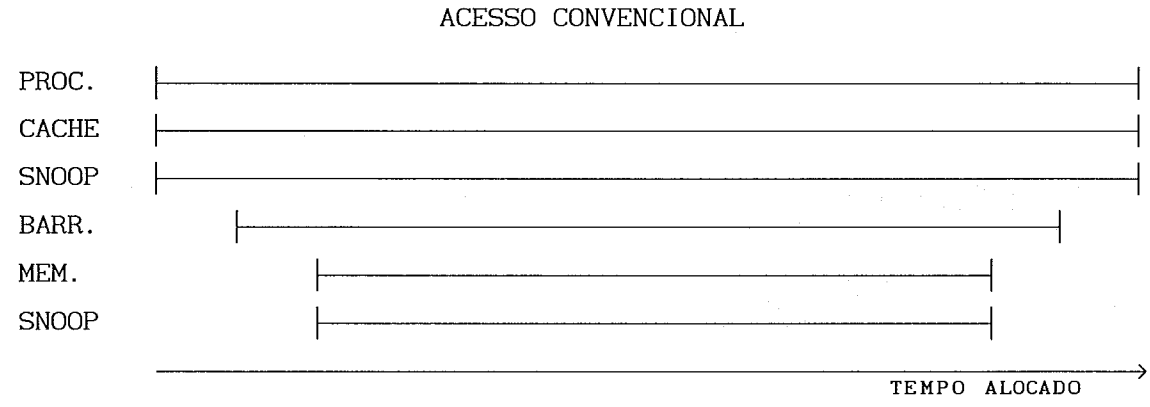


figura 5.1.a: diagrama de ciclo de barramento normal

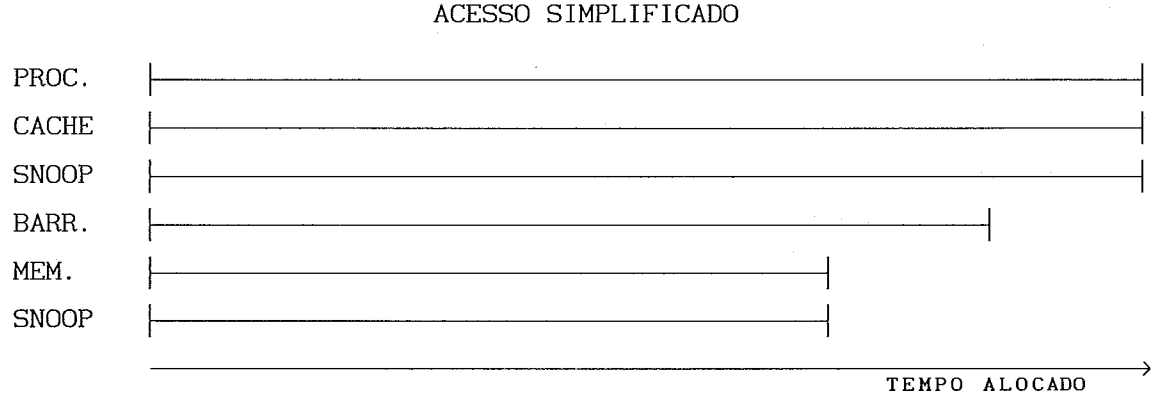


figura 5.1.b: diagrama de ciclo de barramento simplificado

Cada recurso possui dois atributos. O primeiro atributo indica o seu estado atual que pode ser FREE (livre) ou qualquer estado X (executando um acesso). O segundo atributo, que somente é válido quando o primeiro estiver no estado X, representa o tempo necessário para que o acesso corrente termine.

V.1.3 - Fluxo do Simulador

O procedimento executado pelo simulador pode ser dividido em três grandes partes que são: inicialização de variáveis, iteração de execução e contabilização. A iteração de execução, que constitui o núcleo do simulador, é composta basicamente de quatro rotinas:

■ **Rotina de Alocação de Tarefas:** aloca para cada processador do sistema, que esteja no estado FREE, uma tarefa, ou seja, um tipo de acesso à memória, com distribuição aleatória através de um gerador de números aleatórios variando de 0%, inclusive, a 100%, exclusive. Este número é comparado com o percentual estipulado pelo usuário no início da simulação, como por exemplo, taxa de acessos de busca de instrução, leitura de dados e escrita de dados (ver seção 5.2):

```
if random < percentual de acessos para busca de código then
    rotina de escolha do tipo de acesso para busca de código
else if random < percentual de leitura de dado then
        rotina de escolha do tipo de acesso para leitura de dado
    else
        rotina de escolha do tipo de acesso para escrita de dado
```

Pode-se notar que uma vez que o valor retornado pelo gerador de números aleatórios for, na sua primeira chamada do exemplo acima, maior que o percentual de busca de código, o acesso será de leitura ou escrita de dado.

Ao sair desta rotina, todos os processadores estarão ou no estado W, no caso de processadores recém-alocados, ou no estado X, já em execução.

■ **Rotina de Alocação de Recursos:** uma vez que todos os processadores já possuem tarefas para executar, todos os que ainda estiverem no estado W procurarão alocar todos os recursos necessários para que possam iniciar o ciclo de acesso de acordo como seu tipo de estado. O processador somente muda de estado W para X quando conseguir alocar todos os recursos necessários.

Uma vez alocado, o recurso passa do estado FREE para o estado X correspondente ao estado W do processador que o alocou. O seu contador de tempo de acesso é carregado com o tempo total que o recurso passará alocado durante este tipo de acesso.

■ **Rotina de Tempo:** este é o procedimento que emula a passagem do tempo para

o sistema. Em todos os processadores e recursos que estejam em um estado W, executando um acesso, o contador de tempo é decrementado de uma unidade e comparado ao valor zero. Caso seja igual a zero, o recurso ou processador é colocado no estado FREE.

■ **Rotina de Totalização Parcial:** em cada uma das rotinas descritas acima, contadores auxiliares são atualizados para que esta rotina possa contabilizar os diversos parâmetros de saída do simulador. A rotina de totalização geral é chamada ao final da simulação totalizando os resultados obtidos pelo procedimento anterior, e, opcionalmente, salva em disco os resultados para um posterior trabalho de comparação, tração de gráficos, geração de tabelas, etc.

V.1.4 - Modelos de Política de "Cache" no Simulador

Inicialmente estavam previstos quatro tipos de política de atualização da memória principal para serem simuladas, a saber: sem *cache*, *write through*, *copy back* e *write once*. Esta última política, *write once* [GOOD83], foi abandonada por se tratar de uma adaptação do tradicional *copy back*, que embora apresente desempenho superior em determinadas aplicações, é muito dependente do protocolo de barramento utilizado, e também pela falta de maiores informações sobre as características típicas de certos parâmetros que serviriam como entrada para o simulador.

Serão discutidos agora o algoritmo implementado para cada uma das políticas utilizadas.

V.1.4.1 - Sem "Cache"

Esta arquitetura foi simulada apenas para mostrar a influência da hierarquia de memória na redução do uso do barramento comum aos NP's. Nesta arquitetura só existem três tipos de acessos: à memória local do NP, ao *cluster* e a outro NP de outro *cluster*. Os acessos ainda podem ser de busca de instrução, leitura de dado ou escrita de dado.

V.1.4.2 - "Write Through"

Esquema de *cache* muito popular. Aqui foi implementado sem alocação de blocos em escritas (*without write allocate*) como ocorre no controlador de *cache* Cypress CY7C605 [CYPR89a].

Algumas restrições para manter a coerência foram feitas, assim como declarar como não possível de se armazenar em uma determinda *cache* de um NP, variáveis (dados) que são acessadas por este pela rede de interconexão. A busca de instruções é sempre realizada trazendo-se para a *cache* um bloco inteiro, mesmo através da rede, porque a manutenção da coerência fica, neste caso, a cargo do Sistema Operacional.

Os acessos do processador à sua memória local se processam sem a utilização do barramento externo, com exceçao de escrita de dados no modo coerente (de variável armazenada em *cache*), quando o sinal de compartilhamento deste dado na memória local estiver ligado. Neste caso, o processador é obrigado a realizar, em paralelo com a escrita na memória local, um ciclo de invalidação de bloco no barramento externo.

Uma característica deste método é que os blocos substituídos nunca precisam ser escritos na memória principal (*copy-back*).

V.1.4.3 - "Copy Back"

Segue também a implementação da Cypress [CYPR89a] com alocação de blocos em escrita (*write allocate*). A restrição de não armazenar em *cache* variáveis acessadas via rede imposta na implementação anterior foi extendida também a todas as variáveis (dados) que podem ser acessadas por um processador em outro *cluster*. A variável não pode, inclusive, ser armazenada na *cache* do processador que a contém em sua memória local.

A manutenção da coerência da *cache* de instruções continua a cargo

do Sistema Operacional, possibilitando que toda instrução trazida da memória principal seja armazenada em *cache*, mesmo quando a busca utilizar a rede de interconexão.

Os acessos do processador à sua memória local somente precisarão do barramento externo quando estes forem de dados, no modo coerente e o sinal de compartilhamento do bloco referenciado estiver ligado.

Quando um bloco é trazido da memória principal, é simulada a escrita do bloco antigo (alterado) na memória principal (*write back*). A necessidade desta escrita tem probabilidade especificada pelo usuário e reflete o percentual de blocos alterados na *cache*.

V.2 - Parâmetros de Entrada

Nesta seção serão apresentados os parâmetros de entrada para o programa de simulação. Ao mesmo tempo que os parâmetros são apresentados, serão fornecidos exemplos de parâmetros típicos para os sistemas simulados para servir de referência.

Para melhor exemplificação, os parâmetros foram divididos em dois grandes grupos: parâmetros de *hardware*, dependentes apenas do *hardware* do sistema a ser simulado, e parâmetros do sistema, dependentes do tipo de política de *cache* implementada, da hierarquia de memória e, principalmente, da carga de trabalho a ser submetida a simulação.

V.2.1 - Parâmetros de "Hardware"

Os parâmetros de *hardware* são de três tipos: de temporização, de configuração e de arquitetura do sistema.

Com os parâmetros de arquitetura do sistema é possível configurar o número de barramentos (1 ou 2) que interligam os NP's, o número de NP's (1 à 16) por *cluster*, o número de *clusters* do sistema (1 à 256) e a profundidade do *buffer* de escrita (*write buffer*).

As limitações de valores dos parâmetros são inerentes às restrições do protocolo Mbus [CYPR90b], que suporta no máximo 16 unidades endereçáveis em seu barramento, e à interface de rede, que reservou 8 *bits* para endereçar o *cluster* origem e/ou destino da mensagem.

Os parâmetros de temporização são referentes aos tempos de acesso aos diversos níveis da hierarquia de memória. Para poder simular acessos em rajada (*burst*), os tempos de acesso foram separados em tempo entre acessos, que é o tempo de acesso em rajada (*burst time*), e o tempo para se acessar uma palavra menos o tempo de acesso (tempo entre **RAS** e **CAS**). A unidade de tempo é o período de relógio do processador do NP, que foi selecionado para 40MHz, ou seja 25ns.

■ **Acesso à RI:** o atraso da rede de interconexão é fornecido por estágio e foi calculado em aproximadamente três períodos de relógio [BRON90].

■ **Acesso ao Barramento:** um acesso através do barramento Mbus que interliga os NP's pode ser dividido em três fases [CYPR90a]: de aquisição de barramento, de endereço e de dado. A fase de aquisição de barramento tem tempo variável e difícil de ser estimado, mas é simulado pelo árbitro do Simulador. A fase de endereço (figura 5.2) possui tempo fixo e igual a uma unidade de tempo. A este tempo é somado o tempo gasto para que o acesso gerado pelo *cache* seja decodificado dentro do NP, o que consome quatro ciclos de relógio, que são distribuídos entre os seguintes eventos: acionamento pelo *cache* das linhas de acesso, a fase de endereço interna ao NP e a decodificação propriamente dita. A fase de dados é ditada pelo tempo de acesso do *slave* da operação e é proporcional ao número de palavras que é transferido.

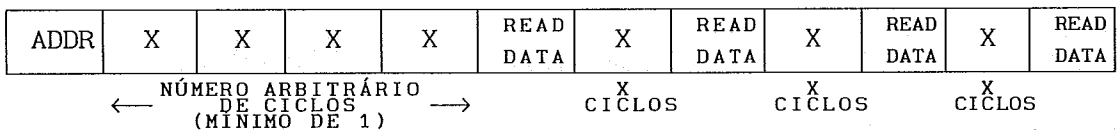


figura 5.2.a: leitura no Mbus externo

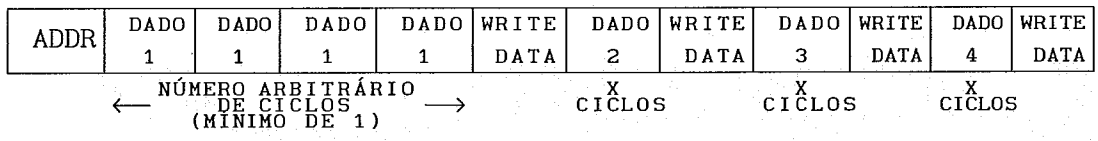


figura 5.2.b: escrita no Mbus externo

■ **Acesso à Memória Local:** como já foi mencionado, o tempo de acesso foi dividido em duas partes (figura 5.3) para refletir o acesso no modo rajada de memórias dinâmicas [MICR89] a serem utilizadas. Dependendo do tempo de acesso da pastilha de memória, que atualmente[2] pode ser de 80ns a 120ns, o tempo de acesso (*skeel time*) e o tempo entre acessos (*burst time*) podem variar. Supondo-se uma memória com tempo de acesso igual a 100ns, o tempo

---

[2] Durante a escrita deste trabalho foram recebidos manuais contendo versões preliminares de memórias dinâmicas com tempo de acesso de 50ns.

entre acessos no modo rajada (*burst*) é de aproximadamente dois períodos de relógio. É necessário mais um período para ser calculado o código de correção (ECC), totalizando três ciclos de relógio. O tempo que antecede o primeiro acesso é igual a aproximadamente três ciclos de relógio.

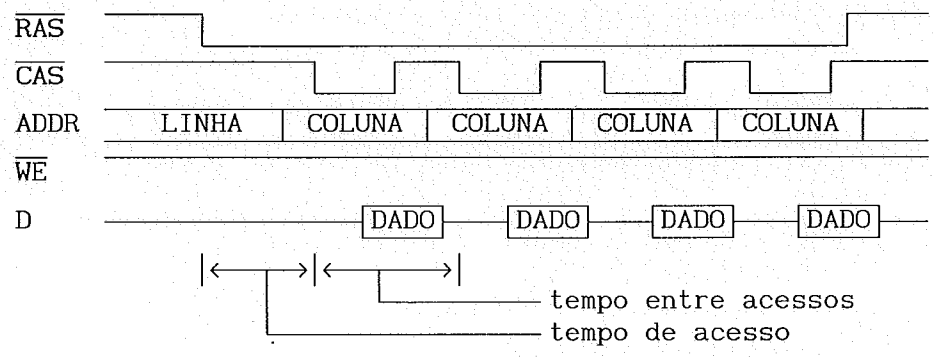


figura 5.3.a: leitura em rajada de memória dinâmica

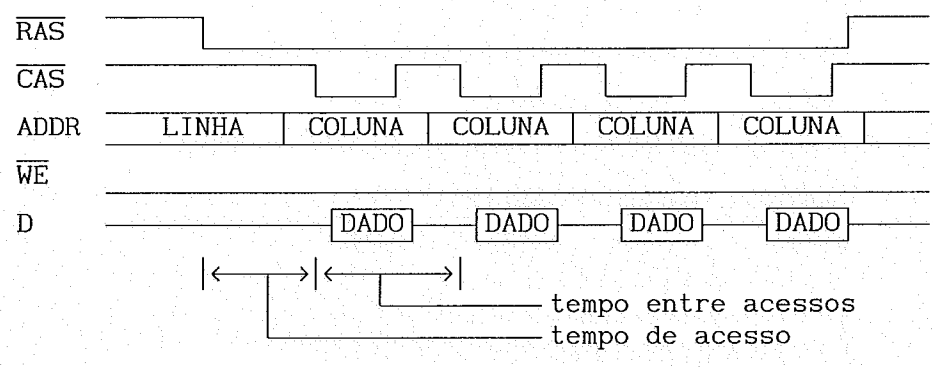


figura 5.3.b: escrita em rajada de memória dinâmica

■ **Acesso à "Cache":** o tempo de acesso à *cache* é de um ciclo de relógio para leitura e de dois ciclos para escrita de acordo com a implementação da Cypress. Não existe modo rajada para a memória *cache*. Existe também um parâmetro que é o tempo necessário pelo *cache* para iniciar um acesso quando ocorre falha na *cache*.

V.2.2 - Parâmetros do Sistema

Os parâmetros do sistema são extremamente dependentes da carga de trabalho a que o sistema a ser simulado está sendo submetido, sendo, portanto, difíceis de serem estimados. Entretanto, na presente simulação, a

validade dos parâmetros do sistema são garantidos, uma vez que os valores obtidos são relativos aos outros sistemas simulados.

■ **Percentual de Acerto na "Cache":** a carga de trabalho esperada no **MULTIPLUS** é basicamente de processamento científico, onde uma grande massa de dados é tratada simultaneamente em diversas iterações, o que favorece ao acerto na *cache*. É previsto, em primeira análise, que a taxa de acerto na *cache* de instruções seja de aproximadamente 95% [SMIT82] e a da *cache* de dados em torno de 85%

A taxa de falha na *cache* no gráfico da figura 5.4 representa a figura típica de um programa; no caso, um programa de avaliação de desempenho de sistemas, sendo executado sem a interferência de outros programas ou do próprio sistema operacional.

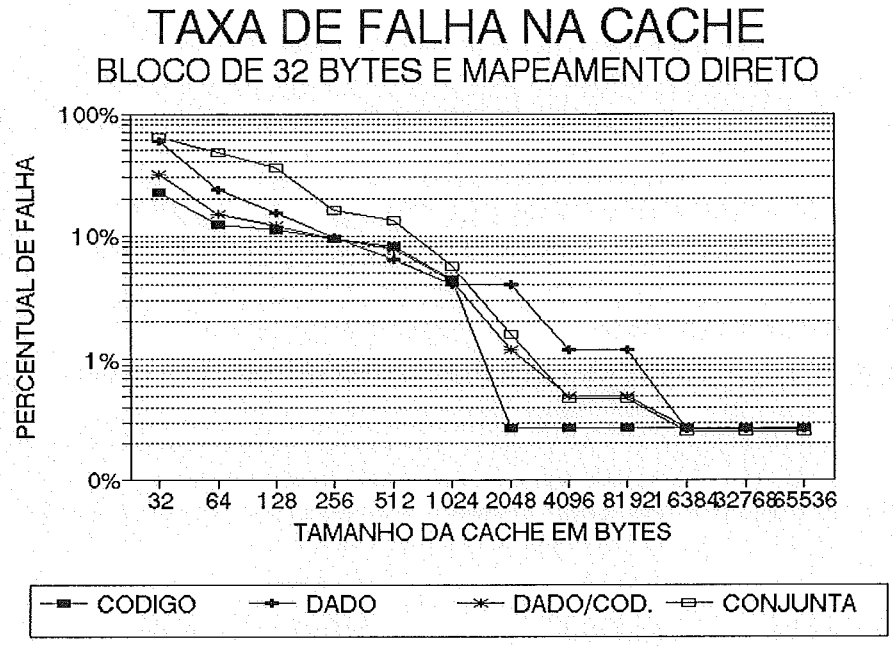


figura 5.4: taxa de falha na *cache* X tamanho da *cache*
Gráfico obtido através de simulação monitorada (*trace driven*)

■ **Percentual de Acerto no "Snoop":** é o percentual de transações de memória no barramento interno de um determinado NP que irá causar invalidação em sua *cache*. Esta taxa foi estimada em 10% do total de acessos que podem causar invalidação. Este valor foi obtido a partir do percentual de variáveis

compartilhadas. Em [WEBE89] este valor foi calculado em 1% do total de acessos. Não há *snoop* no controlador de *cache* de instruções porque a consistência dos dados armazenados é mantida pelo Sistema Operacional. O valor de 10% foi escolhido em função do alto valor do percentual de variáveis compartilhadas (ver item a seguir).

- **Percentual Compartilhado:** é o percentual de acessos à memória local de um processador que referencia variáveis compartilhadas. Em [WEBE89] foi calculado um percentual de variáveis compartilhadas de 2% enquanto que em [FEIT90] e [RUDO85], um percentual de 5%. Ambos os sistemas analisados nestes trabalhos possuem memória compartilhada mas não são tão fortemente acoplados como no caso do Sistema **MULTIPLUS**. Para simular a influência do Sistema Operacional distribuído, que possui muitas variáveis compartilhadas, e poder avaliar uma carga de trabalho não muito ideal para o sistema, o valor utilizado foi de 20%, tanto para código como para dados.

- **Percentual de Acesso Local:** é uma percentagem do total de acessos gerados pelo processador que se referem à memória local do próprio NP. Foi estimado um valor de 80% [PEIX90].

- **Percentual de Acesso no "Cluster":** possui a mesma formação do percentual de acesso local. É o percentual de acessos gerados pelo NP nos barramentos externos que não necessitam da rede de interconexão. Foi utilizado 80%, dos 20% restantes do item anterior, para dados e instruções. No caso de instruções, este percentual reflete a política a ser adotada pelo sistema operacional de transferir todas as páginas de código para a memória do *cluster* que as estejam utilizando.

- **Percentual de Leitura, Escrita de Dados e Busca de Código:** é a maior diferença entre simulações de sistemas baseados em processadores RISC e CISC. Enquanto que em processadores CISC os percentuais de leitura, escrita de dados e busca de código estão em torno de 35%, 15% e 50% respectivamente [SMIT82], os percentuais referentes a processadores RISC são aproximadamente 13%, 7% e 80% [NAMJ88] [TAMI81] (figura 5.5).

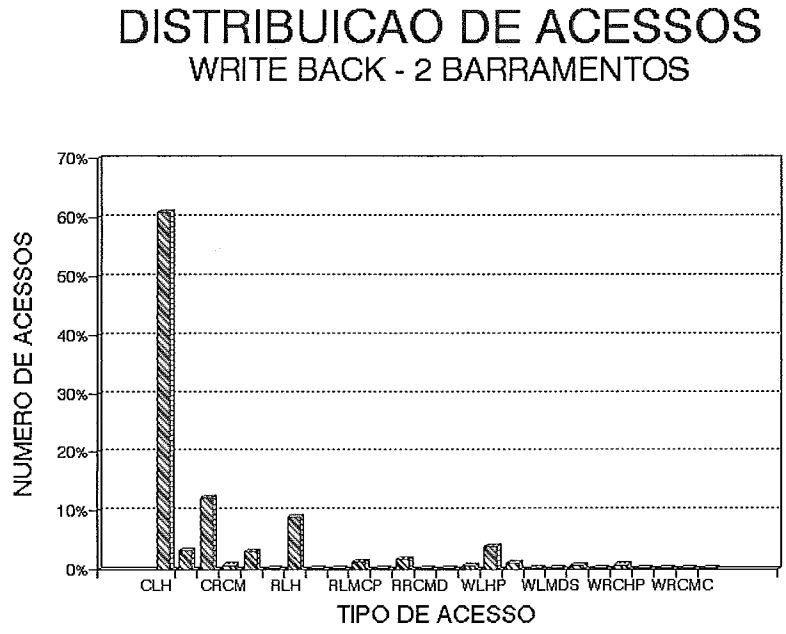


figura 5.5.a: distribuição dos tipos de ciclos de processador
(ver apêndice B para descrição dos símbolos)

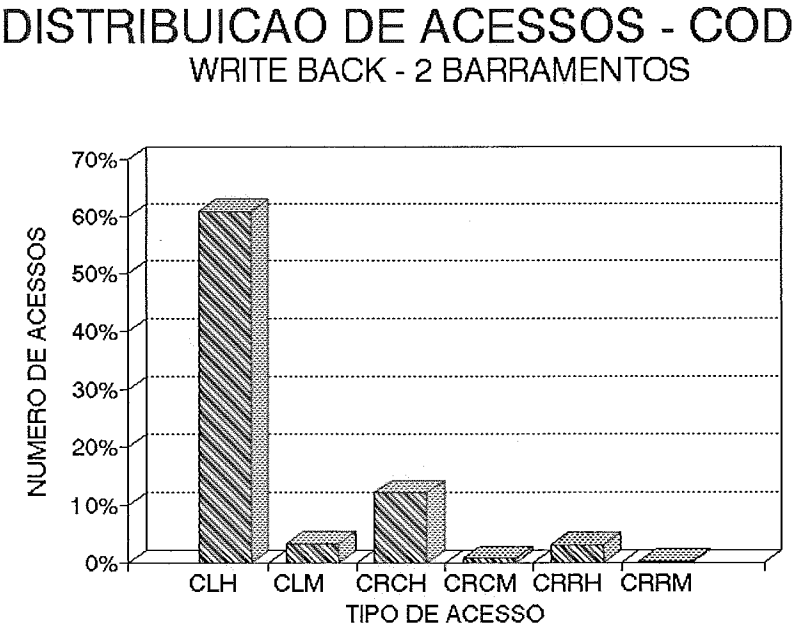


figura 5.5.b: distribuição dos tipos de busca de código do processador
(ver apêndice B para descrição dos símbolos)

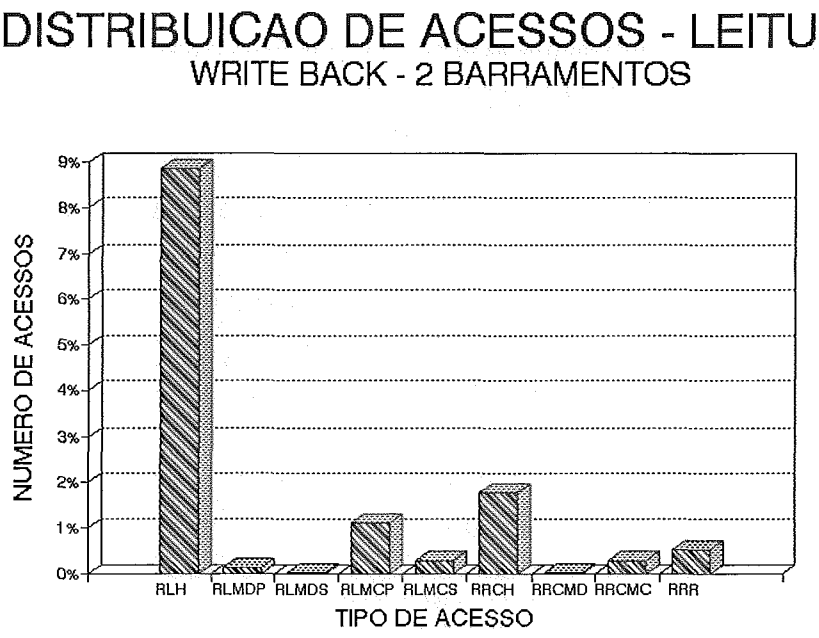


figura 5.5.c: distribuição dos tipos de leitura de dado do processador (ver apêndice B para descrição dos símbolos)

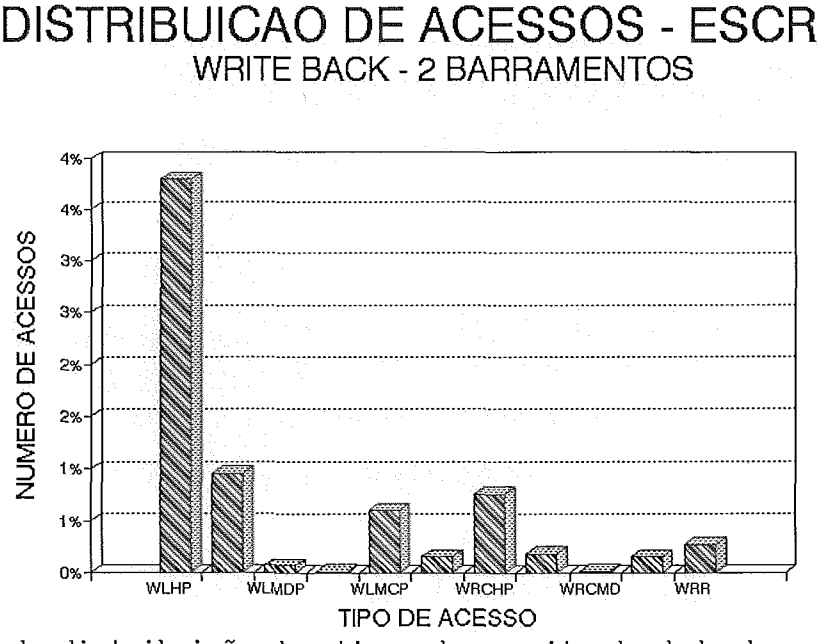


figura 5.5.d: distribuição dos tipos de escrita de dado do processador (ver apêndice B para descrição dos símbolos)

■ **Percentual Escrito:** refere se ao percentual de dados armazenados na *cache* de dados que necessitam de atualização na memória principal (*copy back* ou *write back*) quando forem substituídos por outro bloco na *cache* ou por um

pedido de atualização de outro *cache*. Seu valor foi estimado em 10% (figura 5.6).

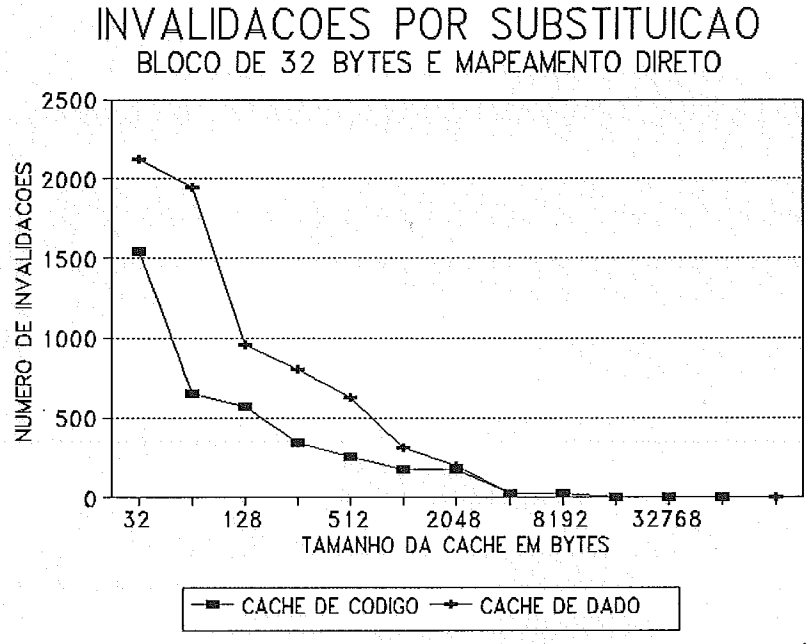


figura 5.6.a: tamanho da *cache* X número de *write-back*'s

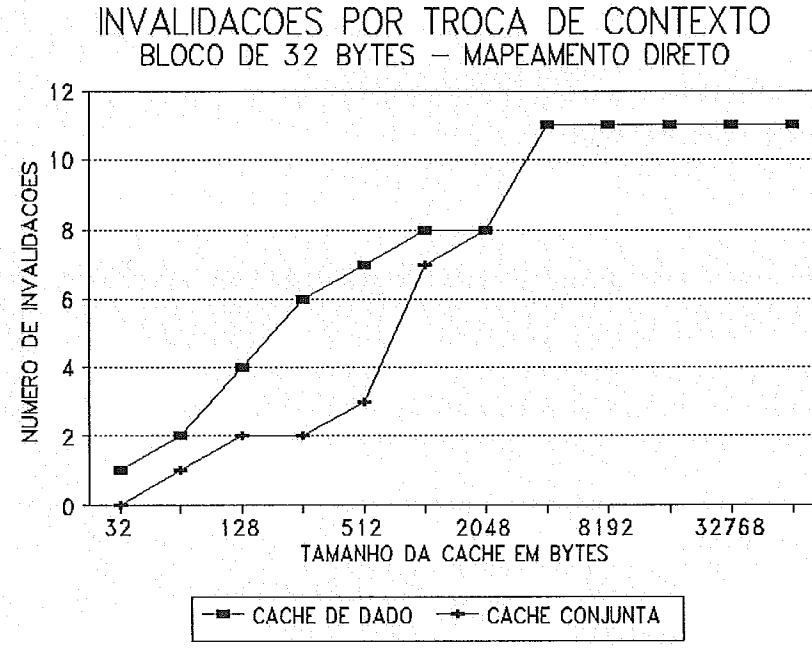


figura 5.6.b: tamanho da *cache* X número de blocos sujos

V.3 - Análise dos Resultados

Nesta seção são apresentados os resultados obtidos através da simulação de quatro sistemas, com diversas alternativas de *cache*, a saber:

- sistema sem *cache* e dois barramentos externos,
- sistema com *cache* utilizando *write through* e dois barramentos externos,
- sistema com *cache* utilizando *write back* e um barramento externo e
- sistema com *cache* utilizando *write back* e dois barramentos externos.

V.3.1 - Intervalo de Tolerância

Devido à pluraridade dos resultados, o cálculo da convergência dos resultados é de custo muito alto, sendo mais razoável a pré-fixação do tempo total de simulação em ciclos de relógio.

O custo de simulação, em relação ao tempo real, é de 11ms por NP, por ciclo de relógio, quando o simulador é executado em um computador compatível com IBM-PC com processador Intel 8088 com relógio de 4.77MHz.

O gráfico da figura 5.7 representa o pior caso em relação ao desvio padrão, que é o resultado da simulação de um sistema com apenas 1 NP no total.

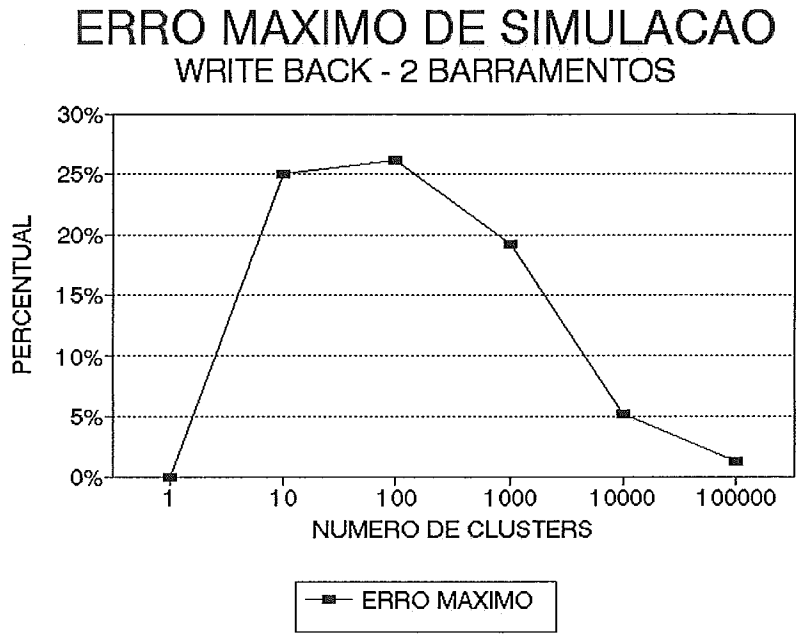


figura 5.7: desvio padrão dos resultados do Simulador

Sem considerar o resultado obtido por apenas um ciclo de simulação, o desvio padrão de uma amostra de simulação, para até 1.000 ciclos de relógio, está próximo de 30%, caindo rapidamente para apenas 5% com 10.000 ciclos de relógio. Com 100.000 ciclos, o erro máximo é de 2%, o que não representa um ganho muito significativo em relação ao aumento do tempo total de simulação, principalmente porque não deve ser esquecido que este é o pior caso. Portanto, o tempo de simulação foi fixado em 10.000 ciclos de relógio.

A cada vez que o número de NP's aumenta, o desvio padrão da amostra diminui proporcionalmente a $\sqrt{1/n}$ [WONN81] conforme equação abaixo:

$$s = \sqrt{\frac{n \sum x^2 - \left(\sum x \right)^2}{n\,(n-1)}}$$

A este erro deve ser somado o erro ocasionado pela função que gera números aleatórios. Esta função não possui capacidade de gerar uma distribuição totalmente aleatória, uma vez que esta é gerada em computador a partir de combinações de funções. A não linearidade da distribuição dos

valores gerados por esta função pode ser observada no gráfico da figura 5.8.

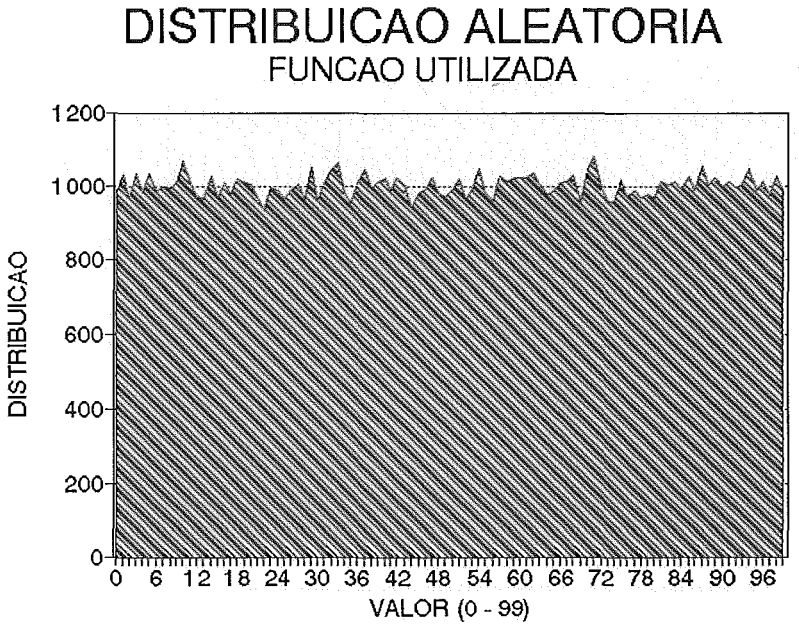


figura 5.8: distribuição da função aleatória

V.3.2 - Desempenho

■ **Throughput nos Barramentos/Taxa de Ocupação dos Barramentos/Tempo de Espera pelos Barramentos** - a baixa taxa de ocupação do barramento de dados para o sistema sem *cache* (figura 5.9) provavelmente é devida ao congestionamento do barramento de código, que é 4 vezes mais requisitado do que o barramento de dados, e também ao fato dos acessos internos serem muito demorados para o processador, diminuindo a relação entre acessos internos e externos, o que faz com que o processador demore mais tempo para requisitar o barramento externo.

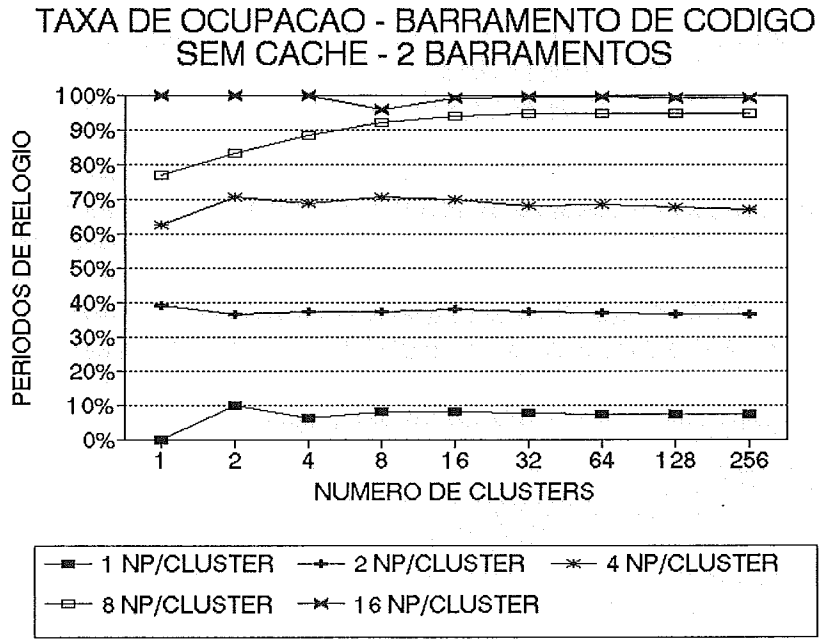


figura 5.9.a: taxa de ocupação do barramento de código

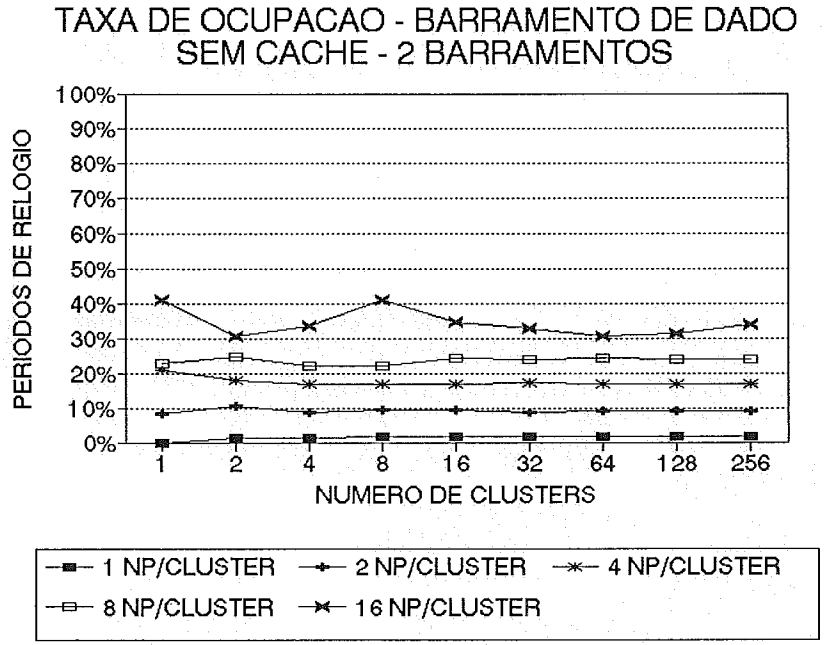


figura 5.9.b: taxa de ocupação do barramento de dado

Pode-se também observar que o aumento do número de processadores devido ao aumento do número de *clusters*, com a relação NP/*cluster* constante, pouco altera as taxas de ocupação dos barramentos (figuras 5.10 e 5.11), o

que é uma das grandes vantagens características das redes de interconexão.

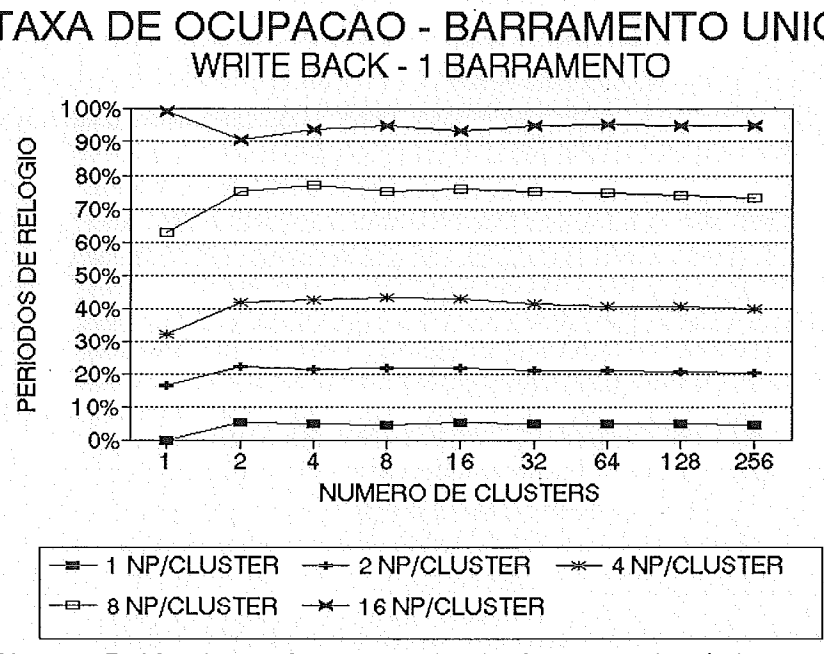


figura 5.10: taxa de ocupação do barramento único

Para qualquer sistema simulado, o uso do barramento externo é nulo para a configuração de 1 NP/*cluster* e 1 *cluster*, uma vez que não existe nada conectado ao barramento, com exceção da interface de E/S que não foi simulada. A taxa de ocupação para o barramento de código é idêntica nos sistemas com *cache* utilizando-se *write-back* ou *write-through*.

Pode-se observar, comparando os gráficos de taxa de ocupação de barramento de dados (figuras 5.11.b e 5.11.d) que o método de *write-back* reduz, ainda que de forma não muito significativa, o tráfego no barramento em sistemas com pouco compartilhamento, ou seja, com poucos NP's para, depois, igualar-se ao *write-through* com o aumento do número de NP's.

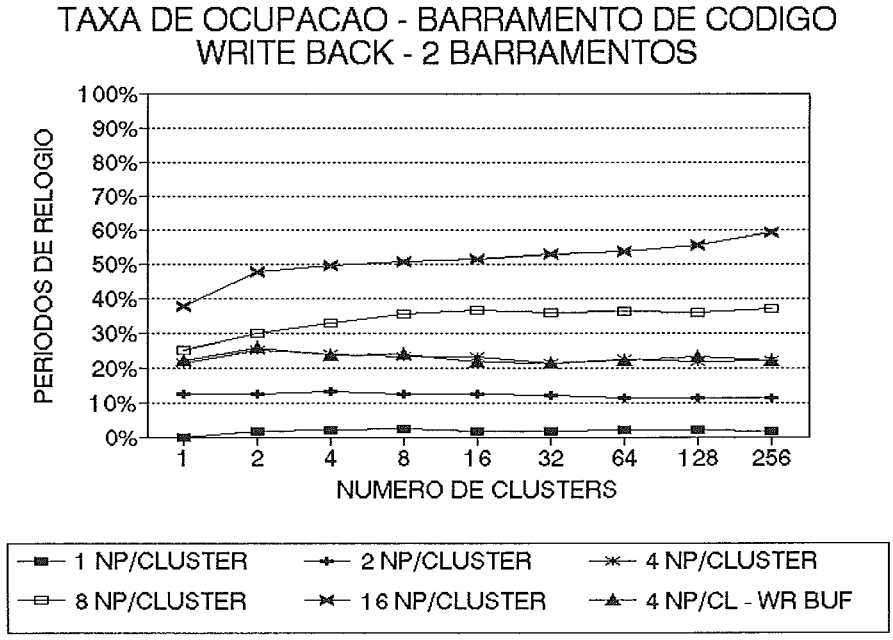


figura 5.11.a: taxa de ocupação do barramento de código

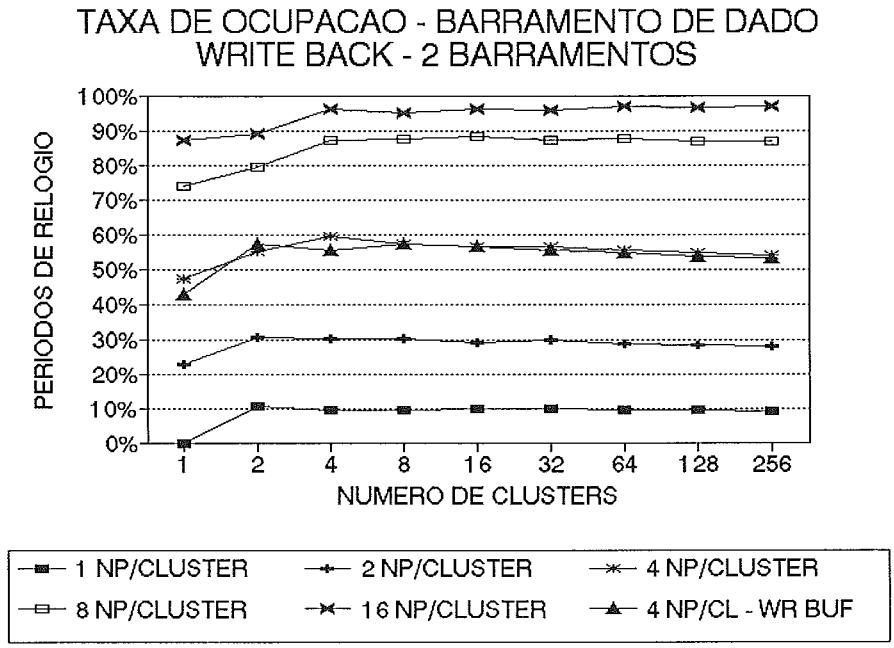


figura 5.11.b: taxa de ocupação do barramento de dado

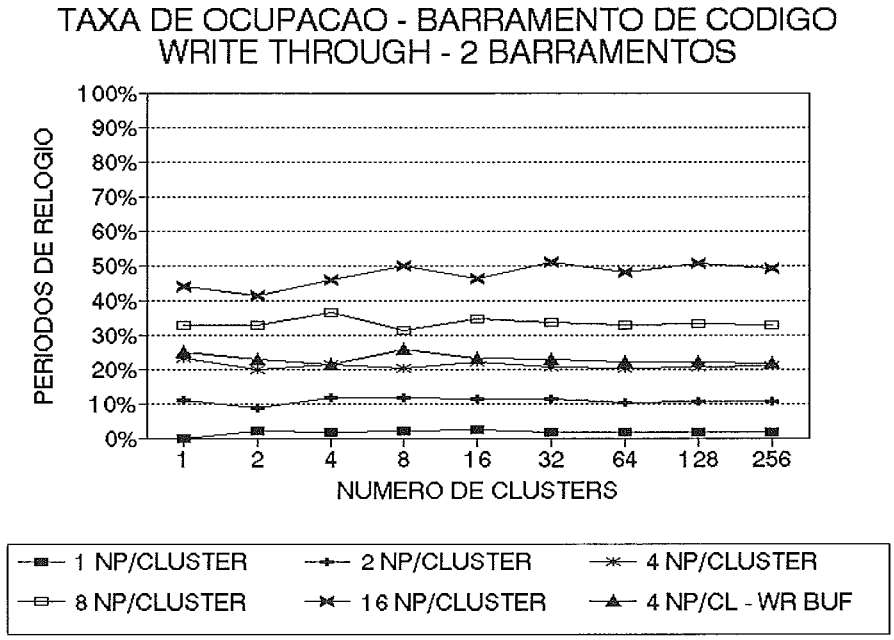


figura 5.11.c: taxa de ocupação do barramento de código

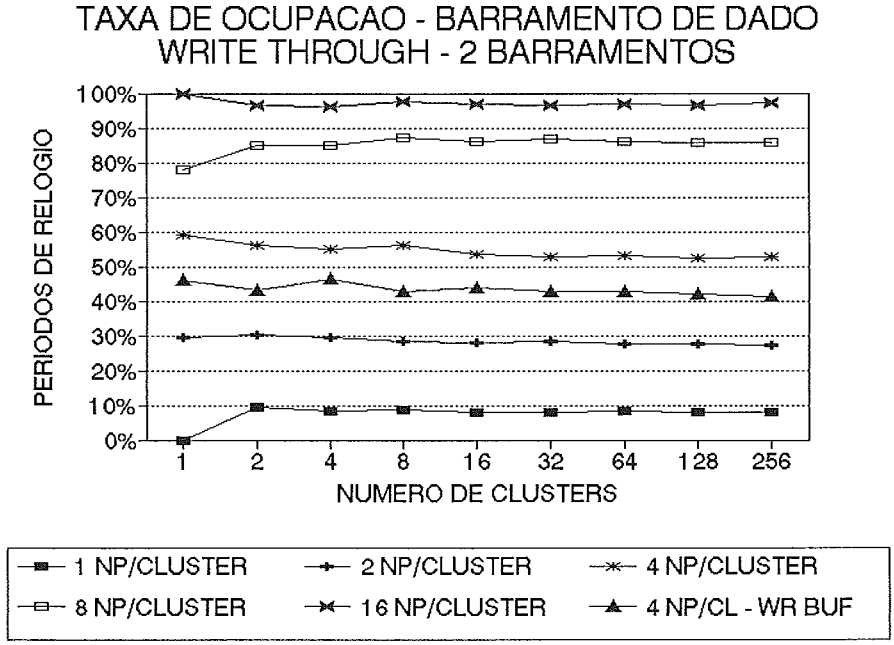


figura 5.11.d: taxa de ocupação do barramento de dado

■ **Total de Ciclos Executados/Duração Média dos Ciclos** - Através do número total de ciclos executados (figura 5.12) pode-se notar que, para até 16 NP's por *cluster*, o desempenho do sistema acompanha o número de NP's. Principalmente, observa-se que existe pouca degradação com o aumento de

clusters, o que se traduz graficamente pelas linhas quase horizontais nos gráficos. Em compensação, a duração média dos ciclos aumenta muito com o aumento do número de NP's.

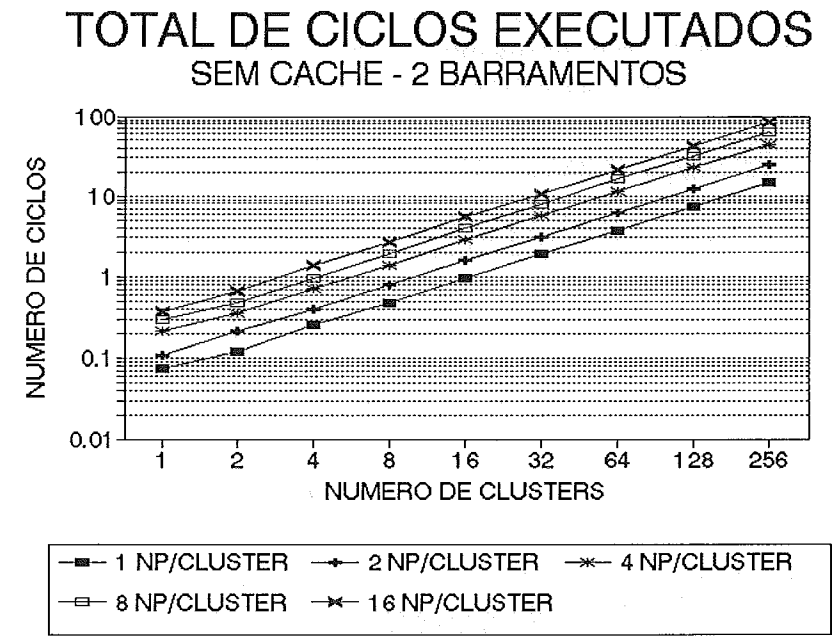


figura 5.12.a: total de ciclos executados

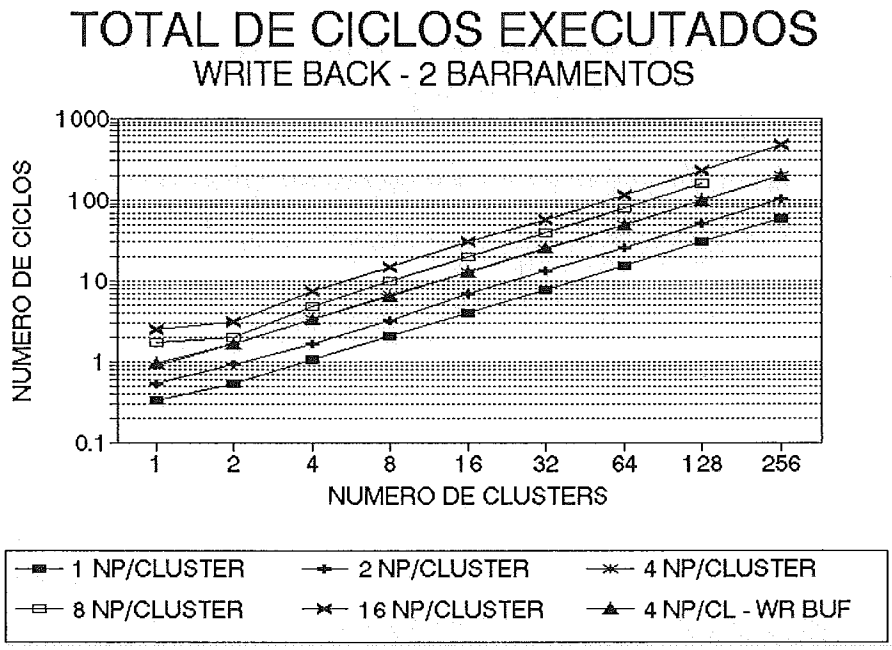


figura 5.12.b: total de ciclos executados

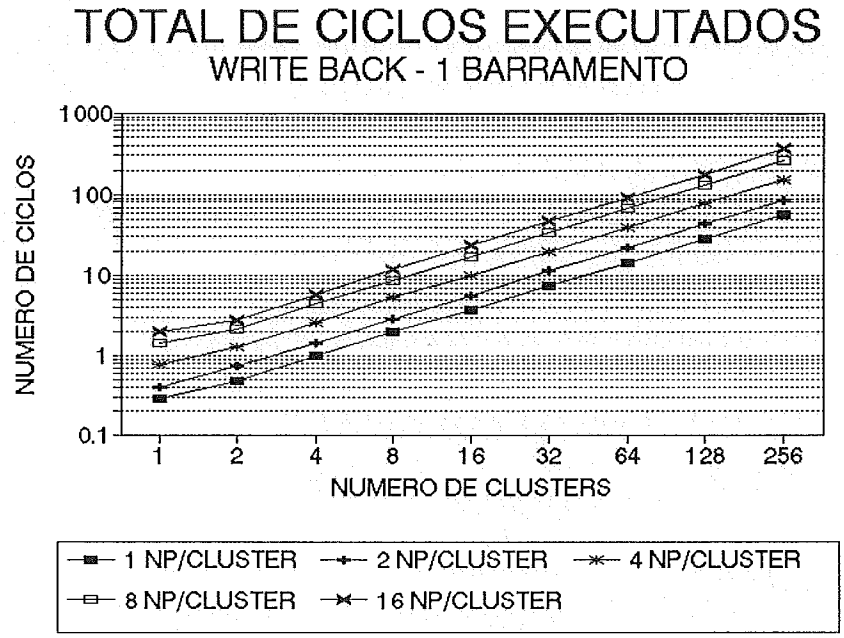


figura 5.12.c: total de ciclos executados

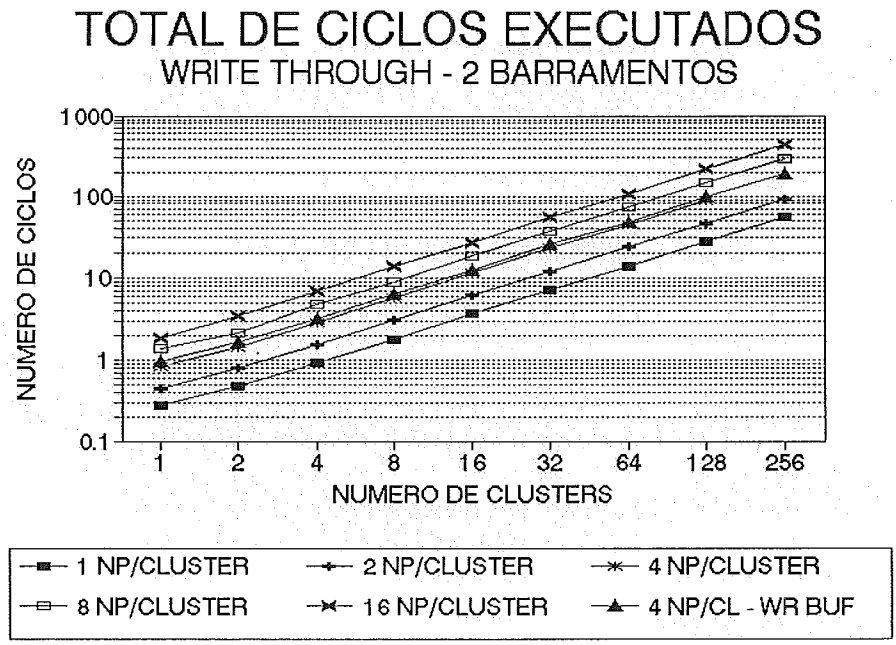


figura 5.12.d: total de ciclos executados

Através da análise dos gráficos de duração média dos ciclos (figura 5.13), pode-se constatar a grande superioridade, em termos de desempenho, dos sistemas com *cache* sobre os sem *cache*. O método *write back* apresenta um desempenho cerca de 20% superior ao método *write through* para

poucos NP's. Esta taxa cai, no entanto, para 4% quando muitos NP's são utilizados.

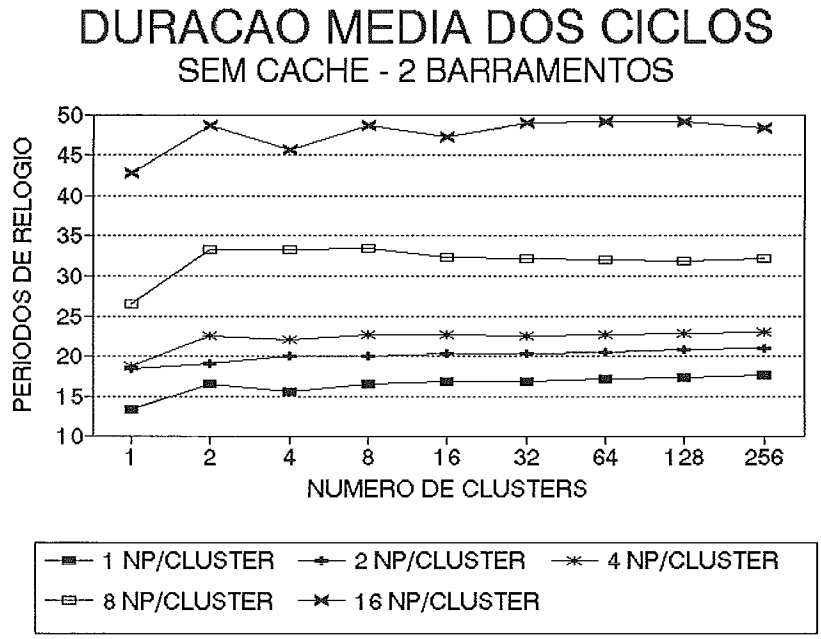


figura 5.13.a: duração média dos ciclos

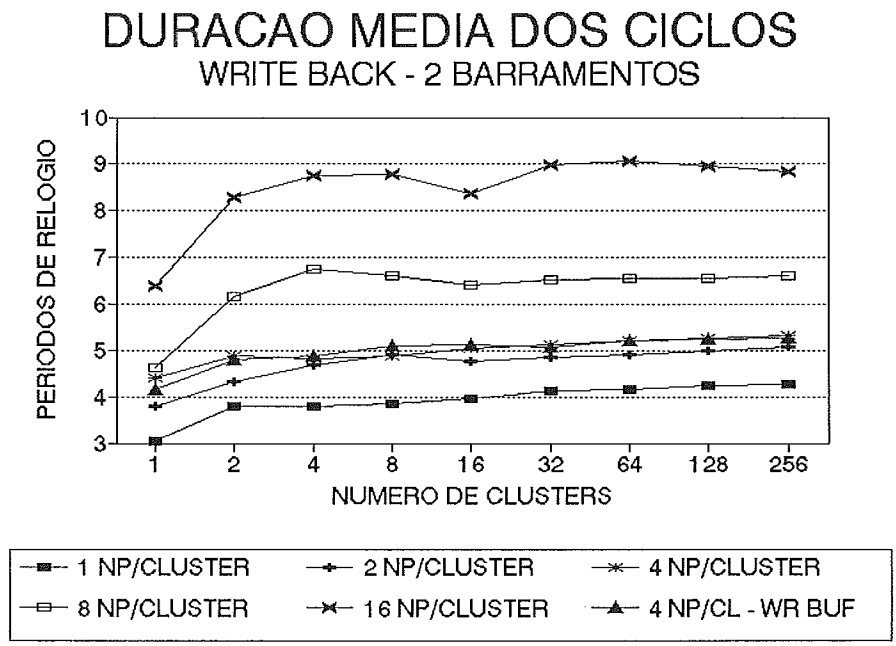


figura 5.13.b: duração média dos ciclos

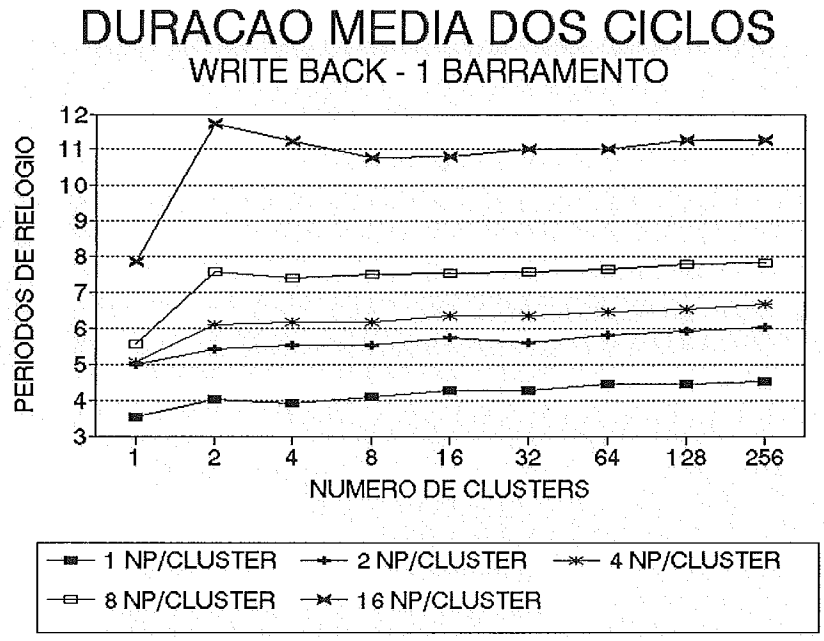


figura 5.13.c: duração média dos ciclos

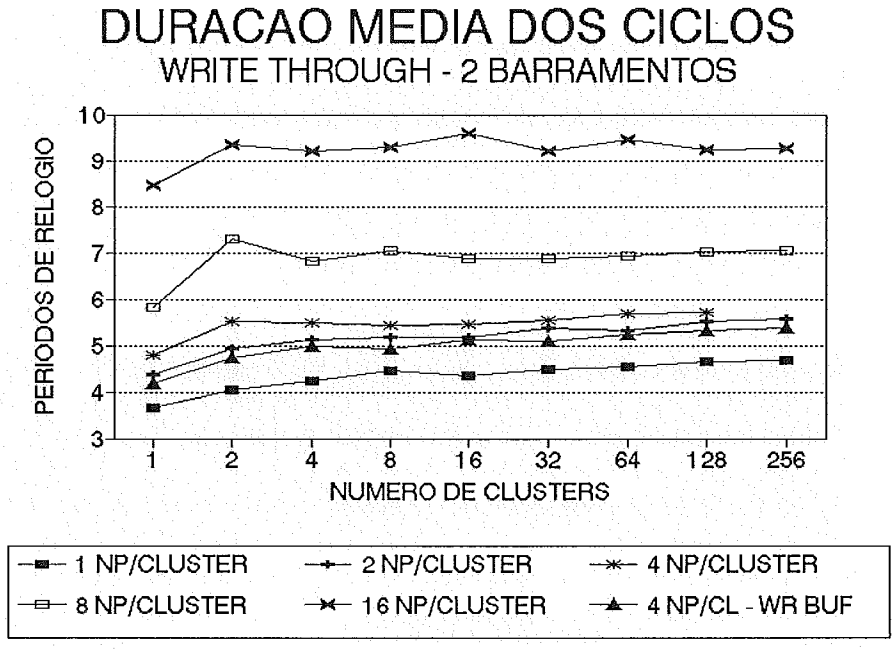


figura 5.13.d: duração média dos ciclos

A inclusão de *write buffer* nos sistemas que utilizam *write through* melhorou significativamente (de 20% a 25%) o desempenho do sistema. Já nos sistemas que utilizam a política de *write back* não houve modificação observável. O principal fator que contribuiu para a pouca melhora está no fato de que a maior parte das escritas realizadas são de blocos inteiros da

cache (32 *bytes* ou 4 acessos) e o *write buffer* implementado somente possui capacidade para armazenar um acesso por vez.

A figura 5.14 mostra a superioridade do sistema de dois barramentos sobre o sistema de um barramento quando o número de NP's por *cluster* é maior ou igual a 8.

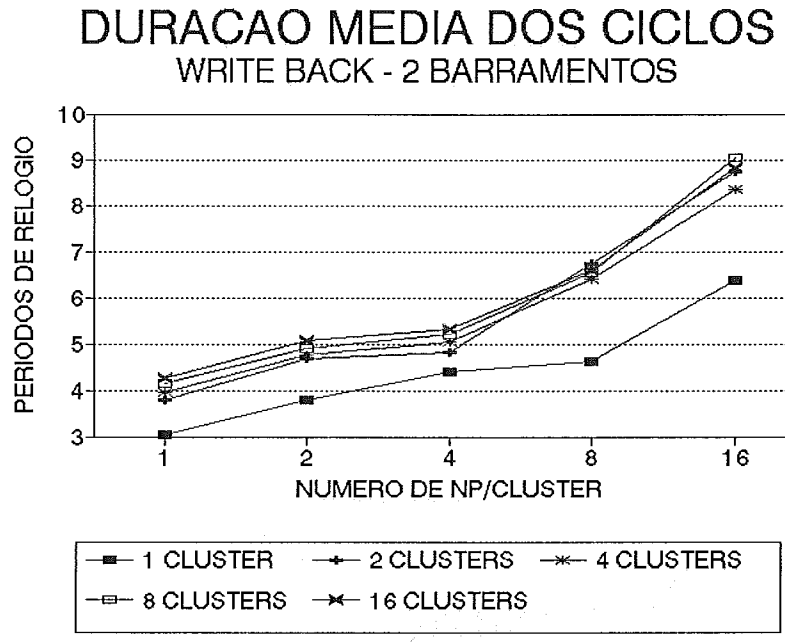


figura 5.14.a: duração média dos ciclos

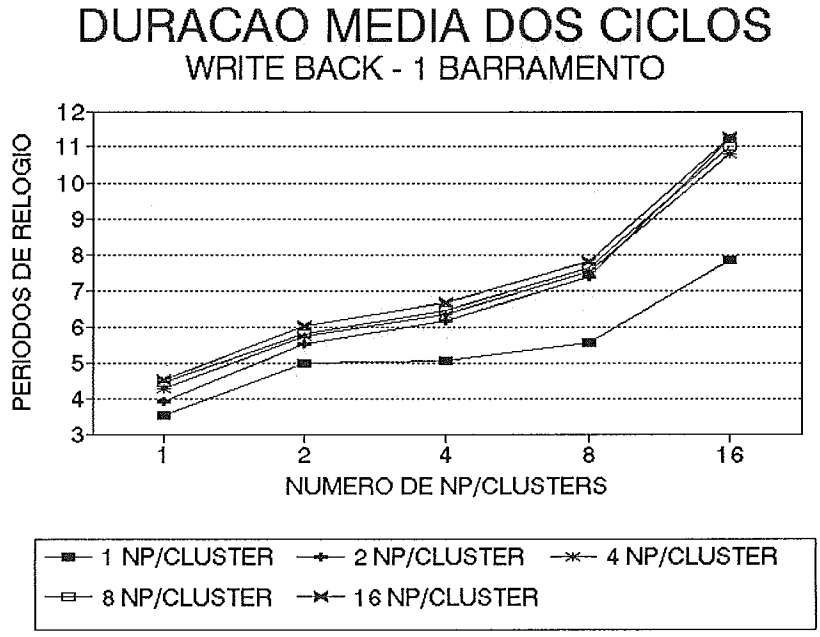


figura 5.14.b: duração média dos ciclos

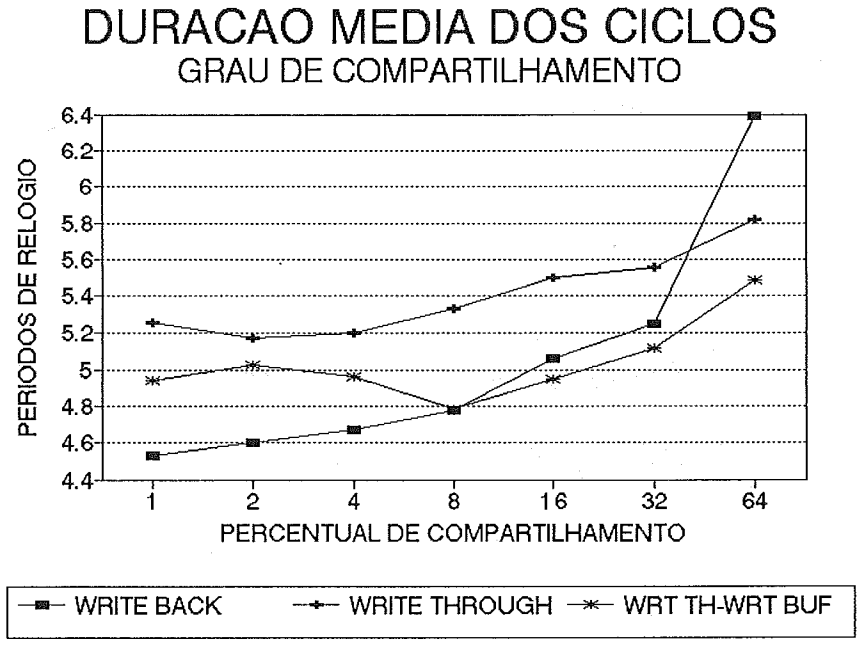


figura 5.15: duração média dos ciclos X compartilhamento

A figura 5.15 representa a simulação de uma configuração do MULTIPLUS com 4 *clusters* e 4 NP's por *cluster*. Observa-se que para valores pequenos para percentual de compartilhamento de dados, o esquema de *write back* é bastante superior ao esquema de *write through*. No entanto, quando

este percentual atinge a faixa de 10%, o esquema de *write through* com *write buffer* passa a apresentar um desempenho superior. Constata-se ainda que o comportamento do sistema com política de *write through* é basicamente invariante com o percentual de compartilhamento. Já o desempenho do sistema utilizando *write back* cai rapidamente com o percentual de compartilhamento.

CAPÍTULO VI

CONCLUSÕES E PERSPECTIVAS FUTURAS

Não existe uma arquitetura de memórias *cache* ideal para uso em todos os sistemas de computadores. Cada sistema necessita um estudo de seu perfil para que uma arquitetura específica de memórias *cache* possa ser definida.

Notadamente, em sistemas multiprocessados, é preferível, em termos de desempenho, a utilização de métodos para manutenção de coerência que minimizem a taxa de utilização do barramento comum aos processadores e memória global. Métodos como o de *write-back* ou o de *write-once* o fazem bem, sendo muito utilizados.

Caso exista alguma limitação no protocolo de barramento utilizado no sistema que inviabilize a utilização de métodos mais elaborados, pode-se optar pelo método de *write-through*, suportado pela grande maioria dos protocolos de barramentos.

Uma comparação entre arquiteturas de *caches* virtuais e físicos mostra que ambos possuem desempenho similar, com a ressalva de que a opção por uma arquitetura de *cache* físico permite que este seja transparente ao *software*. O item decisivo na escolha entre estas duas arquiteturas é a implementação do controlador, que para *cache* virtual é naturalmente mais simples, a não ser que a transparência seja mandatória.

Com relação ao grau de associatividade, o ideal seria um sistema de memória *cache* com mapeamento totalmente associativo. Esta *cache* totalmente associativa é de implementação complexa, e, felizmente, estudos mostram que uma *cache* com quatro conjuntos associativos tem um desempenho comparável com a totalmente associativa [KAPL73]. Verifica-se também que o próprio algoritmo de escolha do bloco a ser substituído não chega a modificar de maneira significativa o desempenho da *cache*.

A título de resumo conclusivo, são enumeradas a seguir as principais características do sistema de memórias *cache* utilizado no MULTIPLUS:

1) O Nó de Processamento do MULTIPLUS terá *cache* de dados e instruções separados, cada um possuindo 64 K*bytes*. Esta capacidade pode ser expandida até 256 K*bytes*.

2) É esperado que a taxa de acerto na *cache* de instrução seja maior do que a de dados, como ocorre em qualquer sistema.

3) Apesar da grande diferença no volume de acesso entre código e dado para máquinas RISC, a diferença entre as taxas de acerto da *cache* de instruções sobre a de dados leva o barramento de instruções a uma taxa de ocupação muito menor do que a do barramento de dados em sistemas utilizando políticas de *cache* do tipo *write through* ou *copy back*. Aproveitando este fato, as transferências de blocos em geral, com exceção de leituras e escritas do controlador de *cache* serão realizadas pelo barramento de intruções, para desafogar o barramento de dados.

4) A coerência dos blocos armazenados na *cache* de intrução será garantida por *software* através de operações de *flush* da *cache* de instruções sempre que uma página de código for liberada ou substituída.

5) A coerência dos dados na *cache* será assegurada por dois mecanismos independentes:
- por *hardware*: o *snoop* do controlador de *cache* monitora todas as ações no barramento de dados de seu *cluster*.
- por *software*: o sistema operacional não permitirá que dados de leitura e escrita compartilhados, localizados em outros *clusters*, sejam armazenados em *cache*. E, para garantir a manutenção da coerência durante E/S, o sistema operacional deverá realizar uma operação de *flush* na *cache* de dados, uma vez que a transferência dos blocos se dará pelo barramento de instruções onde o *snoop* do controlador de *cache* de dados não tem acesso.

6) A política de atualização da memória principal do tipo *write back* mostrou que, em termos de desempenho do sistema, é superior à de *write through*. Entretanto, pôde-se notar que com o aumento do grau de compartilhamento de variáveis, esta diferença no desempenho diminui, em parte devido ao grande número de invalidações de *cache* seguidas de *copy back* para a memória principal.

7) Com a inclusão de um *buffer* de escrita (*write buffer*), possibilitando que escritas na memória principal sejam realizadas, em alguns casos, em apenas um ciclo de relógio, o sistema utilizando a política do tipo *write through* apresentou desempenho equivalente ao sistema baseado na política de *write back*, com a vantagem de ser muito mais simples.

8) A variação da duração média dos ciclos sofre pouca influência com o aumento do número de *clusters* no sistema. Este resultado era esperado uma vez que a rede de interconexão permite que diversos acessos entre pares de *clusters* se realizem em paralelo.

Como continuação deste trabalho de tese, pretende-se incluir as rajadas de E/S no estudo de simulação visando obter-se resultados mais próximos aos de uso normal do sistema. Espera-se ainda incorporar ao simulador um modelo funcional da rede de interconexão, de forma a poder tornar o simulador capaz de modelar contenções na rede.

BIBLIOGRAFIA

[AUDE90] JÚLIO SALEK AUDE e outros
"MULTIPLUS: UM MULTIPROCESSADOR DE ALTO DESEMPENHO"
Anais do 10º Congresso da SBC
pp: 93-105
22-27 de julho de 1990 - Vitória - ES

[AGAR88] ANANT AGARWAL e ANOOP GUPTA
"Memory Reference Characteristics of Multiprocessor Application under MARCH"
ACM SIGMETRICS
1988

[ARCH84] JAMES ARCHIBALD e JEAN-LOUP BAER
"An Economical Solution to the Cache Coherence Problem"
The 11th Annual International Symposium on Computer Architecture
5-7 de junho de 1984
ACM SIGARCH Computer Architecture News
Vol. 12 (3), : 355-362
Junho de 1984

[AUST87] AUSTEK MICROSYSTEMS
"A38152 Microcache™ for Intel 80386-based Microprocessor Systems"
External Reference Specification
1 de junho de 1987

[BBN85] "Butterfly Parallel Processor Overview"
BBN Laboratories Incorporated
versão 1, 17 págs.
13 de junho de 1985

[BRON90] GERSON BRONSTEIN, ADRIANO JOAQUIM DE OLIVEIRA CRUZ e OTTO CARLOS MUNIZ BANDEIRA DUARTE
"Análise do Desempenho de Redes de Interconexão para Máquinas Paralelas"
Anais do III Simpósio Brasileiro de Arquitetura de Computadores Processamento Paralelo
pp: 345-360
7 à 9 de novembro de 1990 - Rio de Janeiro - RJ

[CYPR89a] "CMOS BiCMOS DATA BOOK"
Cypress Semiconductor Corporation
1 de fevereiro de 1989

[CYPR89b] CYPRESS SEMICONDUCTOR CORPORATION
"Cache Memory Design"
Application Handbook
págs. 6-11 à 6-32
1 de agosto de 1989

[CYPR90a] "SPARC RISC USER'S GUIDE"
Cypress Semiconductor Corporation
Ross Technology Subsidiary
Second Edition - Fevereiro de 1990

[CYPR90b] "SPARC™ Mbus Interface Specification"
Cypress Semiconductor Corporation
29 de março de 1990, versão 1.1

[DUBO82] MICHEL DUBOIS e FAYÉ A. BRIGGS
"Effects of Cache Coherence in Multiprocessors"
IEEE Transactions on Computer
vol. C-31 (11), pp: 1083-1099
Novembro de 1982

[EGGE89] SUSAN J. EGGERS e RANDY H. KATZ
"The Effect of Sharing on the Cache and Bus Performance of Parallel Programs"
ACM SIGARCH Computer Architecture News
vol 17 (2), pp: 257-270
Abril de 1989 - New York
ASPLOS III - Proceedings Third International Conference on Architectural Support for Programming Languages and Operating Systems
Abril de 1989 - Boston, Massachusets

[FEIT90] RAUL QUEIROS FEITOSA
"O Problema de Coerência de Memórias Cache Privadas em Grandes Multiprocessadores para Aplicações Numéricas: Uma Nova Solução"
Anais do 10º congresso da SBC
pp: 138-156
22-27 de junho de 1990, Vitória - ES

[GOOD83] JAMES R. GOODMAN
"Using Cache Memory to Reduce Processor-Memory Traffic"
ACM Computer Architecture News
Vol. 17 (3), pp: 124-137
13-17 de junho de 1983

[GOTT83] ALLAN GOTTLIEB, RALPH GRISHMAN, CLYDE P. KRUSKAL, KEVIN P. McAULIFE, LARRY RUDOLPH e MARC SNIR
"The NYU Ultracomputer - Designing a MIMD Shared Memory Parallel Computer"
IEEE Transactions on Computers
Vol. C-32 (2), págs. 175-189
Fevereiro de 1983

[HENN86] JOHN L. HENNESSY e MARK A. HOROWITS
"An Overview of the MIPS-X-MP Project"
Computer Systems Laboratory
Stanford University - Stanford - CA 94305-2192
Technical Report No. 86-300
Abril de 1986

[IEEE87] "IEEE Standard Backplane Bus Specification for Multiprocessor Architectures: Futurebus"
ANSI/IEEE
Std 896.1-1987, Std 1101-1987
1987

[INTE87] "Microprocessor and Peripheral Handbook"
Intel Corporation
Vol I - Microprocessor
1987

[KAPL73] K. R. KAPLAN e R. O. WINDER
"Cache-based Computer Systems"
IEEE Computer
vol. 6, pp: 30-36
Março de 1973

[LEE69] F. F. LEE
"Study of 'Look-Aside' Memory"
IEEE Transactions on Computer
Vol. C18, pp: 1062-1064
Novembro de 1969

[LIPT68] J. S. LIPTAY
"Structural aspects of the System/360 Model 85-II - The cache"
IBM System Journal
Vol. 7 (1), págs. 15-21
1 de julho de 1968

[MANU87] TOM MANUEL
"How Sequent's New Model Outruns Most Mainframes"
Electronics
pp: 76-78
Maio de 1987

[MICR90] "MOS DATA BOOK"
Micron Technology, Inc
1990

[MOTO83] "MC68000 16 Bits Microprocessor Reference Manual"
Motorola Semiconductor Inc.
julho de 1983

[MOTO88] "MC88200 Cache/Memory Management Unit User's Manual"
Motorola Inc.
Primeira edição
8 de dezembro de 1988

[NAMJ88] M. NAMJOO, F. ABU-NOFAL, D. CARMEAN, R. CHANDRAMOULI, Y. CHANG, J.GOFORTH, W. HSU, R. IWAMOTO, C. MURPHY, U. NAOT, M. PARKIN, J. PEDDETON, C. PORTER, J. RAVES, R. REDDY, G. SWAN, D. TINKER, P. TANG e L. YANG
"CMOS Custom Implementation of the SPARC Architecture"
33º IEEE Computer Society International Conference

pp: 18-20
29 de fevereiro à 4 de março de 1988 - California

[PEIX90] GUSTAVO PEIXOTO DE AZEVEDO, RAFAEL PEIXOTO DE AZEVEDO, NORIVAL RIBEIRO FIGUEIRA e JÚLIO SALEK AUDE
"MULPLIX: Um Sistema Operacional Tipo UNIX para o Multiprocessador MULTIPLUS"
Anais do III Simpósio Brasileiro de Arquitetura de Computadores Processamento Paralelo
pp: 122-137
7 à 9 de novembro de 1990 - Rio de Janeiro - RJ

[PFIS85] G. P. PFISTER
"The Arquitecture of the IBM Research Parallel Processor Prototype (RP3)"
IBM T. J. Watson Research Laboratory
Research Report - Relatório Técnico, 18 págs.
24 de junho de 1985 - Yorktown Heights, NY

[PFIS86] G. P. PFISTER, W. C. BRANTLEY, D. A. GEORGE, S. L. HARVEY, J. KLEINFELDER, K. P. McAULIFFE, E. A. MELTON, V. A. NORTON e J. WEISS
"An Introduction to the IBM Research Parallel Processor Prototype (RP3)"
IBM T. J. Watson Research Center
Research Report - Relatório Técnico, 32 págs.
Yorktown Heights, NY
13 de junho de 1986

[RUDO85] LARRY RUDOLPH e ZARRY SEGALL
"Dinamic Decentralized Cache Schemes for MIMD Parallel Processors"
Proceedings of the 12th International Symposium on Computer Architecture - ACM SIGARCH Newsletter
vol. 3 (3), pp: 340-347
1985

[SEIT85] CHARLES L. SEITZ
"The Cosmic Cube"
Communicaction of the ACM
Vol 28 (1), págs. 22-33
Janeiro de 1985

[SMIT82] ALAN JAY SMITH
"Cache Memories"
Computing Surveys
vol 14 (3), pp: 473-530
setembro de 1982

[SMIT83] JAMES E. SMITH e JAMES R. GOODMAN
"A STUDY OF CACHE ORGANIZATION AND REPLACEMENT POLICIES"
ACM Computer Architercture News
Vol 17 (3), pp: 132-137
13-17 de junho de 1983

[SMIT87] ALAN JAY SMITH
"Line (Block) Size Choice for CPU Cache Memories"
IEEE Transactions on Computer
vol C-36 (9), pp: 1063-1075
setembro de 1987

[STEN89] PER STENTRÖM
"A Cache Consistency Protocol for Multiprocessors with Multistage Networks"
ACM Computer Architecture News
Vol. 17 (3), pp: 407-415
Junho de 1989

[STON87] HAROLD S. STONE
"High-Performance Computer Architecture"
Addison-Wesley Publishing Company
1987

[TAMI81] YUVAL TAMIR
"Simulation and Performance Evaluation of the RISC Architecture"
University of California
College of Engineering and Computer Sciences
Computer Science Division
março de 1981

[TANG76] C. K. TANG
"Cache system design in the tightly coupled multiprocessor system"
AFIPS Conference Proceedings, National Computer Conference
New York, NY, vol 45, págs. 749-753
Junho de 1976

[TEST86] A. J. TEST
"Multi-processor Management in the Concentrix Operating System"
Usenix Conference Proceedings
Denver, Colorado, pp: 173-182
Inverno de 1986

[WEBE89] W-D WEBER e A. CUP
"Analysis of Cache Invalidation Patterns in Multiprocessors"
ACM SIGARCH Computer Architectural News
New York, vol. 17 (2), pp:243-256
Abril de 1989
ASPLOS III Proceedings Third International Conference on Architectural Support for Programming Languages and Operating Systems
Boston
Abril de 1989

[WILK65] M. V. WILKES
"Slave Memories and Dynamic Storage Allocation"
IEEE Transactions on Electronic Computers
Vol EC-14, pp: 270-271

Abril de 1965

[WONN81] THOMAS H. WONNACOTT e RONALD J. WONNACOTT
Tradução de Alfredo Alves de Farias
"Estatística Aplicada à Economia e à Administração"
Livros Técnicos e Científicos Editora S/A
1ª edição - Rio de Janeiro - RJ - 685 páginas
1981

Apêndice A - Árvore de acessos

EXEMPLO DE PERCENTUAL DE TIPOS DE ACESSOS

Política de *Write-Back*

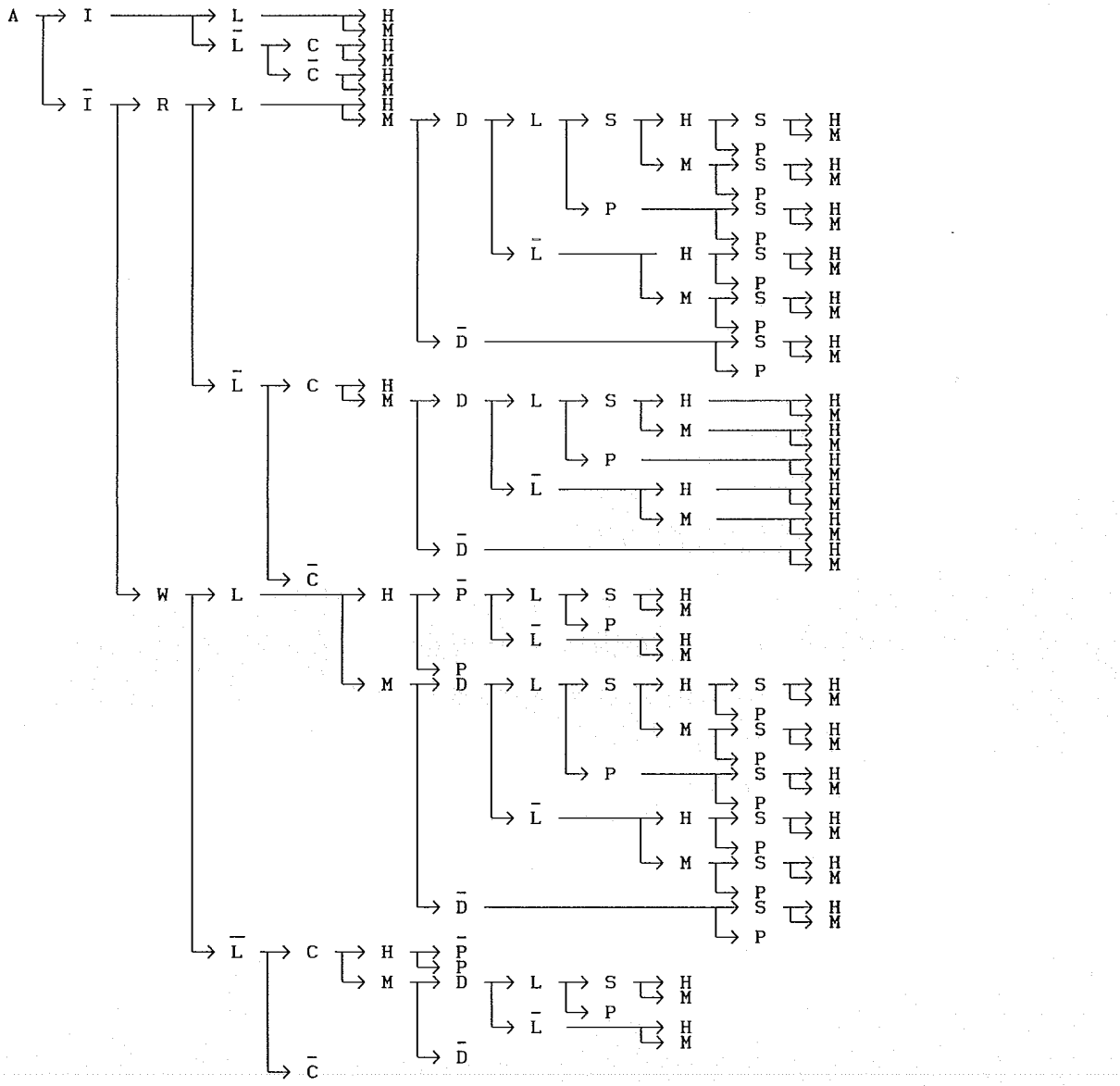
legenda:

- A - acesso em geral
- B - acesso local ao barramento do cluster
- $\overline{B}$ - acesso através das chaves
- I - acesso de código
- $\overline{I}$ - acesso de dado
- H - acesso com *hit* na *cache*
- $\overline{H}$ - acesso com *miss* na *cache*
- L - acesso local à placa
- $\overline{L}$ - acesso ao barramento ou através das chaves
- R - acesso de leitura de dado
- W - acesso de escrita de dado
- C - bloco no estado *clean*
- $\overline{C}$ - bloco no estado *dirty*
- S - bloco no estado *shared*
- P - bloco no estado *private*

Suposições:

instruções	= 80% dos acessos
dados	= 20% dos acessos
escritas de dados	= 35% dos acessos de dados
leituras de dados	= 65% dos acessos de dados
acerto na *cache* de código	= 95% dos acessos de instruções
acerto na *cache* de dado	= 80% dos acessos de dados
acessos locais ao NP	= 80% dos acessos
acessos locais ao *cluster*	= 80% dos acessos

Acesso
Instrução/Dado
Read/Write
Local/Remoto
Cluster/Rede
Hit/Miss
Dirty/Clear - Shared/Private
Local/Remoto (old-cache)
Shared/Private (old-memória)
Hit/Miss (snoop)
Shared/Private (memória)
Hit/Miss (snoop)



Apêndice B - Estados dos Recursos e Processadores

FREE livre

ESTADOS W

WAIT esperando (*Wait*) recurso

WCLH esperando (*Wait*) busca de Código na memória Local com acerto (*Hit*) na cache de código

WCLM esperando (*Wait*) busca de Código na memória Local com falha (*Miss*) na cache de código

WCRCH esperando (*Wait*) busca de Código Remoto no Cluster com acerto (*Hit*) na cache de código

WCRCM esperando (*Wait*) busca de Código Remoto no Cluster com falha (*Miss*) na cache de código

WCRRH esperando (*Wait*) busca de Código Remoto na Rede com acerto (*Hit*) na cache de código

WCRRM esperando (*Wait*) busca de Código Remoto na Rede com falha (*Miss*) na cache de Código

WRLH esperando (*Wait*) leitura (*Read*) de dado na memória Local com acerto (*Hit*) na cache de dado

WRLMDP esperando (*Wait*) leitura (*Read*) de dado na memória Local com falha (*Miss*) na cache de dado e bloco antigo sujo (*Dirty*), Privado

WRLMDS esperando (*Wait*) leitura (*Read*) de dado na memória Local com falha (*Miss*) na cache de dado e bloco antigo sujo (*Dirty*), compartilhado (*Shared*)

WRLMCP esperando (*Wait*) leitura (*Read*) de dado na memória Local com falha (*Miss*) na cache de dado e bloco antigo clear, Privado

WRLMCS esperando (*Wait*) leitura (*Read*) de dado na memória Local com falha (*Miss*) na cache de dado e bloco antigo clear, compartilhado (*Shared*)

WRRCH esperando (*Wait*) leitura (*Read*) de dado Remoto no Cluster com acerto (*Hit*) na cache de dado

WRRCMD esperando (*Wait*) leitura (*Read*) de dado Remoto no Cluster com falha (*Miss*) na cache de dado e bloco antigo sujo (*Dirty*)

WRRCMC esperando (*Wait*) leitura (*Read*) de dado Remoto no Cluster com falha (*Miss*) na cache de dado e bloco antigo limpo (*Clean*)

WRRR esperando (*Wait*) leitura (*Read*) de dado Remoto na Rede

WWLHP esperando (*Wait*) escrita (*Write*) de dado na memória Local com acerto (*Hit*) na cache de dado, Privado

WWLHS esperando (*Wait*) escrita (*Write*) de dado na memória Local com acerto (*Hit*) na cache de dado e dado compartilhado (*Shared*)

WWLMDP esperando (*Wait*) escrita (*Write*) de dado na memória Local com falha (*Miss*) na cache de dado e dado antigo sujo (*Dirty*), Privado

WWLMDS esperando (*Wait*) escrita (*Write*) de dado na memória Local com falha (*Miss*) na cache de dado e dado antigo sujo (*Dirty*), compartilhado (*Shared*)

WWLMCP esperando (*Wait*) escrita (*Write*) de dado na memória Local com falha (*Miss*) na cache de dado e dado antigo limpo (*Clean*), Privado

WWLMCS esperando (*Wait*) escrita (*Write*) de dado na memória Local com falha (*Miss*) na cache de dado e dado antigo limpo (*Clean*), compartilhado (*Shared*)

WWRCHP esperando (*Wait*) escrita (*Write*) de dado Remoto no Cluster com acerto (*Hit*) na cache de dado, Privado

WWRCHS esperando (*Wait*) escrita (*Write*) de dado Remoto no Cluster com acerto (*Hit*) na cache de dado, compartilhado (*Shared*)

WWRCMD esperando (*Wait*) escrita (*Write*) de dado Remoto no Cluster com falha (*Miss*) na cache de dado e Bloco anterior sujo (*Dirty*)

WWRCMC esperando (*Wait*) escrita (*Write*) de dado Remoto no Cluster com falha (*Miss*) na cache de dado e bloco anterior limpo (*Clean*)

WWRR esperando (*Wait*) escrita (*Write*) de dado Remoto na Rede, Privado

ESTADOS X

BUSY ocupado

XCLH executando busca de Código na memória Local com acerto (*Hit*) na cache de código

XCLM executando busca de Código na memória Local com falha (*Miss*) na cache de código

XCRCH executando busca de Código Remoto no Cluster com acerto (*Hit*) na cache de código

XCRCM executando busca de Código Remoto no Cluster com falha (*Miss*) na cache de código

XCRRH executando busca de Código Remoto na Rede com acerto (*Hit*) na cache de código

XCRRM executando busca de Código Remoto na Rede com falha (*Miss*) na cache de Código

XRLH executando leitura (*Read*) de dado na memória Local com acerto (*Hit*) na cache de dado

XRLMDP executando leitura (*Read*) de dado na memória Local com falha (*Miss*) na cache de dado e bloco antigo sujo (*Dirty*), Privado

XRLMDS executando leitura (*Read*) de dado na memória Local com falha (*Miss*) na cache de dado e bloco antigo sujo (*Dirty*), compartilhado (*Shared*)

XRLMCP executando leitura (*Read*) de dado na memória Local com falha (*Miss*) na cache de dado e bloco antigo clear, Privado

XRLMCS executando leitura (*Read*) de dado na memória Local com falha (*Miss*) na cache de dado e bloco antigo clear, compartilhado (*Shared*)

XRRCH executando leitura (*Read*) de dado Remoto no Cluster com acerto (*Hit*) na cache de dado

XRRCMD executando leitura (*Read*) de dado Remoto no Cluster com falha (*Miss*) na cache de dado e bloco antigo sujo (*Dirty*)

XRRCMC executando leitura (*Read*) de dado Remoto no Cluster com falha (*Miss*) na cache de dado e bloco antigo limpo (*Clean*)

XRRR executando leitura (*Read*) de dado Remoto na Rede

XWLHP executando escrita (*Write*) de dado na memória Local com acerto (*Hit*) na cache de dado, Privado

XWLHS executando escrita (*Write*) de dado na memória Local com acerto (*Hit*) na cache de dado e dado compartilhado (*Shared*)

XWLMDP executando escrita (*Write*) de dado na memória Local com falha (*Miss*) na cache de dado e dado antigo sujo (*Dirty*), Privado

XWLMDS executando escrita (*Write*) de dado na memória Local com falha (*Miss*) na cache de dado e dado antigo sujo (*Dirty*), compartilhado (*Shared*)

XWLMCP executando escrita (*Write*) de dado na memória Local com falha (*Miss*) na cache de dado e dado antigo limpo (*Clean*), Privado

XWLMCS executando escrita (*Write*) de dado na memória Local com falha (*Miss*) na cache de dado e dado antigo limpo (*Clean*), compartilhado (*Shared*)

XWRCHP executando escrita (*Write*) de dado Remoto no Cluster com acerto (*Hit*) na cache de dado, Privado

XWRCHS executando escrita (*Write*) de dado Remoto no Cluster com acerto (*Hit*) na cache de dado, compartilhado (*Shared*)

XWRCMD executando escrita (*Write*) de dado Remoto no Cluster com falha

(*Miss*) na cache de dado e bloco anterior sujo (*Dirty*)

XWRCMC executando escrita (*Write*) de dado Remoto no Cluster com falha (*Miss*) na cache de dado e bloco anterior limpo (*Clean*)

XWRR executando escrita (*Write*) de dado Remoto na Rede, Privado

Apêndice C - Custos de Sincronização

Exemplos de casos de sequências de sincronização com instruções atômicas para política de *cache* do tipo *write-back*.

legenda:

CPU 0 - processador 0
CPU 1 - processador 1
CACHE 0 - *cache* do processador 0
CACHE 1 - *cache* do processador 1
BARRA - barramento comum (contém o número do processador que o está ocupando)
AÇÃO - ação que está ocorrendo no barramento
LOCK - início de instrução atômica
LE - ação de leitura
ESC - ação de escrita
(W) - processador no estado de espera (*wait*)
CICLOS - número de ciclos requeridos pela ação
N/N' - número variável de ciclos para aquisição do barramento

WRITE-BACK com 4 estados (modo não reflexivo):
READ MODIFY WRITE com escrita de outro processador
sem *cache* ■ (N + 10 + 5 + 1)+ (N' + 10 + 1) = 27 + N + N'

CPU 0	CPU1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		I	I	-	0	-
LOCK		I	I	0	1 + N	NOP
LE		I → P	I	0	10	LE
	ESC(W)	P	MISS	0	0	-
ESC	ESC(W)	P → M	I	0	0	-
UNLOCK	ESC(W)	M	I	0	1	-
	ESC	M → I	I → M	1	10	LE (CHE 0) e INVALIDA

CPU 0	CPU1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		I	P	-	0	-
LOCK		I	P	0	1 + N	NOP
LE		I → S	P → S	0	10	LE
	ESC(W)	S	S	0	0	-
ESC	ESC(W)	S → M	S → I	0	4	INVALIDA
UNLOCK	ESC(W)	M	I	0	1	-
	ESC	M → I	I → M	1	1+N'+4	LE (CHE 0) e INVALIDA

CPU 0	CPU1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		I	M	-	0	-
LOCK		I	M	0	1 + N	NOP
LE		I → S	M → S	0	10	LE
	ESC(W)	S	S	0	0	-
ESC	ESC(W)	S → M	S → I	0	4	INVALIDA
UNLOCK	ESC(W)	M	I	0	1	-
	ESC	M → I	I → M	1	1+N'+4	LE (CHE 0) e INVALIDA

CPU 0	CPU1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		I	S	-	0	-
LOCK		I	S	0	1 + N	NOP
LE		I → S	S	0	10	LE
	ESC(W)	S	S	0	0	-
ESC	ESC(W)	S → M	S → I	0	4	INVALIDA
UNLOCK	ESC(W)	M	I	0	1	-
	ESC	M → I	I → M	1	1+N'+4	LE (CHE 0) e INVALIDA

CPU 0	CPU1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		P	I	-	0	-
LOCK		P	I	0	1 + N	NOP
LE		P	I	0	1	-
	ESC(W)	P	MISS	0	0	-
ESC	ESC(W)	P → M	I	0	1	-
UNLOCK	ESC(W)	M	I	0	1	-
	ESC	M → I	I → M	1	1+N'+4	LE (CHE 0) e INVALIDA

CPU 0	CPU 1	CACHE 0	CACHE 1
		P	P

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		P	M

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		P	S

NÃO EXISTE ESTE CASO

CPU 0	CPU1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		M	I	-	0	-
LOCK		M	I	0	1 + N	NOP
LE		M	I	0	1	-
	ESC(W)	M	MISS	0	0	-
ESC	ESC(W)	M	I	0	1	-
UNLOCK	ESC(W)	M	I	0	1	-
	ESC	M → I	I → M	1	1+N'+4	LE (CHE 0) e INVALIDA

CPU 0	CPU 1	CACHE 0	CACHE 1
		M	P

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		M	M

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		M	S

NÃO EXISTE ESTE CASO

CPU 0	CPU1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		S	I	-	0	-
LOCK		S	I	0	1 + N	NOP
LE		S	I	0	1	-
	ESC(W)	S	MISS	0	0	-
ESC	ESC(W)	S → M	I	0	4	INVALIDA
UNLOCK	ESC(W)	M	I	0	1	-
	ESC	M → I	I → M	1	1+N'+4	LE (CHE 0) e INVALIDA

CPU 0	CPU 1	CACHE 0	CACHE 1
		S	P

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		S	M

NÃO EXISTE ESTE CASO

CPU 0	CPU1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		S	S	-	0	-
LOCK		S	S	0	1 + N	NOP
LE		S	S	0	1	-
	ESC(W)	S	S	0	0	-
ESC	ESC(W)	S → M	S → I	0	4	-
UNLOCK	ESC(W)	M	I	0	1	-
	ESC	M → I	I → M	1	1+N'+4	LE (CHE 0) E INVALIDA

Dois casos de READ MODIFY WRITE:

CPU 0	CPU 1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		I	I	-	0	-
LOCK		I	I	0	1 + N	NOP
LE	LOCK	I → P	I	0	10	LE
	(W)	P	I	0	0	-
ESC	(W)	P → M	I	0	1	-
UNLOCK	(W)	M	I	0	1	-
	LE	M → S	I → S	1	1+N'+4	LE (CHE 0)
	ESC	S → I	S → M	1	4	INVALIDA
	UNLOCK	I	M	1	1	-

CPU 0	CPU 1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		I	P	-	0	-
LOCK		I	P	0	1 + N	NOP
LE	LOCK	I → S	P → S	0	10	LE
	(W)	S	S	0	0	-
ESC	(W)	S → M	S → I	0	4	INVALIDA
UNLOCK	(W)	M	I	0	1	-
	LE	M → S	I → S	1	1+N'+4	LE (CHE 0)
	ESC	S → I	S → M	1	4	INVALIDA
	UNLOCK	I	M	1	1	-

CPU 0	CPU 1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		I	M	-	0	-
LOCK		I	M	0	1 + N	NOP
LE	LOCK	I → S	M → S	0	10	LE
	(W)	S	S	0	0	-
ESC	(W)	S → M	S → I	0	4	INVALIDA
UNLOCK	(W)	M	I	0	1	-
	LE	M → S	I → S	1	1+N'+4	LE (CHE 0)
	ESC	S → I	S → M	1	4	INVALIDA
	UNLOCK	I	M	1	1	-

CPU 0	CPU 1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		I	S	-	0	-
LOCK		I	S	0	1 + N	NOP
LE	LOCK	I → S	S	0	10	LE
	(W)	S	S	0	0	-
ESC	(W)	S → M	S → I	0	4	INVALIDA
UNLOCK	(W)	M	I	0	1	-
	LE	M → S	I → S	1	1+N'+4	LE (CHE 0)
	ESC	S → I	S → M	1	4	INVALIDA
	UNLOCK	I	M	1	1	-

CPU 0	CPU 1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		P	I	-	0	-
LOCK		P	I	0	1 + N	NOP
LE	LOCK	P	I	0	1	LE
	(W)	P	I	0	0	-
ESC	(W)	P → M	I	0	1	INVALIDA
UNLOCK	(W)	M	I	0	1	-
	LE	M → S	I → S	1	1+N'+4	LE (CHE 0)
	ESC	S → I	S → M	1	4	INVALIDA
	UNLOCK	I	M	1	1	-

CPU 0	CPU 1	CACHE 0	CACHE 1
		P	P

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		P	M

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		P	S

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		M	I	-	0	-
LOCK		M	I	0	1 + N	NOP
LE	LOCK	M	I	0	1	-
	(W)	M	I	0	0	-
ESC	(W)	M	I	0	1	-
UNLOCK	(W)	M	I	0	1	-
	LE	M → S	I → S	1	1+N'+4	LE (CHE 0)
	ESC	S → I	S → M	1	4	INVALIDA
	UNLOCK	I	M	1	1	-

CPU 0	CPU 1	CACHE 0	CACHE 1
		M	P

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		M	M

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		M	S

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		S	I	-	0	-
LOCK		S	I	0	1 + N	NOP
LE	LOCK	S	I	0	1	-
	(W)	S	I	0	0	-
ESC	(W)	S → M	I	0	1	INVALIDA
UNLOCK	(W)	M	I	0	1	-
	LE	M → S	I → S	1	1+N'+4	LE (CHE 0)
	ESC	S → I	S → M	1	4	INVALIDA
	UNLOCK	I	M	1	1	-

CPU 0	CPU 1	CACHE 0	CACHE 1
		S	P

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		S	M

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		S	S	-	0	-
LOCK		S	S	0	1 + N	NOP
LE	LOCK	S	S	0	1	-
	(W)	S	S	0	0	-
ESC	(W)	S → M	S → I	0	4	INVALIDA
UNLOCK	(W)	M	I	0	1	-
	LE	M → S	I → S	1	1+N'+4	LE (CHE 0)
	ESC	S → I	S → M	1	4	INVALIDA
	UNLOCK	I	M	1	1	-

WRITE-BACK com 5 estados (modo reflexivo):

READ MODIFY WRITE com escrita de outro processador

sem *cache* ≡ (N + 10 + 5 + 1) + (N' + 10 + 1) = 27 + N + N'

CPU 0	CPU1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		IN	IN	-	0	-
LOCK		IN	IN	0	1 + N	NOP
LE		IN → EC	IN	0	10	LE
	ESC(W)	EC	MISS	0	0	-
ESC	ESC(W)	EC → EM	IN	0	1	-
UNLOCK	ESC(W)	EM	IN	0	1	-
	ESC	EM → IN	IN → EM	1	1+4+N'	LE (CHE 0) e INV

CPU 0	CPU1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		IN	SC	-	0	-
LOCK		IN	SC	0	1 + N	NOP
LE		IN → SC	SC	0	10	LE
	ESC(W)	SC	SC	0	0	-
ESC	ESC(W)	SC → EM	SC → IN	0	4	INVALIDA
UNLOCK	ESC(W)	EM	IN	0	1	-
	ESC	EM → IN	IN → EM	1	1+4+N'	LE (CHE 0) e INV

CPU 0	CPU1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		IN	EC	-	0	-
LOCK		IN	EC	0	1 + N	NOP
LE		IN → SC	EC → SC	0	10	LE
	ESC(W)	SC	SC	0	0	-
ESC	ESC(W)	SC → EM	SC → IN	0	4	INVALIDA
UNLOCK	ESC(W)	EM	IN	0	1	-
	ESC	EM → IN	IN → EM	1	1+4+N'	LE (CHE 0) e INV

CPU 0	CPU1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		IN	SM	-	0	-
LOCK		IN	SM	0	1 + N	NOP
LE		IN → SC	SM	0	10	LE
	ESC(W)	SC	SM	0	0	-
ESC	ESC(W)	SC → EM	SM → IN	0	4	INVALIDA
UNLOCK	ESC(W)	EM	IN	0	1	-
	ESC	EM → IN	IN → EM	1	1+4+N'	LE (CHE 0) e INV

CPU 0	CPU1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		IN	EM	-	0	-
LOCK		IN	EM	0	1 + N	NOP
LE		IN → SC	EM → SM	0	10	LE
	ESC(W)	SC	SM	0	0	-
ESC	ESC(W)	SC → EM	SM → IN	0	4	INVALIDA
UNLOCK	ESC(W)	EM	IN	0	1	-
	ESC	EM → IN	IN → EM	1	1+4+N'	LE (CHE 0) e INV

CPU 0	CPU1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		SC	IN	-	0	-
LOCK		SC	IN	0	1 + N	NOP
LE		SC	IN	0	10	-
	ESC(W)	SC	MISS	0	0	-
ESC	ESC(W)	SC → EM	IN	0	4	INVALIDA
UNLOCK	ESC(W)	EM	IN	0	1	-
	ESC	EM → IN	IN → EM	1	1+4+N'	LE (CHE 0) e INV

CPU 0	CPU1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		SC	SC	-	0	-
LOCK		SC	SC	0	1 + N	NOP
LE		SC	SC	0	10	-
	ESC(W)	SC	SC	0	0	-
ESC	ESC(W)	SC → EM	SC → IN	0	4	INVALIDA
UNLOCK	ESC(W)	EM	IN	0	1	-
	ESC	EM → IN	IN → EM	1	1+4+N'	LE (CHE 0) e INV

CPU 0	CPU 1	CACHE 0	CACHE 1
		SC	EC

NÃO EXISTE ESTE CASO

CPU 0	CPU1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		SC	SM	-	0	-
LOCK		SC	SM	0	1 + N	NOP
LE		SC	SM	0	10	-
	ESC(W)	SC	SM	0	0	-
ESC	ESC(W)	SC → EM	SM → IN	0	4	INVALIDA
UNLOCK	ESC(W)	EM	IN	0	1	-
	ESC	EM → IN	IN → EM	1	1+4+N'	LE (CHE 0) e INV

CPU 0	CPU 1	CACHE 0	CACHE 1
		SC	EM

NÃO EXISTE ESTE CASO

CPU 0	CPU1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		EC	IN	-	0	-
LOCK		EC	IN	0	1 + N	NOP
LE		EC	IN	0	10	-
	ESC(W)	EC	MISS	0	0	-
ESC	ESC(W)	EC → EM	IN	0	1	-
UNLOCK	ESC(W)	EM	IN	0	1	-
	ESC	EM → IN	IN → EM	1	1+4+N'	LE (CHE 0) e INV

CPU 0	CPU 1	CACHE 0	CACHE 1
		EC	SC

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		EC	EC

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		EC	SM

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		EC	EM

NÃO EXISTE ESTE CASO

CPU 0	CPU1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		EC	IN	-	0	-
LOCK		EC	IN	0	1 + N	NOP
LE		EC	IN	0	10	-
	ESC(W)	EC	MISS	0	0	-
ESC	ESC(W)	EC → EM	IN	0	1	-
UNLOCK	ESC(W)	EM	IN	0	1	-
	ESC	EM → IN	IN → EM	1	1+4+N'	LE (CHE 0) e INV

CPU 0	CPU 1	CACHE 0	CACHE 1
		EC	SC

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		EC	EC

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		EC	SM

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		EC	EM

NÃO EXISTE ESTE CASO

CPU 0	CPU1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		SM	IN	-	0	-
LOCK		SM	IN	0	1 + N	NOP
LE		SM	IN	0	10	-
	ESC(W)	SM	MISS	0	0	-
ESC	ESC(W)	SM → EM	IN	0	4	INVALIDA
UNLOCK	ESC(W)	EM	IN	0	1	-
	ESC	EM → IN	IN → EM	1	1+4+N'	LE (CHE 0) e INV

CPU 0	CPU1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		SM	SC	-	0	-
LOCK		SM	SC	0	1 + N	NOP
LE		SM	SC	0	10	-
	ESC(W)	SM	SC	0	0	-
ESC	ESC(W)	SM → EM	SC → IN	0	4	INVALIDA
UNLOCK	ESC(W)	EM	IN	0	1	-
	ESC	EM → IN	IN → EM	1	1+4+N'	LE (CHE 0) e INV

CPU 0	CPU 1	CACHE 0	CACHE 1
		SM	EC

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		SM	SM

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		SM	EM

NÃO EXISTE ESTE CASO

CPU 0	CPU1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		EM	IN	-	0	-
LOCK		EM	IN	0	1 + N	NOP
LE		EM	IN	0	10	-
	ESC(W)	EM	MISS	0	0	-
ESC	ESC(W)	EM	IN	0	1	-
UNLOCK	ESC(W)	EM	IN	0	1	-
	ESC	EM → IN	IN → EM	1	1+4+N'	LE (CHE 0) e INV

CPU 0	CPU 1	CACHE 0	CACHE 1
		EM	SC

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		EM	EC

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		EM	SM

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		EM	EM

NÃO EXISTE ESTE CASO

Dois casos de READ MODIFY WRITE:

sem *cache* ⊟ 1 + N + 10 + 5 + 1 + N'+ 10 + 5 = 32 + N + N'

CPU 0	CPU 1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		IN	IN	-	0	-
LOCK		IN	IN	0	1 + N	NOP
LE	LOCK	IN → EC	IN	0	10	LE
	(W)	EC	IN	0	0	-
ESC	(W)	EC → EM	IN	0	1	-
UNLOCK	(W)	EM	IN	0	1	-
	LE	EM → SM	IN → SC	1	1+4+N'	LE (CHE 0)
	ESC	SM → IN	SC → EM	1	4	INVALIDA
	UNLOCK	IN	EM	1	1	-

CPU 0	CPU 1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		IN	SC	-	0	-
LOCK		IN	SC	0	1 + N	NOP
LE	LOCK	IN → SC	SC	0	10	LE
	(W)	SC	SC	0	0	-
ESC	(W)	SC → EM	SC	0	4	INVALIDA
UNLOCK	(W)	EM	SC	0	1	-
	LE	EM → SM	SC	1	1+4+N'	LE (CHE 0)
	ESC	SM → IN	SC → EM	1	4	INVALIDA
	UNLOCK	IN	EM	1	1	-

CPU 0	CPU 1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		IN	EC	-	0	-
LOCK		IN	EC	0	1 + N	NOP
LE	LOCK	IN → SC	EC → SC	0	10	LE
	(W)	SC	SC	0	0	-
ESC	(W)	SC → EM	SC → IN	0	4	INVALIDA
UNLOCK	(W)	EM	IN	0	1	-
	LE	EM → SM	IN → SC	1	1+4+N'	LE (CHE 0)
	ESC	SM → IN	SC → EM	1	4	INVALIDA
	UNLOCK	IN	EM	1	1	-

CPU 0	CPU 1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		IN	SM	-	0	-
LOCK		IN	SM	0	1 + N	NOP
LE	LOCK	IN → SC	SM	0	10	LE
	(W)	SC	SM	0	0	-
ESC	(W)	SC → EM	SM → IN	0	4	INVALIDA
UNLOCK	(W)	EM	IN	0	1	-
	LE	EM → SM	IN → SC	1	1+4+N'	LE (CHE 0)
	ESC	SM → IN	SC → EM	1	4	INVALIDA
	UNLOCK	IN	EM	1	1	-

CPU 0	CPU 1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		IN	EM	-	0	-
LOCK		IN	EM	0	1 + N	NOP
LE	LOCK	IN → SC	EM → SM	0	10	LE
	(W)	SC	SM	0	0	-
ESC	(W)	SC → EM	SM → IN	0	4	INVALIDA
UNLOCK	(W)	EM	IN	0	1	-
	LE	EM → SM	IN → SC	1	1+4+N'	LE (CHE 0)
	ESC	SM → IN	SC → EM	1	4	INVALIDA
	UNLOCK	IN	EM	1	1	-

CPU 0	CPU 1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		SC	IN	-	0	-
LOCK		SC	IN	0	1 + N	NOP
LE	LOCK	SC	IN	0	10	-
	(W)	SC	IN	0	0	-
ESC	(W)	SC → EM	IN	0	4	INVALIDA
UNLOCK	(W)	EM	IN	0	1	-
	LE	EM → SM	IN → SC	1	1+4+N'	LE (CHE 0)
	ESC	SM → IN	SC → EM	1	4	INVALIDA
	UNLOCK	IN	EM	1	1	-

CPU 0	CPU 1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		SC	SC	-	0	-
LOCK		SC	SC	0	1 + N	NOP
LE	LOCK	SC	SC	0	10	-
	(W)	SC	SC	0	0	-
ESC	(W)	SC → EM	SC → IN	0	4	INVALIDA
UNLOCK	(W)	EM	IN	0	1	-
	LE	EM → SM	IN → SC	1	1+4+N'	LE (CHE 0)
	ESC	SM → IN	SC → EM	1	4	INVALIDA
	UNLOCK	IN	EM	1	1	-

CPU 0	CPU 1	CACHE 0	CACHE 1
		SC	EC

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		SC	SM	-	0	-
LOCK		SC	SM	0	1 + N	NOP
LE	LOCK	SC	SM	0	10	-
	(W)	SC	SM	0	0	-
ESC	(W)	SC → EM	SM → IN	0	4	INVALIDA
UNLOCK	(W)	EM	IN	0	1	-
	LE	EM → SM	IN → SC	1	1+4+N'	LE (CHE 0)
	ESC	SM → IN	SC → EM	1	4	INVALIDA
	UNLOCK	IN	EM	1	1	-

CPU 0	CPU 1	CACHE 0	CACHE 1
		SC	EM

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		EC	IN	-	0	-
LOCK		EC	IN	0	1 + N	NOP
LE	LOCK	EC	IN	0	10	-
	(W)	EC	IN	0	0	-
ESC	(W)	EC → EM	IN	0	1	-
UNLOCK	(W)	EM	IN	0	1	-
	LE	EM → SM	IN → SC	1	1+4+N'	LE (CHE 0)
	ESC	SM → IN	SC → EM	1	4	INVALIDA
	UNLOCK	IN	EM	1	1	-

CPU 0	CPU 1	CACHE 0	CACHE 1
		EC	SC

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		EC	EC

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		EC	SM

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		EC	EM

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		EC	IN	-	0	-
LOCK		EC	IN	0	1 + N	NOP
LE	LOCK	EC	IN	0	10	-
	(W)	EC	IN	0	0	-
ESC	(W)	EC → EM	IN	0	1	-
UNLOCK	(W)	EM	IN	0	1	-
	LE	EM → SM	IN → SC	1	1+4+N'	LE (CHE 0)
	ESC	SM → IN	SC → EM	1	4	INVALIDA
	UNLOCK	IN	EM	1	1	-

CPU 0	CPU 1	CACHE 0	CACHE 1
		EC	SC

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		EC	EC

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		EC	SM

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		EC	EM

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		SM	IN	-	0	-
LOCK		SM	IN	0	1 + N	NOP
LE	LOCK	SM	IN	0	10	-
	(W)	SM	IN	0	0	-
ESC	(W)	SM → EM	IN	0	4	INVALIDA
UNLOCK	(W)	EM	IN	0	1	-
	LE	EM → SM	IN → SC	1	1+4+N'	LE (CHE 0)
	ESC	SM → IN	SC → EM	1	4	INVALIDA
	UNLOCK	IN	EM	1	1	-

CPU 0	CPU 1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		SM	SC	-	0	-
LOCK		SM	SC	0	1 + N	NOP
LE	LOCK	SM	SC	0	10	-
	(W)	SM	SC	0	0	-
ESC	(W)	SC → EM	SC → IN	0	4	INVALIDA
UNLOCK	(W)	EM	IN	0	1	-
	LE	EM → SM	IN → SC	1	1+4+N'	LE (CHE 0)
	ESC	SM → IN	SC → EM	1	4	INVALIDA
	UNLOCK	IN	EM	1	1	-

CPU 0	CPU 1	CACHE 0	CACHE 1
		SM	EC

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		SM	SM

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1
		SM	EM

NÃO EXISTE ESTE CASO

CPU 0	CPU 1	CACHE 0	CACHE 1	BARRA	CICLOS	AÇÃO
		EM	IN	-	0	-
LOCK		EM	IN	0	1 + N	NOP
LE	LOCK	EM	IN	0	10	-
	(W)	EM	IN	0	0	-
ESC	(W)	EM	IN	0	1	-
UNLOCK	(W)	EM	IN	0	1	-
	LE	EM → SM	IN → SC	1	1+4+N'	LE (CHE 0)
	ESC	SM → IN	SC → EM	1	4	INVALIDA
	UNLOCK	IN	EM	1	1	-

CPU 0	CPU 1	CACHE 0	CACHE 1
		EM	SC
NÃO EXISTE ESTE CASO			

CPU 0	CPU 1	CACHE 0	CACHE 1
		EM	EC
NÃO EXISTE ESTE CASO			

CPU 0	CPU 1	CACHE 0	CACHE 1
		EM	SM
NÃO EXISTE ESTE CASO			

CPU 0	CPU 1	CACHE 0	CACHE 1
		EM	EM
NÃO EXISTE ESTE CASO			

Apêndice D - Desenvolvimento de Fórmulas

$$NPE = \frac{TCACHE * PHIT}{(1 - PHIT)\ TBUS} + 1$$

onde: NPE = número de processadores

PHIT = taxa de acerto (*hit rate*)

TBUS = tempo total de acesso ao barramento

TCACHE = tempo total de acesso à *cache*

Em um determinado intervalo de tempo T, um processador ficou realizando acessos na *cache* durante um intervalo de tempo A equivalente a:

$$A = T \times TCACHE \times PHIT$$

Neste mesmo intervalo de tempo, este mesmo processador ficou realizando acessos a memória utilizando o barramento com tempo total B equivalente a:

$$B = T \times TBUS \times (1 - PHIT)$$

Durante o tempo B, outros processadores não podem utilizar o barramento, restando apenas o tempo A. Sem perda da generalidade, pode-se supor que o tempo A é maior do que o tempo B. Então, durante o tempo em que o primeiro processador ficou realizando acessos à sua *cache*, puderam ser realizados B/A tempos B referentes a B/A processadores no barramento. Logo, este sistema suporta (B/A) +1 processadores, ou seja:

$$NPE = \frac{TCACHE * PHIT}{(1 - PHIT)\ TBUS} + 1$$

No caso de B não ser multiplo de A, o resultado deve ser arrendondado para cima e acarretará em um acréssimo no tempo parado dos processadores devido a conflitos no barramento.