Análise e Simulação de Protocolos de Coerência de Cache para Sistemas Multiprocessados

Antonio Carlos Fontes Atta

1993

Análise e Simulação de Protocolos de Coerência de Cache para Sistemas Multiprocessados

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pelo Sr. Antonio Carlos Fontes Átta e aprovada pela Comissão Julgadora.

Campinas, 03 de fevereiro de 1994.

Prof. Célio Cardoso Guimarães
Orientador

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Análise e Simulação de Protocolos de Coerência de Cache para Sistemas Multiprocessados[1]

Antonio Carlos Fontes Atta[2]

Departamento de Ciência da Computação

IMECC – UNICAMP

Banca Examinadora:

- Célio Cardoso Guimarães (Orientador)[3]
- Nelson Castro Machado[3]
- Maria Beatriz Felgar de Toledo[3]

[1]Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação da UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

[2]O autor é Bacharel em Ciência da Computação pela Universidade Federal da Bahia.

[3]Professor do Departamento de Ciência da Computação - IMECC - UNICAMP.

À minha família, em especial a meus pais.

Agradecimentos

Ao CNPq e à UFBa, pelo apoio financeiro recebido.

Ao meu orientador, Prof. Célio Cardoso Guimarães, pelo tempo dedicado e pelos comentários valiosos que, sem dúvida, influenciaram e deram forma a este trabalho.

Ao Prof. Geovane Cayres Magalhães, pelo incentivo e apoio a mim dispensados desde a época da graduação.

A Inês, por sua companhia e apoio irrestritos.

Finalmente, a todos os meus colegas de pós–graduação, principalmente os da turma do vôlei.

Resumo

Para garantir um rendimento aceitável dos sistemas multiprocessados de memória compartilhada através da redução das disputas pelo acesso à memória e à rede de interconexão, memórias cache têm sido utilizadas, a exemplo dos sistemas monoprocessados, para armazenar localmente as informações mais freqüentemente requeridas pelos processadores. A possibilidade de existência de diversas cópias de um mesmo dado espalhadas pelos caches do sistema, entretanto, dá origem ao problema da consistência ou coerência da informação armazenada em cache nos sistemas multiprocessados.

Nesta dissertação, nós avaliamos conceitualmente algumas das soluções propostas para o problema, explorando tanto as soluções voltadas a sistemas multiprocessados que adotam o barramento como rede de interconexão, quanto as soluções voltadas a redes mais genéricas, como as redes tipo multiestágios. Adicionalmente, o estudo dessa última classe de soluções é aprofundado para 2 soluções básicas da classe, a que emprega diretórios totalmente mapeados e a que emprega diretórios limitados, sendo proposta uma extensão à técnica de diretórios limitados de modo a tornar seu desempenho tão alto quanto o obtido com os diretórios totalmente mapeados — mais caros em termos de espaço — mantendo a mesma eficiência de espaço da solução original. Para comparar as três soluções foi desenvolvido um simulador baseado na geração sintética de referências à memória a partir das estatísticas divulgadas de aplicações paralelas reais.

Abstract

In order to guarantee reasonable performance of shared-memory multiprocessors reducing memory and interconnect network contention, cache memories have been used, as in uniprocessors systems, to keep locally frequently required by processors information. The possibility of existence of many modifiable copies of the same data spread into the caches of the system originates the cache coherence problem though.

In this dissertation, we conceptually study some of the proposed solutions to the problem, exploring solutions suitable for shared bus multiprocessors and solutions oriented to systems where the processors and memories are interconnected by more general networks, such as multistage network. Furthemore this last class of solutions is detailed for 2 basic techniques, full map directories and limited directories. We propose an extension to the limited directory technique with the aim of getting performance as high as with full map directories — which are more expensive with regard to space — but with the same space efficiency of the original solution. In order to compare these 3 solutions we developped a simulator based on synthetic trace derived from real applications.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Sistemas multiprocessados de memória compartilhada têm sido propostos como uma das soluções viáveis para suprir a necessidade de equipamentos com maior poder computacional que o oferecido pelos sistemas monoprocessados produzidos com a tecnologia atual. Em tais sistemas, diversos processadores operam de forma independente e compartilham um único espaço global de endereçamento formado pelos módulos de memória, segundo uma configuração semelhante à mostrada na Figura 1.1. O acesso à memória é feito através de uma rede de interconexão.

Dentre as vantagens apontadas por aqueles que defendem a solução multiprocessada de memória compartilhada, podem ser destacadas [Rash87]:

- A tecnologia de microprocessadores encontra-se bastante evoluída produzindo componentes com capacidade de processamento e espaço de endereçamento suficientes para permitirem a construção de sistemas multiprocessados com um número elevado de unidades processadoras a um custo reduzido.
- O ambiente de programação paralela com acesso compartilhado à memória é compatível com o ambiente de programação convencional tornando mais rápida e fácil a adaptação do programador e das aplicações já desenvolvidas no ambiente convencional ao novo modelo.

Por outro lado, a existência de diversos processadores disputando a utilização da rede de interconexão e dos módulos de memória tornam esses componentes elementos críticos para o desempenho do sistema. A exemplo dos sistemas monoprocessados, memórias cache têm sido usadas nos sistemas multiprocessados como forma de obter-se a redução na necessidade de efetuar acessos à memória e, por conseguinte, na utilização da rede, objetivando, com isso, o

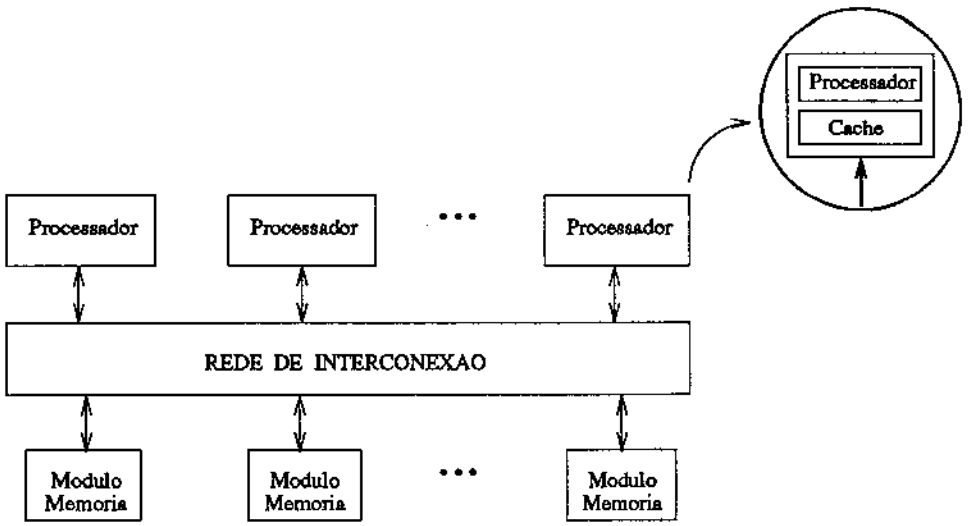

Figura 1.1: Exemplo de configuração de um sistema multiprocessado de memória compartilhada.

aumento do desempenho global do sistema. Para que isso seja possível, cada processador deve ser dotado de uma memória cache local (detalhe na Figura 1.1) responsável pelo atendimento da maioria das referências enviadas pelo processador, só sendo utilizado o subsistema de memória (rede de interconexão e módulo de memória) quando não for possível efetuar o atendimento localmente. A existência de caches locais, entretanto, dá origem ao problema da consistência das informações distribuídas pelos caches ou, simplesmente, **problema da coerência de cache**.

Na seção 1.1 a seguir, o objetivo e as principais contribuições desta dissertação são apresentados. A seção 1.2 seguinte revê brevemente alguns conceitos básicos das memórias cache, enquanto que na seção 1.3 o problema da coerência de cache é apresentado de forma mais ampla. A seção 1.4 termina o capítulo com a descrição de como a dissertação está organizada.

1.1 Objetivo e Principais Contribuições

O objetivo principal desta dissertação é avaliar conceitualmente as soluções existentes a nível de *hardware* para o problema da coerência de cache em sistemas multiprocessados de memória compartilhada, analisando sempre a eficiência e as restrições impostas por cada uma delas. Basicamente duas classes de soluções são estudadas. A primeira delas compreende as soluções orientadas a sistemas que usam o barramento como rede de interconexão entre os processadores e a memória — geralmente sistemas pequenos com poucos processadores. A segunda classe, abrange as soluções orientadas a redes de interconexão mais genéricas, como as redes tipo

multiestágios, que permitem a conexão de um número elevado de processadores a um custo do aumento da latência média da comunicação à medida que o sistema expande.

Adicionalmente, considerando a crescente necessidade por sistemas multiprocessados cada vez maiores e mais potentes, o estudo das soluções orientadas às redes genéricas é aprofundado com uma avaliação comparativa, baseada em simulação, de uma solução proposta neste trabalho juntamente com dois tipos básicos de soluções existentes nessa classe.

As principais contribuições desta dissertação estão resumidas a seguir.

1. Uma revisão ampla das principais soluções propostas até hoje para o problema da coerência de cache. Essa é uma área de importância fundamental para as futuras gerações de sistemas multiprocessados de memória compartilhada e, por conseguinte, tem recebido grande atenção por parte de vários grupos de pesquisa, resultando em um número razoável de soluções propostas.

2. Uma nova solução que trata o problema da coerência de forma eficiente em sistemas com um número de processadores que supera os conseguidos nos sistemas efetivamente desenvolvidos até hoje.

3. Um ambiente de simulação, altamente parametrizável, com funções que compreendem todo subsistema de memória de uma máquina multiprocessada comum, capaz de gerar, em tempo de simulação, referências à memória (*traces*) sintéticas (baseadas no comportamento de aplicações paralelas reais) dispensando a necessidade de armazenamento dessas referências, que normalmente demanda uma enorme quantidade de espaço em disco. Esse ambiente foi usado na avaliação de algumas das soluções estudadas e a sua utilização no estudo de novas propostas, protocolos de comunicação, modelos de rede de interconexão, modelos de consistência de memória, etc. pode ser conseguida, com muito pouco esforço de reprogramação.

1.2 Memórias Cache

Memórias cache são memórias de alta velocidade especialmente projetadas para armazenarem temporariamente partes do conteúdo da memória principal em uso corrente pelo processador [Smit82], [HwBr90]. Por possuírem um tempo de resposta cerca de 5 a 10 vezes inferior ao da memória principal [Ston87] elas são capazes de reduzir o tempo efetivo de espera do processador pelo atendimento às referências feitas durante a execução de uma aplicação.

O algoritmo executado pelo controlador da memória cache é relativamente simples. A cada referência enviada pelo processador, a memória cache ou, simplesmente, cache [1] é verificada e, caso o dado solicitado esteja disponível, diz-se que um *acerto* (*hit*) ocorreu e o atendimento é feito imediatamente sem incorrer em atrasos para o processador. Caso o cache não contenha o dado solicitado, diz-se que uma *ausência* (*miss*) ocorreu e um processo de busca pelo dado na memória principal é iniciado. A resposta enviada pela memória é tanto suprida ao processador quanto armazenada no cache.

Internamente, o cache é normalmente constituído por duas partes: um *diretório* e uma área de armazenamento. O diretório é usado pelo controlador do cache para identificar que partes da memória principal estão disponíveis na área de armazenamento, já que, seu tamanho é inferior ao da memória principal. A unidade de transferência entre a memória principal e o cache é chamada *bloco* ou *linha* do cache. Um bloco é constituído de um número fixo de unidades de endereçamento (*bytes*) que definem seu tamanho, geralmente maior que 1 (4, 8 ou 16 *bytes* são valores típicos). Os blocos são alocados em posições fixas na área de armazenamento. Sempre que um bloco é trazido da memória e uma posição já ocupada é alocada para ele, uma *substituição* é feita e o bloco substituído só pode ser alcançado mediante um novo acesso à memória.

As três principais formas segundo as quais os blocos são alocados na área de armazenamento são as seguintes:

1. **Mapeamento Direto** (*Direct Mapping*) - cada bloco de memória está associado a uma posição fixa na área de armazenamento, não podendo ser colocado em nenhuma outra posição, mesmo que estas estejam livres. Essa política tem a vantagem de tornar a lógica do controlador simples apesar de ocasionar taxas de acerto (*hit ratio*) menores que as apresentadas por outras políticas.

2. **Conjunto Associativo** (*Set-Associative*) - a área de armazenamento é dividida em conjuntos com n posições, sendo que cada bloco de memória está associado a um conjunto fixo mas pode ser armazenado em qualquer posição dentro do conjunto (com isso, retarda-se a necessidade de substituição até o instante em que todas as posições do conjunto estiverem ocupadas – além de poder-se optar por algum algoritmo mais elaborado para a escolha do bloco a ser substituído). Essa política apresenta taxas de acerto superiores à solução anterior mas torna a lógica do controlador de cache mais complexa. Tipicamente,

[1] Considerando a freqüência com que o termo "cache" é usado neste trabalho, suas ocorências não serão grifadas (como em *cache*)

o número de posições em cada conjunto é reduzido (2–4). É a política mais adotada nos sistemas atuais.

3. **Mapeamento aleatório** (*Fully-Associative*) - Não existem posições fixas para alocar os blocos de memória na área de armazenamento. Qualquer bloco pode ocupar qualquer posição. Essa política é a mais cara e complexa das três, sendo raramente adotada.

Outro fator que caracteriza um projeto que utiliza memórias cache diz respeito à forma com que a memória é atualizada quando as referências feitas pelo processador modificam o conteúdo do bloco no cache (operações de escrita). Uma das políticas possíveis, conhecida como *write-through*, atualiza a memória concorrentemente com a modificação do bloco no cache. Uma outra política, a *copy-back*, modifica apenas o dado no cache retardando a atualização da memória até o momento em que o bloco modificado tenha que ser substituído no cache.

O sucesso das memórias cache, mais do que às suas características físicas e de implementação, está associado a uma propriedade comum apresentada pelas aplicações quando em execução, relacionada à distribuição das referências feitas no seu espaço de endereçamento. Essa propriedade, também explorada pelos sistemas de gerenciamento de memória virtual, é conhecida como "localidade de referência" [Smit82]. Em seu aspecto temporal, a localidade de referência dita que, ao longo de curtos períodos de tempo um programa em execução distribui suas referências não uniformemente no espaço de endereçamento e que algumas dessas referências são favorecidas sendo largamente repetidas no decorrer do período (ex. referências internas a um *loop*). Em seu aspecto espacial, a localidade de referência dita que partes do espaço de endereçamento em uso corrente geralmente corresponde a porções contíguas desse espaço (ex. operações em estruturas de dados como *arrays*).

1.3 O problema da Coerência de Cache

A possibilidade de existência de cópias de um dado em diferentes caches e na memória em um sistema multiprocessado de memória compartilhada dá origem ao problema da consistência ou da coerência de cache. Segundo Censier e Feautrier [CeFe78], um dado está coerente quando qualquer referência feita a esse dado retorna sempre o seu valor após a última modificação que ele tenha sofrido. Para que essa premissa seja garantida, a modificação de um dado compartilhado deve ocasionar a invalidação ou a atualização de todas as cópias dos dados distribuídos no sistema. Adicionalmente, para garantir um modelo de consistência de memória *seqüencial*

como definido por Lamport[2] [Lamp79], essas invalidações ou atualizações devem ser feitas de forma atômica com a modificação que causou a inconsistência.

Ao conjunto de ações que são efetuadas visando garantir a consistência dos caches de um sistema multiprocessado dá-se o nome de protocolo de coerência de cache. Basicamente, existem protocolos de coerência de cache a nível de *software* e a nível de *hardware*. Nas soluções por *software* a tarefa de manter a coerência recai, normalmente, sobre o compilador. Através da análise da aplicação paralela, procura-se detectar e marcar as estruturas de dados que podem ser livremente compartilhadas pelos caches e as que não devem existir em cache — o acesso a estas últimas é feito diretamente através da memória principal. No primeiro caso encontram-se, por exemplo, as estruturas compartilhadas para leitura apenas; no segundo, as estruturas compartilhadas que podem vir a sofrer modificações [Sten90], [MiBa92]. Soluções por *software*, todavia, apresentam algumas limitações. Como a identificação do tipo da estrutura é feita em tempo de compilação, os protocolos são obrigados a assumir o pior quando essa identificação não é possível, ou seja, que a estrutura envolvida deve residir exclusivamente na memória principal; além disso, efetuar esse tipo de identificação em programas que fazem uso pesado de apontadores se torna bastante complexo e muitas vezes inviável. Este trabalho não pretende se aprofundar nas soluções a nível de *software*, restringindo-se apenas às soluções a nível de *hardware*.

A manutenção da coerência a nível de *hardware* é realizada em tempo de execução pelos controladores de memória e de cache sendo diretamente influenciada pela rede de interconexão usada no sistema. Basicamente, soluções envolvendo dois tipos de redes serão apresentadas e discutidas neste trabalho: soluções voltadas para redes com mecanismo eficiente de difusão (*broadcasting*), como as redes tipo barramentos, e redes incapazes de oferecer esse mecanismo de forma eficiente, como as redes de chaves multiestágios. As soluções voltadas para barramentos representam a primeira geração de protocolos de coerência e está restrita, devido às características desse tipo de rede, à arquiteturas com poucas dezenas de elementos processadores. As soluções para redes não barramentos representam a atual geração de protocolos (apesar de alguns deles terem sido propostos mesmo antes das soluções para barramentos surgirem) e visam vencer o desafio da construção de arquiteturas multiprocessadas com centenas e milhares de elementos processadores.

[2]Segundo Lamport, um sistema multiprocessado está seqüencialmente consistente se o resultado de qualquer execução de um programa paralelo é similar ao obtido se as operações de todos os processadores fossem executadas em alguma ordem seqüencial, e as operações em cada processador fossem executadas na ordem estabelecida por sua parte do programa.

1.4 Organização da Dissertação

Este trabalho pode ser dividido em quatro partes:

Parte I – representada por este capítulo, apresenta o problema e define alguns dos conceitos (particularmente relacionados às memórias cache) que serão utilizados no restante do trabalho.

Parte II – representada pelos capítulos 2 e 3, apresenta e discute as diversas soluções propostas para arquiteturas com redes tipo barramento e outros tipos de redes respectivamente.

Parte III – representada pelo capítulo 4, propõe um protocolo de coerência para redes não barramentos que resgata a utilização (especializada) desses dispositivos nas futuras gerações de sistemas multiprocessados.

Parte IV – representada pelos capítulos 5 e 6, apresenta um ambiente de simulação (cap. 5) desenvolvido para avaliar o desempenho (cap. 6) do protocolo proposto no capítulo 4 comparativamente com outras soluções da mesma classe revistas no capítulo 3.

O capítulo 7 conclui o trabalho destacando as suas contribuições e apresentando algumas propostas de futuras extensões.

Capítulo 2

Protocolos direcionados a redes tipo barramento

Tradicionalmente, os projetistas de sistemas monoprocessados têm utilizado o barramento como meio físico básico para a comunicação entre os diversos dispositivos que compõem um sistema de computação. É natural, portanto, que os primeiros protótipos de sistemas multiprocessados também fossem construídos com base nesse tipo de rede de interconexão.[1] Além disso, fatores comerciais, como o baixo custo e a possibilidade de padronização, e de projeto, como a capacidade de adição incremental de novos dispositivos — placas — a um sistema já existente, também contribuíram fortemente.

Nos sistemas multiprocessados de memória compartilhada baseados em barramentos, os processadores enxergam os módulos de memória como um único espaço global de endereçamento cujo acesso é feito através do barramento, como esquematizado na Figura 2.1. Pesquisas constantes na área de barramentos e a necessidade de alto desempenho das presentes e futuras gerações de processadores proporcionaram o ambiente ideal para o surgimento de diversos protocolos,[2] dentre eles, o VME-bus, o Multibus, o NuBus, o Fastbus e o FutureBus, cada um buscando solucionar as deficiências impostas por seus antecessores [SwSm86]. Por mais eficientes que esses padrões sejam, contudo, não é difícil perceber que, devido à forma compar-

[1] Mesmo nos sistemas monoprocessados, a maioria dos dispositivos, como os de E/S por exemplo, são interligados ao barramento através de processadores dedicados.

[2] O termo protocolo, nesse contexto, significa método adotado em um barramento para sinalizar a validade de endereços, dados, comandos e estados, especificando, dessa forma, um padrão para os dispositivos a ele conectados. A menos que explicitamente referenciado de outra forma, sua utilização neste trabalho estará associada aos protocolos de coerência.

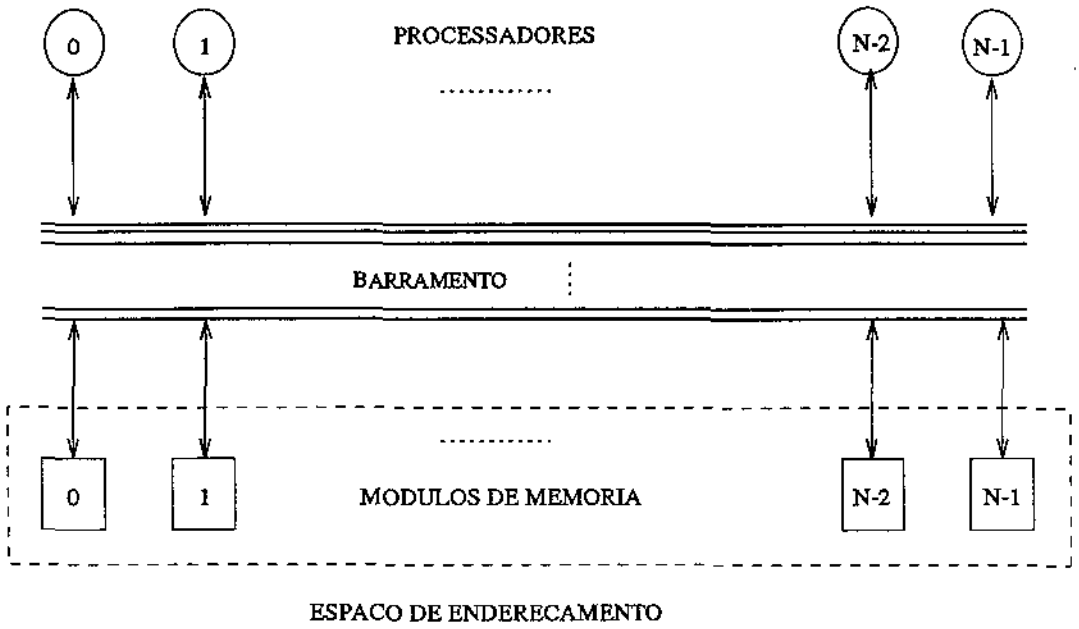

Figura 2.1: Modelo de sistema multiprocessado usando barramento.

tilhada com que os processadores usam o barramento (um único por vez com o direito de acesso arbitrado pelo protocolo do barramento), mesmo com um número não muito significativo de processadores, o barramento se torna alvo de competição resultando na redução do desempenho global do sistema.

O emprego de memórias cache associadas a cada processador — caches privados — mais do que visar a redução do tempo de acesso aos dados, tem como objetivo reduzir ao mínimo possível a competição pelo barramento. Dessa forma, um número maior de processadores pode ser conectado ao sistema tornando atraente o investimento na construção de sistemas multiprocessados. Essa configuração, porém, introduz, como visto, o problema da coerência ou consistência dos dados armazenados no cache.[3]

A consistência dos dados nos caches tem sido mantida em sistemas com barramento, por meio de protocolos que, em sua maioria, exploram a característica de transmissão (*broadcast*) inerente a esse tipo de rede. No universo de sinais que transitam pelo barramento passa a existir, nos sistemas multiprocessados, um subconjunto conhecido como *comandos de consistência* que, enviados por um dos caches, devem ser recebidos e tratados pelos demais a fim de garantir a consistência. A eficiência de um protocolo de coerência está, entre outros fatores, associada à quantidade e a forma com que esses comandos de consistência são tratados. Um determinado

[3]Uma outra configuração possível é aquela em que os caches são também compartilhados pelos processadores. Apesar de eliminar o problema da coerência, essa configuração não ajuda a diminuir a contensão no barramento.

conjunto de comandos define e diferencia um protocolo dos demais. Para que o sistema de transmissão — detecção, captação e tratamento dos comandos — funcione sem degradação no desempenho global não se deve, entretanto, impor nenhum tipo de envolvimento dos processadores nessa tarefa. Caso isso ocorresse, grande parte do tempo útil dos processadores seria gasto observando e tratando comandos de consistência. É necessário, portanto, que cada unidade de cache disponha de um mecanismo com funções de monitoração do barramento e com "inteligência" suficiente para distinguir e tratar os comandos de consistência dos demais sinais que circulam pelo barramento. Tal mecanismo, na prática, fica "escutando" os sinais que circulam pelo barramento, o que justifica o nome dado a essa classe de protocolos: *snoopy cache protocols*. O sistema de cache passa então a ser composto de um dispositivo monitor, além das tradicionais unidades de controle, diretório, e área de armazenamento de dados. A Figura 2.2 detalha o dispositivo monitor na configuração da Figura 2.1.

Diversos protocolos *snoopy* têm sido desenvolvidos e utilizados em protótipos de sistemas multiprocessados. A solução clássica é a adotada em sistemas com apenas dois processadores [4] ou em sistemas monoprocessados com processadores de E/S independentes [CeFe78]. Essa solução utiliza a política *write–through* para manter os dados na memória sempre atualizados. Todos os caches monitoram permanentemente e invalidam os blocos (através de um *bit* de validade) correspondentes a endereços detectados no barramento em operações de escrita. Um bloco invalidado equivale a um bloco ausente e deve ser trazido da memória no momento em que o processador voltar a requisitá-lo. A coerência fica, dessa forma, garantida com um custo mínimo de comandos de consistência.

A solução clássica, todavia, apesar de simples, apresenta características que tornam a sua aplicação inadequada em sistemas com um número maior de processadores. A principal delas está relacionada à política de atualização de memória adotada. A taxa média de operações de escrita para muitas arquiteturas é de 10 – 30% do total de acessos à memória [Smit82]; tal taxa limita o número máximo de processadores conectados a um mesmo barramento a cerca de 2 – 4. Essa restrição tem justificado a utilização da política *write–back* em soluções mais recentes.

Neste capítulo serão descritos alguns dos protocolos recentes orientados para sistemas com barramento, com base na classificação e terminologia apresentadas a seguir. A escolha desses protocolos especificamente foi influenciada, principalmente, pelas idéias e/ou conceitos novos introduzidos por cada um deles e que são utilizados, de forma melhorada ou não, por outros protocolos não discutidos neste trabalho. Um aspecto comum a esses protocolos é a

[4] Nesse caso, devido ao número reduzido de processadores, a solução com caches compartilhados também é adotada.

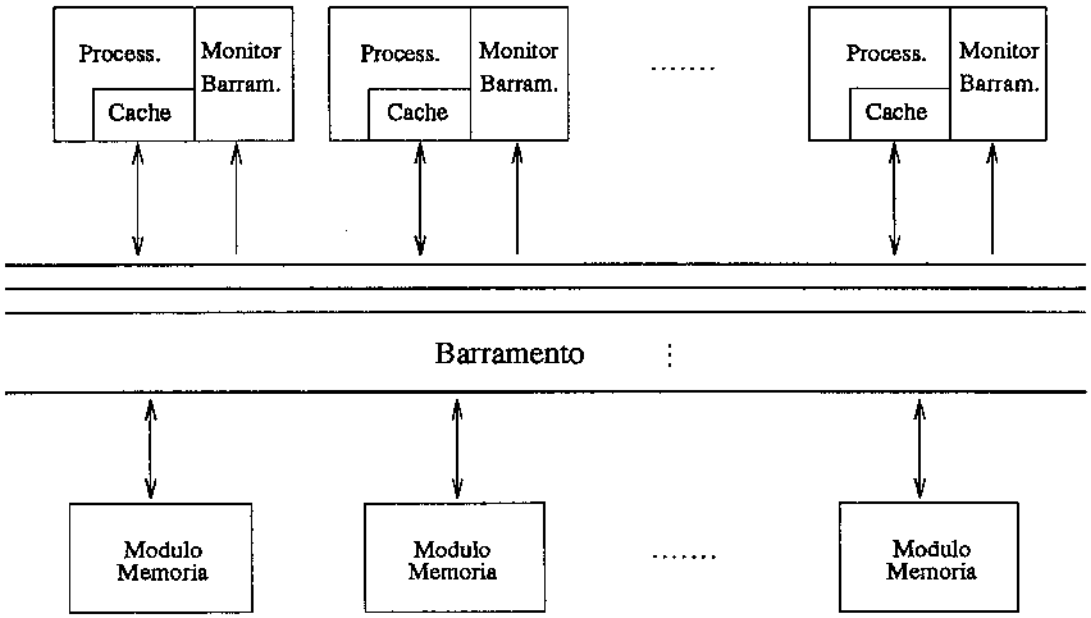

Figura 2.2: Modelo de sistema de cache com dispositivo monitor.

busca pela utilização mínima do barramento (tanto nos acessos à memória quanto na emissão de comandos de consistência), como forma de tornar a solução viável para sistemas com o maior número possível de processadores. Para cada protocolo serão apresentados o ambiente em que ele fora desenvolvido, a descrição do algoritmo usado e uma avaliação com relação aos seus pontos fortes e fracos. Opcionalmente, são descritas também outros trabalhos afins e extensões que trouxeram melhorias aos protocolos apresentados — quando eles existirem.

2.1 Classificação e Terminologia

A análise e comparação de protocolos de coerência diversos requer o estabelecimento de uma classificação e terminologia comum, sobre a qual eles possam ser especificados. Bril [Bril87] apresenta uma classificação genérica que define estados e classes abstratas que um bloco pode assumir em um sistema de caches. Com algumas adaptações, essa classificação é descrita a seguir e, juntamente com uma terminologia também introduzida, ela será utilizada na descrição dos protocolos de coerência na próxima seção.

Em sistemas multiprocessados, do ponto de vista do cache, um bloco pode estar *presente* ou *ausente*. Se o bloco está presente, seu conteúdo pode ser válido ou inválido resultando na seguinte classificação de estados possíveis:

E.1 *Ausente*(A): o bloco não pertence ao cache.

E.2 *Inválido*(I): o bloco pertence ao cache apesar de não conter informação útil.

E.3 *Válido*(V): o bloco pertence ao cache e contém informação útil.

Considerando que a informação na memória pode também ser vista como dividida em blocos tem-se, da mesma forma, *estados de memória* similares aos *estados de cache* definidos acima.[5]

Do ponto de vista do sistema (onde se procura observar o estado de um bloco em relação a **todos** os caches do sistema), o estado de um bloco em um multiprocessador com n caches é descrito por uma *(n+1)–tupla* ordenada, onde o primeiro elemento descreve o estado do bloco na memória e os n elementos seguintes descrevem seu estado em cada um dos caches. Um bloco trazido da memória para um único cache, por exemplo, resultaria no seguinte *estado de sistema*: $< V_m, V_{c1}, A_{c2}, ..., A_{cn} >$. O conjunto dos diversos *estados de sistemas* (tuplas) possíveis pode, ainda, ser subdividido em *classes* distintas. Uma possível relação de classes para um sistema de caches é a seguinte:

C.1 **Não–Referenciado**: nenhum dos caches contém uma cópia *válida* do bloco. O estado de memória do bloco é *válido*.

C.2 **Intacto–Privado**: apenas um cache contém uma cópia *válida* do bloco. O estado de memória do bloco é *válido*.

C.3 **Intacto–Compartilhado**: ao menos dois caches contêm uma cópia *válida* do bloco. O estado na memória é *válido*.

C.4 **Modificado–Privado**: apenas um cache contém uma cópia *válida* do bloco. O estado na memória é *inválido* ou (exclusivo) *ausente*.

C.5 **Modificado–Compartilhado**: ao menos dois caches contêm uma cópia válida do bloco. O estado de memória é *inválido* ou (exclusivo) *ausente*.

A definição formal da classe **Não–Referenciado** seria:

[5] A visualização dos estados *válido* e *inválido* é óbvia. O estado *ausente* pode ser caracterizado, por exemplo, quando um bloco deixa de pertencer à memória, em virtude de uma operação de substituição de página — nos sistemas paginados —, apesar de ter sido lido anteriormente por algum dos caches do sistema.

Não–Referenciado: $\{< EM, EC_1, ..., EC_n > / EM = V \text{ e } (EC_i = I \; xor \; EC_i = A), 1 \leq i \leq n\}$

onde:

EM – estado do bloco na memória;

EC_j – estado do bloco no cache j;

Considerando a ordem de distribuição dos estados nos caches irrelevante, uma definição mais clara seria:

Não–Referenciado: $V_m \quad A_c^n$ (considerando n caches)

Os índices m e c denotam estado na memória e estado no cache respectivamente.

Da mesma forma:

Intacto–Privado: $V_m \quad A_c^{n-i-1} \quad I_c^i \quad V_c \quad (0 \leq i \leq n-1)$

Intacto–Compartilhado: $V_m \quad A_c^{n-i-j} \quad I_c^i \quad V_c^j \quad (0 \leq i \leq n-j \text{ e } 2 \leq j \leq n)$

Modificado–Privado: $(I_m \; xor \; A_m) \quad A_c^{n-i-1} \quad I_c^i \quad V_c \quad (0 \leq i \leq n-1)$

Modificado–Compartilhado: $(I_m \; xor \; A_m) \quad A_c^{n-i-j} \quad I_c^i \quad V_c^j \quad (0 \leq i \leq n-j \text{ e } 2 \leq j \leq n)$

Os estados do bloco no cache e na memória podem sofrer transições de acordo com as operações que são captadas no barramento. Essas transições, por sua vez, também ocasionam transições nos estados de sistema dos blocos. As transições de estado no cache e na memória, considerando os estados *ausente*, *válido* e *inválido* e algumas das operações comuns em um sistema de cache estão relacionadas na Tabela 2.1.

O modelo de classes e estados definido aqui tem dois objetivos: 1. mostrar que é possível estabelecer um ambiente padronizado, independente de implementação, para a descrição de protocolos de coerência; 2. usá-lo como modelo inicial, sobre o qual serão descritos os protocolos da próxima seção. Sempre que algum conceito introduzido por um dos protocolos motivar a extensão ou modificação desse modelo (inclusão de novos estados, por exemplo), isso será feito juntamente a cada protocolo.

Operação	Transição
busca	$A \rightarrow V$
atualização	$I \rightarrow V$
invalidação	$V \rightarrow I$
substituição	$IxorV \rightarrow A$

Tabela 2.1: Transições de estados de blocos no cache e na memória.

2.2 Protocolos para redes tipo barramento

2.2.1 Protocolo *Write-Once*

Considerado o primeiro protocolo de coerência de cache para barramento que utiliza a política *write-back* descrito na literatura, esse esquema, proposto por Goodman [Good83], resultou de suas investigações sobre a possibilidade de implementar, efetivamente, um sistema de memória cache para um ambiente multiprocessado baseado no Multibus — um protocolo de barramento desenvolvido pela Intel que oferecia generalidade, simplicidade e baixo custo, mas que apresentava restrições severas quanto ao desempenho. Especificamente, o sistema sob investigação era composto de placas, cada uma contendo um microprocessador desprovido de memória local, exceto o cache, sendo o acesso à memória global feito via barramento. Esperava-se que a baixa taxa de transferência (*bandwidth*) do barramento fosse compensada pela alta eficiência do cache. A pesquisa buscava respostas a questões como: 1. seria possível desenvolver um sistema de cache que permitisse a conexão de várias placas ao barramento? 2. qual seria o número máximo de placas, considerando que, sem o cache, o sistema já apresentava sobrecarga com uma única placa e demais dispositivos periféricos? O enfoque adotado foi reduzir ao máximo a utilização do barramento por processadores individuais, mesmo que isso implicasse em alguma ociosidade por parte deles durante algum intervalo de tempo. Para tanto, o cache deveria ter uma alta taxa de eficiência e o *overhead* imposto pelo protocolo deveria ser mínimo.

O protocolo *write-once* recebeu esse nome devido à sua política de atualização de memória, que é um misto entre as políticas *write-through* e *write-back*. A **primeira** vez que um processador escreve num bloco, a modificação é transferida para a memória (como na política *write-through*); as próximas escritas no bloco, contudo, são feitas exclusivamente no cache sendo a memória atualizada somente quando o bloco sofrer substituição ou for requisitado por um outro processador (política *write-back*). Como não existem sinais explícitos de invalidação, a política *write-through* na primeira escrita é usada para este fim; ou seja, ao perceberem a

presença da palavra alterada no barramento através de seus dispositivos monitores, todos os outros caches verificam a existência da palavra nos seus diretórios e, caso encontrem, efetuam a invalidação do bloco correspondente.

ESTADOS

Associado a cada bloco no cache existem dois *bits* que definem um dos 4 estados para o bloco:

E.1 *Válido*(V): a informação contida no bloco está consistente com a memória; possivelmente, outros caches compartilham o bloco.

E.2 *Inválido*(I): a informação contida no bloco não está coerente com a memória.

E.3 *Reservado*(R): houve uma alteração pela primeira vez no bloco e ela foi transmitida para a memória.

E.4 *Modificado*(M): houve mais de uma modificação no bloco e a última modificação não foi transmitida para a memória.

Apesar de não existirem *bits* de estado associados aos blocos na memória, implicitamente, eles podem assumir os estados *válido*, quando a cópia na memória está atualizada, e *inválido*, caso contrário. Outro estado implícito tanto para blocos no cache quanto na memória é *ausente*.

Os *bits* de estado nos caches partem de um valor inicial (*inválido*, por exemplo) e são atualizados à medida que operações vão sendo efetuadas sobre o bloco (ele é trazido da memória, sofre uma operação de escrita, etc.). O gerenciamento desses *bits* está a cargo do controlador do cache e do monitor do barramento que fazem o acesso aos *bits* de maneira exclusiva.

ALGORITMO *WRITE-ONCE*

LEITURA

Não–Referenciado $[V_m \quad A_c^n]$
Intacto–Privado $[V_m \quad A_c^{n-i-1} \quad I_c^i \quad V_c \quad (0 \leq i \leq n-1)]$
Intacto–Compartilhado $[V_m \quad A_c^{n-i-j} \quad I_c^i \quad V_c^j \quad (0 \leq i \leq n-j \text{ e } 2 \leq j \leq n)]$

Inválido ou *Ausente* – a memória supre uma cópia do bloco que assume o estado local *válido*.

Válido – a leitura é atendida localmente.

Modificado $[I_m \quad A_c^{n-i-1} \quad I_c^i \quad R_c \; xor \; M_c \quad (0 \leq i \leq n-1)]$

Inválido ou *Ausente* – um acesso é feito à memória; se o bloco está *reservado* em outro cache, seu dispositivo monitor, ao perceber a operação no barramento, atualiza o estado de sua cópia para *válido* e a memória supre o bloco; se está *modificado*, a memória é inibida[6] e o próprio cache se encarrega de suprir o bloco, mudando o estado local para *válido* e atualizando imediatamente a seguir a cópia da memória.

Reservado ou *Modificado* – a leitura é atendida localmente.

ESCRITA

Não–Referenciado $[V_m \quad A_c^n]$

Intacto–Privado $[V_m \quad A_c^{n-i-1} \quad I_c^i \quad V_c \quad (0 \leq i \leq n-1)]$

Intacto–Compartilhado $[V_m \quad A_c^{n-i-j} \quad I_c^i \quad V_c^j \quad (0 \leq i \leq n-j \text{ e } 2 \leq j \leq n)]$

Inválido ou *Ausente* – o bloco é trazido da memória e assume o estado *modificado*; ao perceber a operação de escrita, todos os outros caches invalidam suas cópias.

Válido – a escrita é efetuada imediatamente e o bloco assume o estado *reservado*; a *palavra* que está sendo alterada é enviada à memória que também atualiza sua cópia; ao perceber a operação de escrita, todos os outros caches invalidam suas cópias.

Modificado $[I_m \quad A_c^{n-i-1} \quad I_c^i \quad R_c \; xor \; M_c \quad (0 \leq i \leq n-1)]$

Inválido ou *Ausente* – o bloco é trazido como na leitura e é alterado, assumindo o estado *modificado*; em todos os outros caches o estado do bloco fica sendo *inválido*.

[6] Através de mecanismos próprios do barramento.

Reservado – a escrita é feita imediatamente na cópia local que assume o estado *modificado*.

Modificado – a escrita é efetuada imediatamente na cópia local.

As entrada neste algoritmo correspondem à operação efetuada pelo processador (**LEITURA** ou **ESCRITA**), à classe ou estado do bloco no sistema (**Não-Referenciado**, **Intacto-Privado**, etc.) e ao estado local do bloco (*Inválido*, *Válido*, etc.) respectivamente. A Figura 2.3 apresenta o diagrama de estados que de um bloco no cache para o algoritmo *write-once*.

AVALIAÇÃO

Da forma como está descrito, o protocolo *write-once* permite a ligação direta ao barramento de qualquer tipo de dispositivo que necessite efetuar acessos à memória e que disponha ou não de cache (dispositivos de E/S, por exemplo); a característica de monitoração do barramento, permite que cada cache se responsabilize por manter a consistência nos blocos que, por ventura, sejam afetados por operações que trafegam no barramento (inclusive operações de DMA).

Segundo o autor, outro ponto positivo do protocolo, detectado somente durante as simulações, foi o desempenho superior em termos de utilização do barramento que a política mista de atualização da memória apresentou em relação às mais comumente adotadas *write-through* e *write-back*: "...para um cache de 2048 *bytes*, com blocos de 32 *bytes*, o tráfego médio no barramento para três programas executados no PDP-11 foi de 30,768%, 17,55% e 17,38% para as políticas *write-through*, *write-back* e *write-once* respectivamente". Essa constatação, contudo, contraria as expectativas, já que, o protocolo exige uma operação de escrita inicial, mesmo quando um bloco não está sendo realmente compartilhado — tal necessidade representa uma utilização extra do barramento, quando mais de uma escrita é feita no bloco.

EXTENSÕES

A partir do protocolo *write-once* surgiram diversas extensões visando, principalmente, melhorar sua eficiência em pontos específicos. Rudolph e Segall [RuSe84] procuram aperfeiçoar a atuação do *write-once* em operações de sincronização (acesso a "trancas" (*locks*) em instruções tipo *test-and-set*, por exemplo) e comunicação (utilização de uma posição comum de memória para troca de informações) — operações bastante comuns em ambientes multiprocessados de

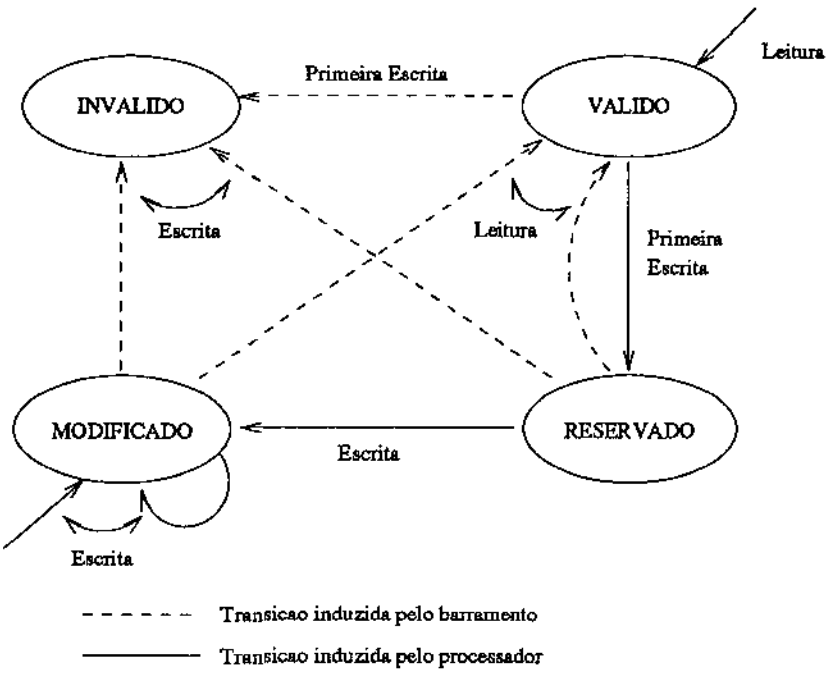

Figura 2.3: Diagrama de estados de um bloco no cache segundo o algoritmo *write–once*.

memória compartilhada. Sua proposta também explora a capacidade de transmissão do barramento, mas, dessa vez, para distribuir cópias atualizadas dos blocos para os caches com cópias antigas ou inválidas, como efeito colateral das operações de leitura e escrita. Dois esquemas são apresentados: 1. difusão nas leituras e 2. difusão nas leituras / escritas. No primeiro esquema, a primeira operação de escrita em um cache i, por exemplo, irá atualizar a memória e invalidar as cópias nos outros caches como no *write–once* original; futuras operações de escrita no cache i continuam modificando apenas sua cópia local. Uma operação de leitura desse bloco no cache j ($j \neq i$), contudo, é recebida por todos os caches que, ao perceberem a operação no barramento, atualizam suas cópias inválidas do bloco (quando existirem). No segundo esquema, além de perceber e captar os dados durante operações de leitura, um procedimento semelhante é adotado no caso de escritas: por ocasião da atualização da memória causada pela primeira escrita em um bloco, todos os caches que possuem cópias também efetuam a atualização de suas cópias locais; nessa hora, portanto, nenhuma invalidação é registrada. Uma futura operação de escrita, estando o bloco no estado *reservado*, força o envio de um sinal de invalidação que irá atuar sobre todas as outras cópias antes que a escrita possa se completar.

Apesar dessa extensão parecer atender bem a operações de sincronização e comunicação via a memória compartilhada, uma questão a ser respondida seria em que nível a eficiência do cache no atendimento ao processador é afetada, já que, mesmo com a duplicação dos diretórios

no cache, existem pontos de exclusão mútua (atualização dos *bits* de estados do bloco, por exemplo) que passam a ser muito mais disputados nessa proposta.

Eggers e Katz [EgKa89] avaliam o esquema de difusão nas leituras a partir da observação do número de sinais de invalidação e número total de ciclos de execução de aplicações com comportamento de acesso à memória distintos, quando se aumenta o tamanho dos blocos ou do cache. Comparações feitas com os dados colhidos das execuções nas mesmas aplicações usando um protocolo de invalidação simples (sem difusão nas leituras) demonstraram que a proporção de ausências por invalidação no total de ausências registradas foi menor no esquema de difusão na leitura, causando uma redução da utilização do barramento da ordem de 22% em alguns casos. Por outro lado, foi constatado um aumento real da interferência nos acessos do processador ao cache devido às operações de atualização dos blocos do esquema difusão nas leituras; em algumas aplicações esse aumento chegou a ser 3 vezes maior que na invalidação simples. Essa interferência, entretanto, causou um aumento de apenas 5,8% em média no número total de ciclos de execução das aplicações.

2.2.2 Protocolo Illinois

O protocolo de Illinois [PaPa84] baseia-se na constatação de que os sinais de invalidação são responsáveis por uma parte considerável do tráfego no barramento e pelas interferências registradas no acesso aos caches pelos processadores. É fácil verificar, contudo, que, em ao menos uma situação comum, o envio de sinais de invalidação é desnecessário: quando o bloco não está sendo compartilhado com nenhum outro cache.

A utilização de diretórios duplicados nos caches e de recursos oferecidos pelo protocolo do barramento para indicação do uso compartilhado ou não dos blocos pelos caches,[7] formam o conjunto de elementos básicos para determinar, no atendimento a uma ausência, se o bloco será usado de maneira exclusiva ou não. Dessa forma, é possível prever a necessidade do envio de sinais de invalidação em operações de escrita. O tráfego no barramento fica, portanto, restrito ao atendimento a ausências nos caches, à atualização da memória e a sinais indispensáveis de invalidação.

[7]No FutureBus [SwSm86] por exemplo, existe uma linha específica (*CH - Cache Hit*) que indica, durante um ciclo de endereçamento, se outro cache contém o bloco endereçado em seu diretório.

ESTADOS

Associado a cada bloco no cache existem 2 *bits* que definem um dos 4 estados para o bloco:

E.1 *Inválido*(I): a informação contida no bloco não está consistente com a memória.

E.2 *Exclusivo*(E): não existem cópias dos blocos em outros caches; a informação contida no bloco está coerente com a memória.

E.3 *Compartilhado*(C): ao menos dois caches possuem cópias do bloco; a informação no bloco está consistente com a memória.

E.4 *Modificado*(M): não existem cópias atuais do bloco em outros caches; o bloco foi modificado localmente e, portanto, está inconsistente com a memória.

A política de atualização adotada é a *write–back*. Por razões de simplicidade de implementação, um bloco no estado *compartilhado* não muda de estado quando, devido a operações de substituição nos outros caches, ele passa a ser a única cópia em uso — situação na qual, idealmente, o bloco deveria assumir o estado *exclusivo*.

ALGORITMO ILLINOIS

LEITURA

Não–Referenciado $[V_m \quad A_c^n]$

Inválido ou *Ausente* – a memória supre uma cópia do bloco que assume o estado local *exclusivo*.

Intacto–Privado $[V_m \quad A_c^{n-i-1} \quad I_c^i \quad E_c \quad (0 \leq i \leq n-1)]$
Intacto–Compartilhado $[V_m \quad A_c^{n-i-j} \quad I_c^i \quad C_c^j \quad (0 \leq i \leq n-j \text{ e } 2 \leq j \leq n)]$

Inválido ou *Ausente* – a memória é inibida e o pedido é atendido pelo cache que possui a cópia; caso exista mais de uma cópia, um mecanismo de prioridade deve assegurar que apenas um dos caches, o de maior prioridade por exemplo, irá suprir o bloco. Se o estado do bloco no cache que o supriu for *exclusivo*, ele é mudado para *compartilhado*.

Exclusivo ou Compartilhado – a leitura é atendida localmente.

Modificado $[I_m \quad A_c^{n-i-1} \quad I_c^i \quad M_c \quad (0 \leq i \leq n-1)]$

Inválido ou *Ausente* – a memória é inibida e o cache com a cópia modificada supre o bloco ao mesmo tempo que atualiza a memória e assume o estado *compartilhado*; o bloco no cache requisitante também assume o estado *compartilhado*.

Modificado – a leitura é atendida localmente.

ESCRITA

Não–Referenciado $[V_m \quad A_c^n]$

Inválido ou *Ausente* – a memória supre o bloco que assume o estado *modificado* no cache requisitante.

Intacto–Privado $[V_m \quad A_c^{n-i-1} \quad I_c^i \quad E_c \quad (0 \leq i \leq n-1)]$
Intacto–Compartilhado $[V_m \quad A_c^{n-i-j} \quad I_c^i \quad C_c^j \quad (0 \leq i \leq n-j \text{ e } 2 \leq j \leq n)]$

Exclusivo – a escrita é feita localmente; não é necessário enviar nenhum sinal de invalidação pelo barramento.

Compartilhado – a escrita é feita localmente e um sinal de invalidação do bloco é transmitido pelo barramento.

Inválido ou *Ausente* – um pedido de cópia é enviado pelo barramento e é atendido por um dos outros caches; como se trata de uma escrita, todos os caches que compartilham o bloco, ao perceberem a operação, invalidam suas cópias.

Modificado $[I_m \quad A_c^{n-i-1} \quad I_c^i \quad M_c \quad (0 \leq i \leq n-1)]$

Inválido ou *Ausente* – o bloco é trazido e assume o estado *modificado*; o cache que o supriu, invalida sua cópia.

Modificado – a escrita é efetuada imediatamente na cópia local.

As entrada neste algoritmo correspondem à operação efetuada pelo processador (**LEITURA** ou **ESCRITA**), à classe ou estado do bloco no sistema (**Não-Referenciado**, **Intacto-Privado**, etc.) e ao estado local do bloco (*Inválido*, *Válido*, etc.) respectivamente. A Figura 2.4 apresenta o diagrama de estados que de um bloco no cache para o algoritmo Illinois

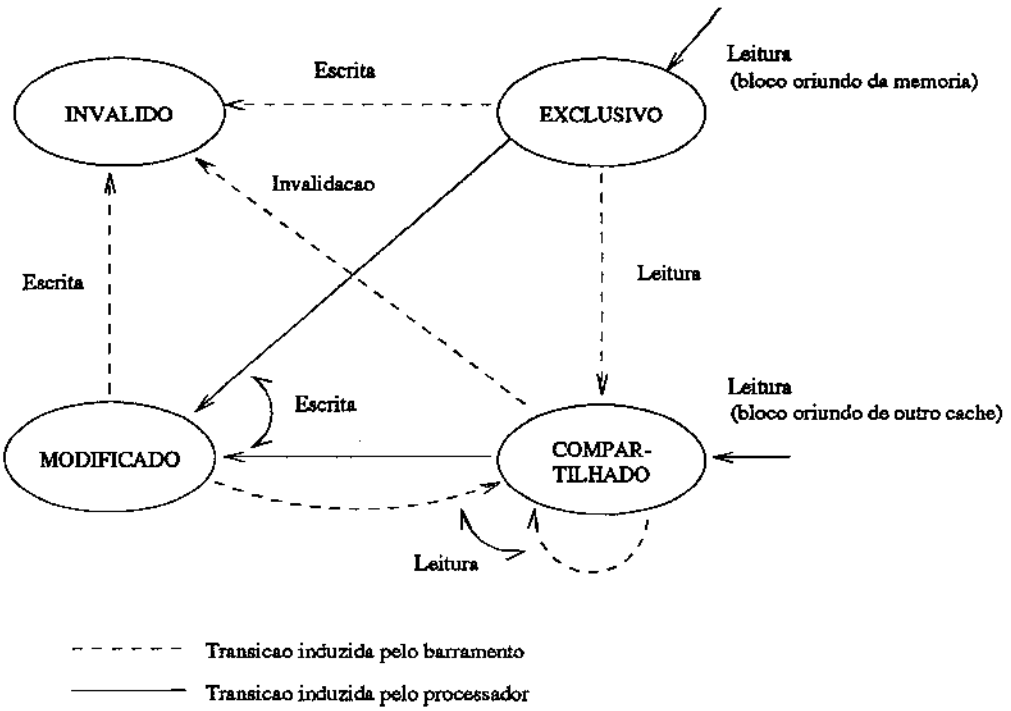

Figura 2.4: Diagrama de estados de um bloco no cache segundo o algoritmo Illinois.

AVALIAÇÃO

A utilização de sinais de invalidação pode ser vista, de uma maneira geral, como um *overhead* inevitável nos protocolos que utilizam essa técnica. Ela influi diretamente no desempenho do sistema, já que, se torna uma componente de peso quando colocada ao lado das operações que competem pelo barramento. A eliminação de operações de invalidação desnecessárias e conseqüente redução do número dessas operações sem a inclusão de outros *overheads*, proposta por esse algoritmo, beneficia diretamente alguns tipos de aplicações; dentre elas, as principais são as que apresentam uma baixa taxa de compartilhamento dos dados entre os processadores, justificando a utilidade do estado *exclusivo*.

2.2.3 Protocolo Berkeley

Katz [Katz85] descreve um protocolo projetado para ser implementado em um único *chip* VLSI constituído de área de armazenamento e controlador de cache (responsáveis pelas funções normais de um cache) além de unidade monitora do barramento (responsável, juntamente com a unidade controladora, pela manutenção da consistência). Na concepção desse *chip* foram considerados fatores como a criação de um número mínimo de comandos de consistência, de modo a

tornar o compartilhamento de dados tão barato quanto possível; além disso, o protocolo deveria ser facilmente adaptável a sistemas de memória e barramento comercialmente disponíveis na época — um dos protocolos de barramento que satisfaziam os requisitos e que, de fato, fora usado no protótipo foi o Multibus da Intel.

Basicamente, a estratégia proposta para que um grande número de processadores pudessem ser conectados a um mesmo barramento, consiste em reduzir o tráfego nesse barramento através do adiamento da atualização da memória até o momento em que um bloco modificado sofra substituição. O suprimento da versão mais atualizada de um bloco é gerenciada pelo próprio sistema de caches: o cache que modificou o bloco por último é o responsável por suprir cópias do bloco aos caches requisitantes. Esse cache é conhecido como o *dono* do bloco e, portanto, tem a sua *posse* — um conceito introduzido em [Fran84]. Como nos demais protocolos, cópias de um bloco podem residir em mais de um cache; no máximo, porém, um cache *possui* o bloco e tem o direito de atualizá–lo. Em contrapartida, o dono de um bloco é obrigado a atender às requisições de bloco, feitas por outros caches e a atualizar a memória em caso de substituição do bloco.

De modo geral, todos os blocos são ditos possuídos unicamente ou por um, e apenas um, cache ou pela memória, esta última sendo considerada o dono *default*.

ESTADOS

Os estados associados a um bloco no protocolo Berkeley são usados basicamente, para informar qual a responsabilidade do cache sobre o bloco (se o cache tem a posse ou não) e qual o tipo de utilização do bloco em relação aos outros caches (se usado exclusivamente ou existe compartilhamento). Blocos armazenados na memória não possuem estados associados, a não ser os implícitos *válido* e *inválido*.

Os 4 estados possíveis são:

E.1 *Inválido*(I): a informação contida no bloco não está consistente com a memória.

E.2 *Público*(P): o bloco contém informação consistente mas sua posse pertence a outro cache ou à memória.

E.3 *Possuído–Compartilhado*(PC): o cache detém a posse mas podem existir cópias nos outros caches do sistema.

E.4 *Possuído*(PP): o cache detém a posse do bloco e, seguramente, nenhum outro cache possui uma cópia do bloco.

Nenhuma modificação local no bloco é permitida, sem que antes o cache obtenha a posse do bloco. Caso a posse não pertença a nenhum cache, ela implicitamente estará associada à memória. Um bloco no estado *possuído–compartilhado* pode ter cópias no estado *público* em outros caches e, portanto, apesar de outras modificações poderem ser feitas localmente, as outras cópias devem ser invalidadas. Blocos no estado *possuído* são alterados localmente sem a necessidade do envio de sinais de invalidação.

CLASSES

No protocolo Berkeley, o fato de um bloco assumir o estado *possuído* em um cache está associado diretamente com a capacidade atribuída ao cache de modificar sua cópia — além dos já abordados deveres de suprir cópias para os outros caches e atualizar a memória em caso de substituição. Portanto, é razoável supor que uma requisição de posse implique diretamente em previsão de alteração no cache requisitante. Conseqüentemente, a substituição das classe **modificado–compartilhado** e **modificado–privado** usados nos algoritmos anteriores apresentado na seção anterior, por **possuído–compartilhado** e **possuído–privado** respectivamente, por razões de clareza, torna–se conveniente. Essa substituição será considerada na apresentação do algoritmo a seguir.

ALGORITMO BERKELEY

LEITURA

Não–Referenciado $[V_m \quad A_c^n]$

Inválido ou *Ausente* – a memória tem a posse e, portanto, supre o bloco que assume o estado *público*.

Intacto–Privado $[V_m \quad A_c^{n-i-1} \quad I_c^i \quad P_c \quad (0 \leq i \leq n-1)]$
Intacto–Compartilhado $[V_m \quad A_c^{n-i-j} \quad I_c^i \quad P_c^j \quad (0 \leq i \leq n-j \text{ e } 2 \leq j \leq n)]$

Público – a leitura é atendida localmente.

Inválido ou *Ausente* – a memória supre o bloco que assume o estado *Público.*

Possuído–Compartilhado $[I_m \quad A_c^{n-i-j-1} \quad I_c^i \; P_c^j \; PC_c \quad (0 \leq i \leq n-j-1 \text{ e } 2 \leq j \leq n-1)]$

Inválido ou *Ausente* – o cache que tem a posse detecta e inibe o pedido suprindo, em seguida, uma cópia do bloco que assume o estado *público* no cache requisitante.

Público ou *Possuído–Compartilhado* – a leitura é atendida localmente.

Possuído–Privado $[I_m \quad A_c^{n-i-1} \quad I_c^i \quad PP_c \quad (0 \leq i \leq n-1)]$

Inválido ou *Ausente* – o cache que tem a posse detecta e inibe o pedido suprindo, em seguida, uma cópia do bloco que assume o estado *público* no cache requisitante; o estado da cópia no cache dono é atualizado para *possuído–compartilhado.*

Possuído – a leitura é atendida localmente.

ESCRITA

Não–Referenciado $[V_m \quad A_c^n]$

Inválido ou *Ausente* – a memória supre o bloco que assume o estado *possuído* no cache requisitante.

Intacto–Privado $[V_m \quad A_c^{n-i-1} \quad I_c^i \quad P_c \quad (0 \leq i \leq n-1)]$

Intacto–Compartilhado $[V_m \quad A_c^{n-i-j} \quad I_c^i \quad P_c^j \quad (0 \leq i \leq n-j \text{ e } 2 \leq j \leq n)]$

Inválido ou *Ausente* – um pedido de leitura do bloco e sua posse é transmitida pelo barramento e atendida pela memória. Ao detectarem o pedido no barramento, os outros caches invalidam suas cópias. O bloco assume o estado *possuído* no cache requisitante.

Público – um comando de invalidação é enviado para os outros caches. A cópia local assume o estado *possuído.*

Possuído–Compartilhado $[I_m \quad A_c^{n-i-j-1} \quad I_c^i \; P_c^j \; PC_c \quad (0 \leq i \leq n-j-1 \text{ e } 2 \leq j \leq n-1)]$

Inválido ou *Ausente* – um pedido de leitura do bloco e sua posse é transmitido pelo barramento; o dono do bloco, ao detectar o pedido,

inibe a memória e supre uma cópia que assume o estado *possuído* no cache requisitante; ao mesmo tempo, o antigo dono e todos os outros caches, ao perceberem a operação, invalidam suas cópias.

Público ou *Possuído–Compartilhado* – um comando de invalidação é enviado pelo barramento para que todos os outros caches invalidem suas cópias; o bloco assume o estado local *possuído*.

Possuído–Privado $[I_m \quad A_c^{n-i-1} \quad I_c^i \quad PP_c \quad (0 \leq i \leq n-1)]$

Inválido ou *Ausente* – um pedido de leitura do bloco e sua posse é transmitido pelo barramento; o dono atual detecta o pedido e inibe a memória, suprindo ele mesmo uma cópia e mudando o estado de sua cópia para *inválido*; no cache requisitante, o bloco assume o estado *possuído*.

Possuído – a escrita é feita localmente, dispensando outras operações.

As entrada neste algoritmo correspondem à operação efetuada pelo processador (**LEITURA** ou **ESCRITA**), à classe ou estado do bloco no sistema (**Não-Referenciado**, **Intacto-Privado**, etc.) e ao estado local do bloco (*Inválido*, *Válido*, etc.) respectivamente. A Figura 2.5 apresenta o diagrama de estados que de um bloco no cache para o algoritmo Berkeley

AVALIAÇÃO

Existem dois fatores que tornam o protocolo Berkeley eficiente com relação à utilização do barramento, quando se considera o compartilhamento de blocos que sofreram modificações: 1. o dono de um bloco é o responsável pelo atendimento às requisições de cópias — uma operação que envolve ciclos de barramento (mais rápidos que os ciclos de memória); 2. a atualização da memória é retardada até o momento da substituição do bloco no cache dono.

Outra característica que torna o protocolo atraente do ponto de vista do desempenho é a indicação, no instante da leitura,[8] da intenção de um processador de modificar um bloco. Esse mecanismo torna desnecessário o envio posterior, como é o caso dos protocolos anteriores, de sinais de invalidação reduzindo, dessa forma, o tráfego no barramento.

[8]Um dos comandos de consistência emitidos pelo cache é o de *leitura com posse*: além de receber uma cópia do bloco, o cache que emite esse comando torna-se o responsável (dono) pelo bloco com, direito de modificação.

OUTROS TRABALHOS

O conceito de posse já havia sido empregado anteriormente no Synapse N+1, um sistema multiprocessado projetado para o processamento de transações tolerante a falhas [Fran84]. No Synapse, dois barramentos interconectam todos os dispositivos do sistema e garantem o maior desempenho possível, com o sistema podendo ser expandido para até 28 processadores e com poder de processamento crescendo linearmente. O protocolo adotado no Synapse apresenta uma diferença básica em relação ao Berkeley: não existe a classe **possuído–compartilhado**. Portanto, para que um bloco no estado *possuído* possa se tornar compartilhado, a cópia na memória deve antes ser atualizada, assumindo a posse do bloco. O cache que fez a requisição do bloco tem seu pedido inibido e deve refazê-lo para que, dessa vez, a cópia seja suprida pela memória — o novo dono do bloco. Transferências de bloco com posse, contudo, são feitas diretamente cache a cache; nesse caso, a memória não intervém na transação, pois um *bit* associado a cada bloco na própria memória indica que ela não é a dona do bloco e que sua cópia está desatualizada.

Resumindo, a única transação em que a memória não participa, é quando ocorre a transferência do bloco com posse de um cache a outro. Em todos os outros casos, ela participa de forma simples — suprindo cópias compartilhadas, o que requer um ciclo de memória — ou impondo um *overhead* maior — quando sofre uma escrita seguida de leitura na transformação de um bloco modificado para compartilhado, requerendo dois ciclos de memória. Essa política, sem dúvida, torna o protocolo Synapse menos eficiente em relação ao Berkeley em termos de utilização do barramento.

2.3 Avaliação Comparativa

Archibald e Baer [ArBa86a] examinaram a eficiência de diversos protocolos de coerência para barramento, entre eles os três descritos neste trabalho, usando um modelo próprio de simulação como base para uma avaliação comparativa dos protocolos. Basicamente, esse modelo[9] buscava obter medidas quantitativas do desempenho dos protocolos a partir da observação dos resultados

[9] O modelo foi implementado em Simula, e consistia na execução de diversos tipos de processos, cada um com funções específicas: cada processador era representado por um processo; da mesma forma, a cada cache e ao barramento correspondiam um processo. Cada fase do experimento correspondia à coleta de resultados durante a execução de vários passos — a cada passo um novo processador (até um total de 15) era introduzido no sistema.

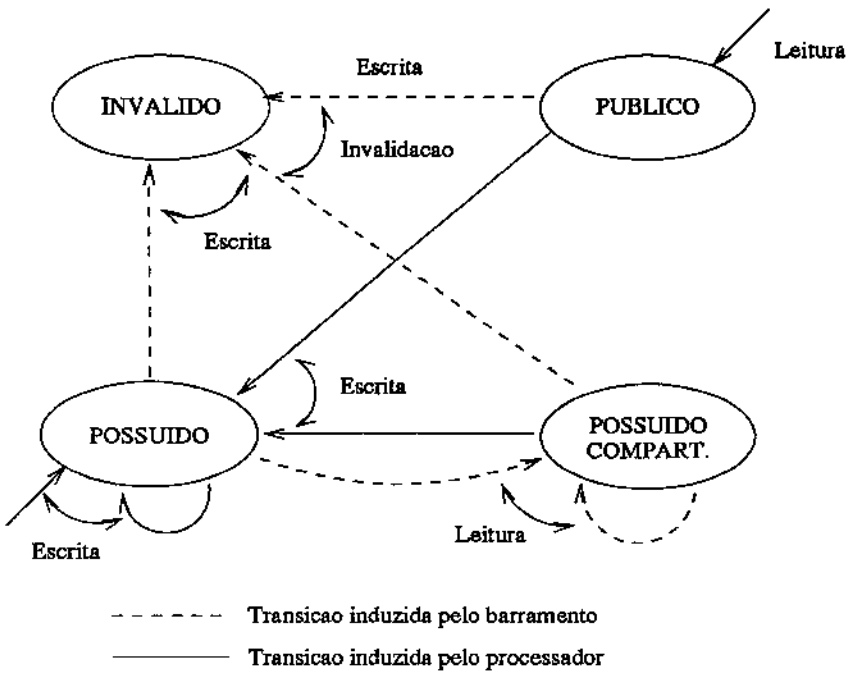

Figura 2.5: Diagrama de estados de um bloco no cache segundo o algoritmo Berkeley.

coletados em diversas fases do experimento. Cada fase correspondia à execução do modelo com a variação de parâmetros tais como: grau de compartilhamento dos blocos, taxa de operações de escrita e taxa de acerto nos caches, considerando o aumento do seu tamanho. Outro ponto que se pretendia determinar, era qual seria o número máximo de processadores que poderiam compartilhar o barramento antes de se caracterizar a *saturação* do barramento (estado em que a contenção no barramento o torna elemento limitante do desempenho do sistema).

Como o objetivo do projeto era avaliar os protocolos entre si e não detalhes de implementação, admitiu-se, na definição do modelo, uma configuração de sistema idêntica para todos os protocolos: um único barramento compartilhado e duas cópias de diretório no cache.[10] Outros parâmetros como número de ciclos da memória, número de ciclos do cache, etc, foram considerados também idênticos para todos os protocolos. Os resultados do experimento estão sumarizados graficamente na Figura 2.6 e textualmente a seguir; esses resultados ratificam as avaliações particulares dos protocolos feitas anteriormente neste capítulo. Nos gráficos da Figura 2.6, o eixo vertical representa o poder de processamento do sistema, obtido multiplicando-se a soma da utilização dos processadores por 100, enquanto que o eixo horizontal indica a quantidade de processadores do sistema no ponto de cada medida; para efeito de comparação,

[10]Essa configuração era compatível com a implementação dos três protocolos apresentados neste capítulo, apesar de não o ser para todos os protocolos (um total de 6) sob avaliação no experimento.

os gráficos também apresentam os resultados obtidos com o protocolo *write-through* puro.

Em uma primeira fase, foi observado o comportamento dos protocolos quando operando quase que exclusivamente com blocos privados (Figura 2.6a). Nesse caso, como era esperado, os protocolos apresentaram comportamento idêntico no tratamento de todas as operações comuns no cache, exceto quando se tratava de alteração de um bloco local não modificado ou durante a execução de substituições. Apenas o protocolo Illinois é capaz de detectar dinamicamente o uso privado do bloco e, com isso, evitar o envio de qualquer sinal de invalidação pelo barramento quando ocorre a primeira modificação em um bloco. Berkeley por outro lado, requer a utilização de um ciclo do barramento para o envio de um sinal de invalidação enquanto que no *write-once* o barramento é usado para o envio de uma palavra que irá atualizar a memória. As diferenças registradas em operações de substituição são conseqüência do fato de, no *write-once*, a probabilidade de um bloco modificado necessitar ser enviado à memória é reduzida: nesse protocolo, os blocos modificados uma única vez (em estado *reservado*) estão consistentes com a memória e, portanto, não ocupam o barramento durante uma substituição. Como resultado, observou-se que os três protocolos possuem comportamento idêntico, com pequena vantagem para o Illinois, seguido de perto por Berkeley. Quanto ao *Write-once*, se for considerado que 33% das modificações ocorrem uma única vez no bloco (uma porcentagem razoável), seu desempenho se equipara com o Illinois — considerando apenas 5%, contudo, seu desempenho degrada bastante em relação aos outros protocolos (na Figura 2.6a está registrado apenas esse último desempenho).

Em outra fase do experimento, procurou-se avaliar o desempenho dos protocolos quando se verifica um aumento no número de blocos compartilhados em relação ao experimento com blocos privados — aumentou-se também a taxa média de compartilhamento real (Figura 2.6b).

Em se tratando de blocos compartilhados, a única operação em que os protocolos apresentaram resultados similares foi na leitura de blocos locais. No atendimento às ausências durante leituras ou escritas, os blocos são enviados por outros caches — no caso de Illinois ou Berkeley —, e sempre da memória — no caso do *Write-once* (com um custo maior em termos de utilização do barramento). Escritas a blocos locais não modificados requerem operações que vão desde a atualização da memória até o envio de sinais de invalidação. Os resultados do experimento com blocos compartilhados mostraram que Berkeley, apesar de menos eficiente com blocos privados, apresenta um desempenho total melhor que o Illinois quando a taxa de compartilhamento é alta. Dois fatores podem se apresentar como responsáveis por esse comportamento: 1. O tratamento da ausência de um bloco que foi modificado em um outro cache não requer, no protocolo Berkeley, que a memória seja atualizada, como é o caso do Illinois;

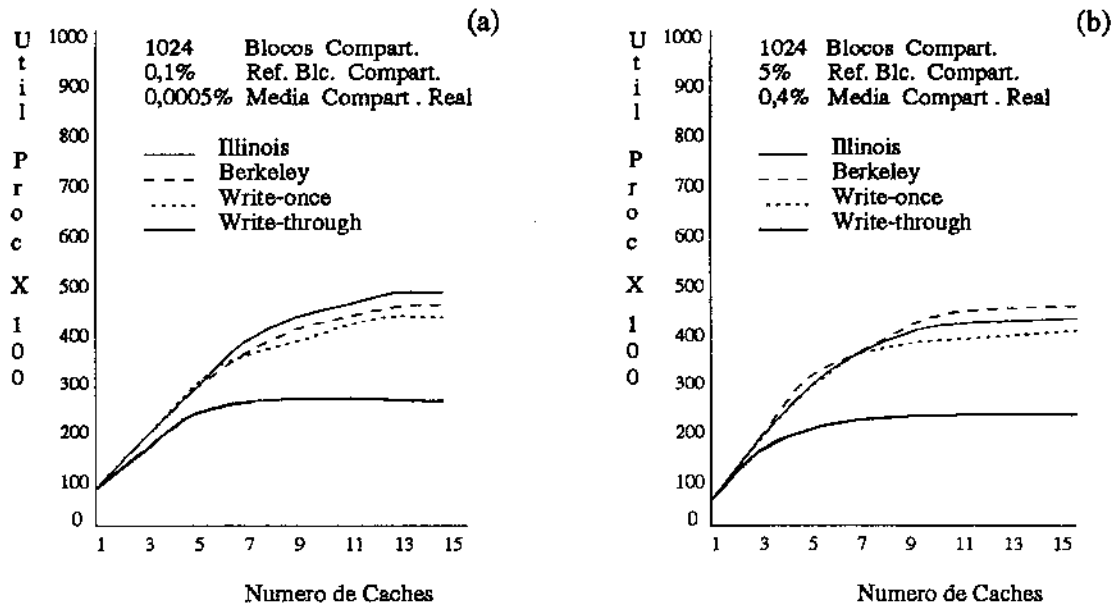

Figura 2.6: Resultados dos experimentos de Archibald e Baer com barramentos (*ACM TOCS*, Vol. 4, Num. 4).

2. Em situações onde a taxa de compartilhamento é bastante elevada é possível obter uma compensação do *overhead* dos sinais de invalidação desnecessários de Berkeley sobre o *overhead* de busca de blocos atualizados na memória de Illinois.

O desempenho do protocolo *Write–once* com blocos compartilhados foi menor que o dos outros protocolos como resultado do *overhead* imposto pela atualização da memória sempre que um bloco modificado mais de uma vez em um cache, era requisitado por outro cache.

Yang [Yang89] desenvolveu um modelo analítico para a avaliação dos mesmos protocolos experimentados por Archibald e Baer [ArBa86a]. Observou-se uma consistência total nos resultados obtidos nos dois tipos de análise. [11]

2.4 Outras Soluções

Uma característica comum aos protocolos analisados na seção anterior, é a adoção do paradigma um escritor/vários leitores: as modificações só podem ser executadas por apenas um dos caches, explicitamente definido em um dado instante. Como conseqüência desse paradigma,

[11] As poucas diferenças encontradas foram atribuídas ao tipo de barramento adotado nos dois experimentos: Archibald usou um barramento tipo *circuit–switched* enquanto que Yang adotou o tipo *packet–switched*.

os protocolos recorrem a algum tipo de invalidação dos blocos nos caches leitores antes que a modificação possa ser concluída. Protocolos que adotam essa filosofia são conhecidos como protocolos *write-invalidate*. Outras soluções, como as adotadas no Firefly, uma estação de trabalho multiprocessada desenvolvida pela Digital e o sistema Dragon da Xerox, empregam uma abordagem diferente: a vários caches é permitido alterar um mesmo bloco simultaneamente. Para que isso seja possível, a palavra alterada em operações de escrita em blocos compartilhados é também enviada aos outros caches para que eles atualizem as suas respectivas cópias. Protocolos nessa categoria são classificados como *write-broadcast*.

A idéia básica usada no Firefly [Thac88] é a de que um cache é capaz de detectar quando um outro cache compartilha uma posição particular de memória. Para tanto, o protocolo recorre ao mesmo recurso de barramento usado no Illinois: uma linha que indica, no instante do endereçamento, se o bloco está sendo compartilhado ou não. Não existem sinais de invalidação. Durante a inicialização do sistema, todas as posições dos caches são carregadas com dados da própria memória de modo a manter-se a consistência desde o início. Os pedidos de leitura são atendidos pelos próprios caches caso um (ou mais de um deles) possua o bloco requisitado. Operações de escrita em blocos compartilhados atualizam os outros caches e a memória simultaneamente. O protocolo implementado no Dragon é semelhante exceto pelo fato de a memória não ser atualizada juntamente com os outros caches nos casos de escrita a blocos compartilhados.

A principal vantagem de protocolos como esses, é a utilização da política *write-through* apenas no suporte à coerência de blocos compartilhados. Assim que o compartilhamento cessa, isso é detectado através da linha de compartilhamento e a política *write-through* é abandonada. Entretanto, essa estratégia não é imune ao mesmo problema enfrentado pelos protocolos *write-invalidate*: o *falso compartilhamento* (fenômeno segundo o qual um bloco permanece no cache, ocasionando o envio de sinais de compartilhamento (ou invalidação), mesmo quando o bloco não é mais útil). O falso compartilhamento pode ser apontado como um dos responsáveis pelo *overhead* em sistemas que permitem a migração de processos, por exemplo.

Apesar da utilização da política *write-through* desencorajar qualquer expectativa de um bom desempenho nesses protocolos, resultados práticos demonstram o contrário. Juntamente com os protocolos *write-invalidate* descritos anteriormente, Archibald [ArBa86a] e Yang [Yang89] também avaliaram a eficiência de protocolos *write-broadcast* como o Firefly e o Dragon. Nos dois experimentos, esses últimos apresentaram desempenho igual ou superior em relação aos primeiros. Como conclusão imediata, observa-se que o nível de ausências em conseqüência das invalidações e o "efeito cascata" decorrente (invalidações forçando ausências que

por sua vez geram novas invalidações) impõem um preço alto aos protocolos *write-invalidate* — nos experimentos com baixo nível de competição por blocos compartilhados, as diferenças registradas nos dois tipos de protocolos foram desprezíveis.

2.5 Observações Finais

A utilização prática dos protocolos direcionados a redes tipo barramento pode ser sentida pela quantidade razoável de sistemas comercialmente disponíveis que os empregam. Entre eles, a série Simmetry da Sequent, composta de sistemas que variam de 10 a 30 CPU's (Intel 80386), implementa um protocolo tipo *write-invalidate* com política *copy-back* de atualização da memória, sendo os blocos modificados supridos pelo cache com a versão mais atualizada; já o sistema Multimax da Encore, baseado em CPU's da National Semiconductor (NS32332 e NS32532), adota o protocolo Berkeley na manutenção da coerência de seus caches [Taba90].

Atualmente, pode-se afirmar que os conceitos envolvidos e as formas de implementação dos protocolos tipo *snoop* estão devidamente estabelecidos e solidificados. Um exemplo disso é o surgimento de uma nomenclatura que vem sendo comumente usada pela literatura técnica da área para referenciar essa classe de protocolos. Tal nomenclatura adota as iniciais dos estados que um bloco de cache pode assumir em cada protocolo para nomear os diversos tipos de protocolos. Assim, a sigla **MESI** (de **M***odified*, **E***xclusive*, **S***hared* e **I***nvalid*) é usada para indicar um protocolo tipo Illinois, **MOSI** (de **M***odified*, **O***wned*, **S***hared* e **I***nvalid*) é usado para protocolos tipo Berkeley, etc.

Os esforços de pesquisa buscam agora novas formas para a manutenção da coerência em sistemas cuja rede de interconexão, diferentemente dos barramentos, não permite ou inviabiliza a monitoração dos sinais que nela trafegam para detecção e tratamento daqueles que afetam a consistência das informações nos caches. Os resultados mais relevantes, divulgados até recentemente, nessa área serão apresentados e discutidos no próximo capítulo.

Capítulo 3

Protocolos direcionados a redes mais genéricas

O emprego das soluções discutidas no capítulo 2 só é viável em sistemas com um número reduzido de processadores (20 – 30). Isso se deve inteiramente às características do barramento que, sendo uma via única de comunicação, impõe a serialização em sua utilização e, conseqüentemente, o perigo potencial de tornar-se um ponto de "gargalo" com a expansão do sistema. Em sistemas com centenas de processadores, outros tipos de rede de interconexão são empregados. Eles buscam a redução da latência da comunicação (nos acessos à memória por exemplo), através do fornecimento de diversas vias que podem ser usadas simultaneamente, e têm a capacidade de expansão juntamente com o sistema. A existência de diversas vias, contudo, torna inviável a exploração da difusão (*broadcasting*) com detecção por monitoramento como forma de envio dos comandos de consistência; a conseqüência direta é a necessidade do emprego de estruturas que indiquem aos protocolos de coerência de que forma as cópias dos blocos estão distribuídas pelos caches do sistema, de modo que, os comandos de consistência possam ser enviados, diretamente, através das ligações ponto a ponto providas pela rede. Tais estruturas são conhecidas como diretórios [1] [Tang76], [CeFe78], [ArBa84].

Neste capítulo serão discutidas algumas propostas de soluções usando diretórios, suas características, vantagens e desvantagens, e como o desempenho global do sistema é afetado por cada uma delas. Diferentemente do capítulo anterior, a ênfase a ser dada aqui diz res-

[1] No decorrer desse capítulo, salvo quando explicitamente indicado, o termo *diretório* será usado para referenciar essas estruturas e não os diretórios existentes nos caches que são usados na identificação dos blocos que estão disponíveis localmente.

peito à organização do diretório e não aos protocolos em si. Isso por que, definida a estrutura, o desenvolvimento dos protocolos consiste, de certa forma, na adaptação dessa estrutura às soluções vistas no capítulo anterior. Antes de apresentar os esquemas propostos que empregam diretórios, porém, serão rapidamente discutidas as características de algumas redes de interconexão que têm sido usadas, acadêmica e comercialmente, na implementação de grandes sistemas.

3.1 Redes de Interconexão

Em sistemas multiprocessados, a rede de interconexão é o elemento que torna possível a comunicação entre os diversos dispositivos que compõem o sistema (processadores, módulos de memória, dispositivos de E/S, etc.)[2]. Como o desempenho de qualquer sistema de computação está associado, entre outros fatores, à eficiência com que essa comunicação é feita, constantes esforços de pesquisa têm sido despendidos no desenvolvimento de redes eficientes e de custo relativamente baixo. A seguir serão apresentados e discutidos alguns modelos de redes de interconexão que têm sido propostos para sistemas multiprocessados.

3.1.1 Barramentos Simples e Múltiplos

O primeiro tipo de rede adotado na construção de sistemas multiprocessados foi uma evolução natural a partir do dispositivo com função semelhante usado nos sistemas monoprocessados: o barramento simples. Algumas de suas características se mostraram, inicialmente, bastante atraentes para aplicação na nascente tecnologia dos multiprocessadores de memória compartilhada. Entre elas, o baixo custo de sua implementação, a utilização de poucos dispositivos lógicos de apoio — circuitos de arbitragem, transmissão e recepção — resultando em uma baixa complexidade, a capacidade de difusão de sinais (*broadcasting*) — característica bastante explorada pelos protocolos de coerência discutidos no capítulo 2 — e, não menos importante, o fato de ser uma tecnologia bem conhecida e testada. Todas essas propriedades consagraram o barramento como a rede mais adotada em sistemas comerciais, especialmente naqueles com número reduzido de processadores, como é o caso das máquinas Balance e Simetry da Sequent e a Multimax da Encore [Taba90].

[2]Na realidade, o mesmo pode ser dito nos sistemas monoprocessados; a diferença é que, neste caso, o barramento tem sido o único dispositivo utilizado, sendo desnecessário uma designação mais genérica.

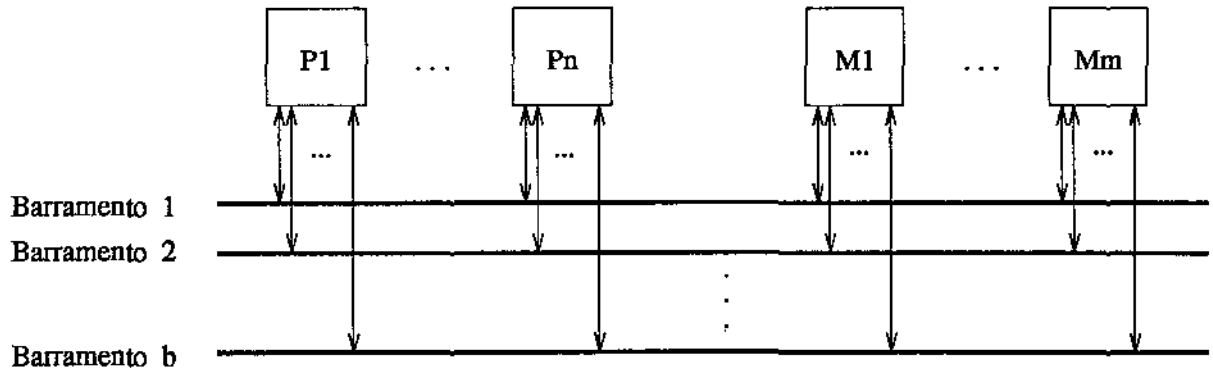

Figura 3.1: Organização multiprocessada usando rede tipo barramentos múltiplos.

Por outro lado, o uso do barramento em sistemas multiprocessados introduziu também algumas limitações. A principal delas está relacionada ao fato do barramento oferecer apenas uma única via de comunicação, compartilhada pelos diversos elementos do sistema; ou seja, apenas dois dispositivos por vez podem estabelecer comunicação através do barramento. Como, normalmente, essa comunicação está relacionada com a transferência de dados entre a memória e os processadores, a ocorrência de disputas pelo controle do barramento, independentemente da velocidade com que ele opera, implica diretamente na redução da taxa de execução (*throughput*) de todo o sistema (enquanto o processador está tentando obter o acesso ao barramento, ele não está executando trabalho útil). Esse é o principal motivo pelo qual o barramento não é o tipo de rede indicado para sistemas com um número muito grande de processadores ou projetados para sofrerem expansões. Além disso, a ocorrência de falhas em um dos módulos do sistema (processador, módulo de memória, etc.) não acarreta o impedimento de operação da máquina, ao passo que uma falha no sistema de barramento pode ser catastrófica nos ambientes que utilizam o barramento simples.

Uma extensão imediata do barramento simples, na tentativa de aliviar o problema da contenção, é a utilização de barramentos múltiplos, numa configuração semelhante à ilustrada na Figura 3.1. Apesar de reduzir a contenção e tornar o sistema como um todo mais tolerante a falhas, esse novo dispositivo impõe também algumas restrições. A multiplicação da lógica de apoio encarece a implementação, e a necessidade mecanismos de arbitragem, tanto no acesso dos processadores ao barramento quanto na interligação destes aos módulos de memória, aumentam consideravelmente a complexidade do gerenciamento. Essas, aliás, parecem ser as principais razões pelas quais os barramentos múltiplos têm sido pouco utilizados no projeto de sistemas multiprocessados [MHW87].

3.1.2 Redes tipo *Crossbar*

Um caso particular de redes tipo barramento múltiplos é a rede tipo *crossbar*. Ela é obtida através do fornecimento de barramentos exclusivos para os processadores, com cada um deles possuindo conexões para todos os módulos de memória (Figura 3.2). A principal característica de uma rede *crossbar* é a capacidade de estabelecer a comunicação concorrente entre todos os processadores e os módulos de memória, desde que o número de módulos de memória seja maior que o de processadores e que os processadores efetuem o acesso a módulos de memória distintos. Cada conexão é constituída de circuitos mutiplexadores e de arbitragem, além da lógica de controle e de vias de endereços e dados (em destaque na Figura 3.2). Como o ponto de conexão de um processador a um módulo de memória não impede a passagem de sinais de outros processadores enviados a outros módulos de memória, esse tipo de rede é dita *não bloqueante*.

A maior vantagem da rede *crossbar* é a sua capacidade de fornecer as mais altas taxas de transferência (*bandwidth*) graças às suas vias de comunicação múltiplas. Sua principal desvantagem é o custo excessivo e a lógica complexa associados a cada ponto de conexão [Taba90]. Se for suposto um número igual de processadores e módulos de memória N, o custo de uma rede *crossbar* é dado por uma função quadrática, ou seja, $\Theta(N^2)$; esse custo desencoraja a sua utilização em grandes sistemas multiprocessados. O avanço das técnicas VLSI, contudo, pode tornar esse tipo de rede um dos mais utilizados nos multiprocessadores do futuro [HwBr90]. Atualmente, as redes *crossbar* foram utilizadas, a nível acadêmico, no Carnegie–Mellow C.mmp, um sistema com 16 processadores e 16 módulos de memória, e, comercialmente, na máquina Alliant FX, na interligação dos processadores com as memórias cache que são compartilhadas, e no IP–1, da International Parallel Machines, um sistema com 8 processadores e 8 módulos de memória [Taba90].

3.1.3 Redes tipo Multiestágios

Como alternativa ao alto custo das redes *crossbar*, as chamadas redes com chaves multiestágios têm sido objeto de muita pesquisa recentemente. Da mesma forma que as redes *crossbar*, uma rede multiestágio é capaz de estabelecer diversas ligações ponto a ponto simultâneas, sendo o custo reduzido, porém, para $\Theta(N \log N)$.

Uma rede multiestágio $N \times N$ conecta N processadores a N memórias. Considerando N uma potência de m, são necessários $\log_m N$ estágios com $\frac{N}{m}$ chaves cada estágio, para a

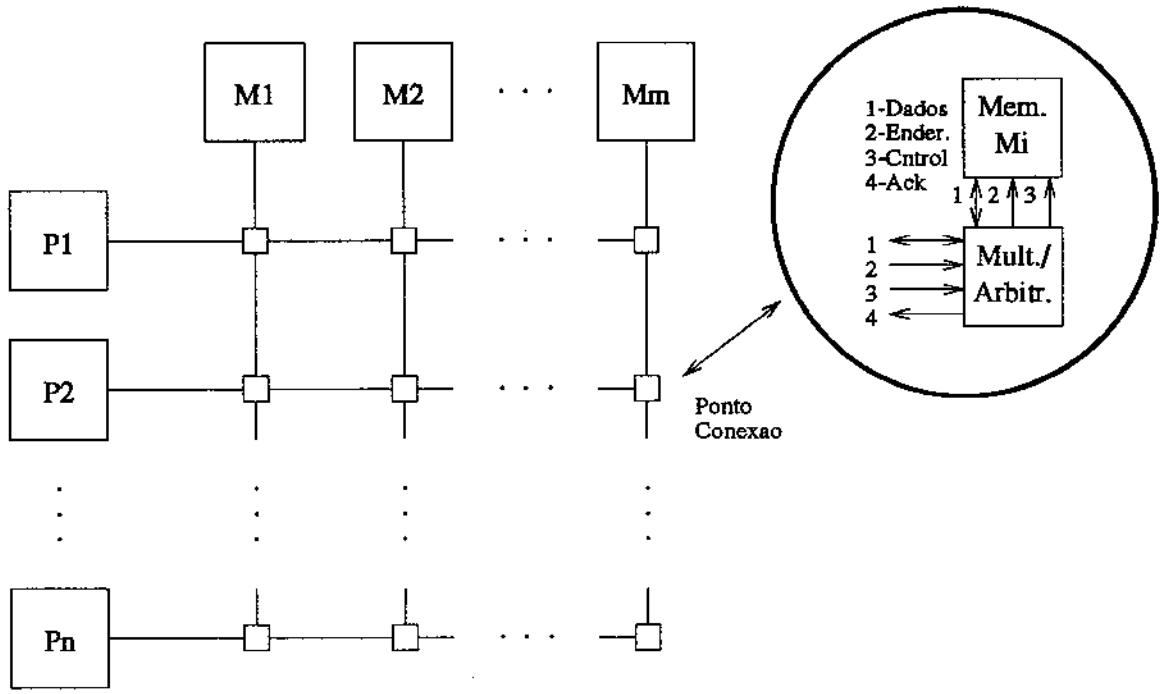

Figura 3.2: Organização multiprocessada usando rede tipo *crossbar*

formação da rede. A Figura 3.3a mostra um exemplo de rede multiestágio 8×8 com chaves 2×2 (duas entradas e duas saídas). Por cada chave passam os sinais de dados e controle que são trocados entre os dois pontos de comunicação. O roteamento dentro da chave é feito a partir de um *bit* de controle cuja forma de funcionamento é mostrada na Figura 3.3b. Uma das formas de roteamento possível é fazer com que o *bit* de controle das chaves de um determinado estágio i corresponda ao i–ésimo *bit* — a partir do *bit* mais significativo — do endereço do ponto de destino. Desse modo, o roteamento dentro da rede multiestágio pode ser feito de forma não–centralizada. Na Figura 3.3a a linha mais densa indica uma via estabelecida, por exemplo, entre um processador conectado à entrada 7 e o endereço 101.

Além do roteamento, cada chave tem ainda a capacidade de arbitrar a passagem a uma única entrada, no caso de acesso simultâneo por mais de uma delas a uma mesma saída. Se, na Figura 3.3b, ambas as entradas A e B requisitarem o mesmo terminal de saida, apenas uma delas será conectada e a outra será bloqueada ou rejeitada, devendo a requisição ser submetida novamente [HwBr90]. Redes multiestágios cujas chaves apresentam essa característica são ditas *bloqueantes*. Obviamente o bloqueio na comunicação é um fator de degradação do desempenho global. Para diminuir o efeito causado pelos bloqueios, *buffers* podem ser inseridos nas chaves. Essa estratégia tem demonstrado, através de análises e simulações, produzir melhorias consideráveis no desempenho desse tipo de rede [BYA89].

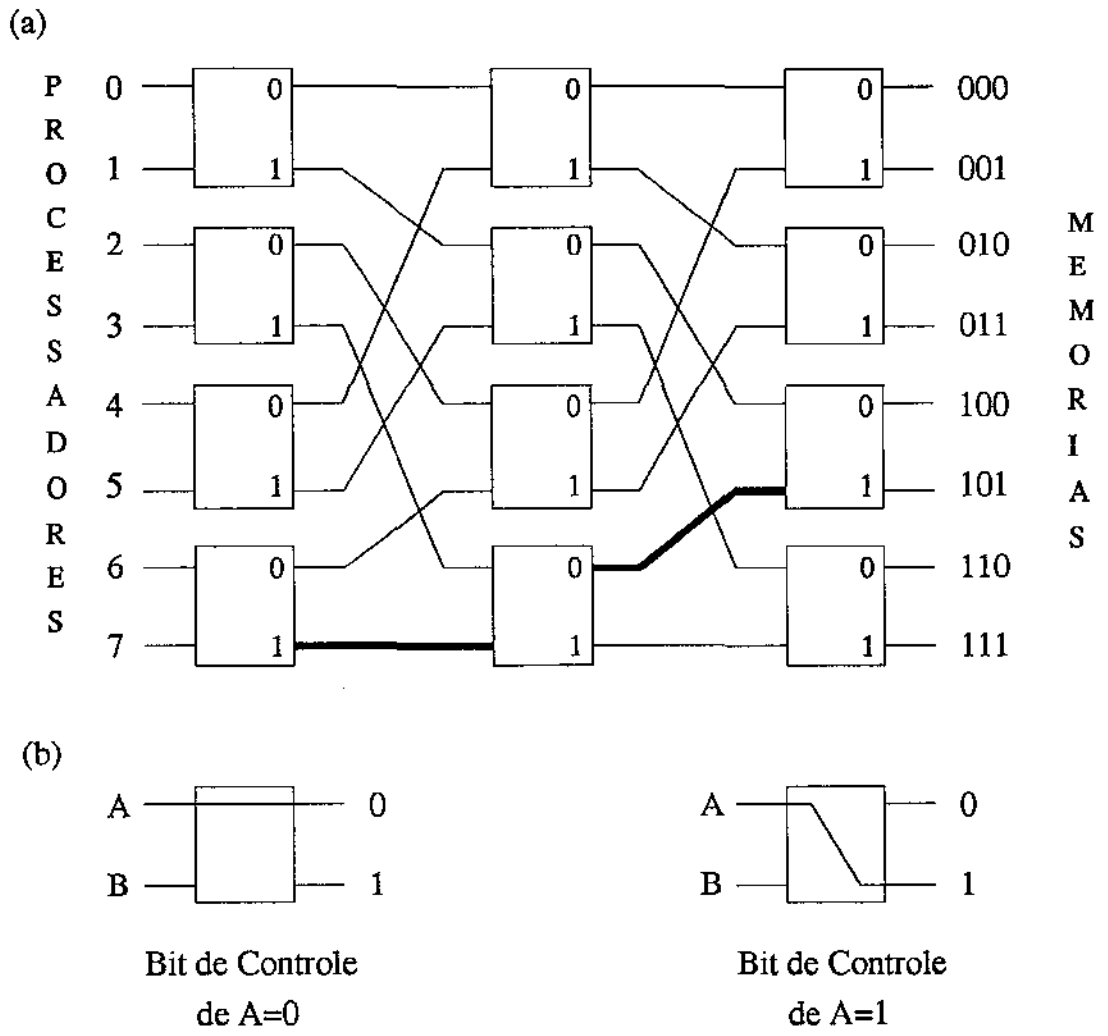

Figura 3.3: (a) Rede multiestágio $2^3 \times 2^3$ tipo Omega; (b) Funcionamento de uma chave

Outro fator de degradação do desempenho inerente às redes multiestágios é o atraso na comunicação inserido pelas chaves (circuitos ativos) em cada estágio. Até chegar ao ponto de destino, um sinal sofre um atraso proporcional ao número de estágios de que a rede é composta. Essa é, sem dúvida, a maior desvantagem das redes multiestágios, já que, ela contribui para aumentar o tempo de latência no acesso à memória.

Diversos outros modelos de redes multiestágios têm sido propostos. Em sua maioria, contudo, a única diferença existente está associada à forma com que as interconexões são feitas entre estágios adjacentes. A rede ilustrada na Figura 3.3, por exemplo, é do tipo Omega e é empregada nos sistemas NYU Ultracomputer e no RP3 da IBM [Taba90].

3.1.4 Comparação

As características dos tipos de rede discutidos nesta seção estão sumarizadas na Tabela 3.1 [Taba90], [BYA89]. É fácil verificar que o alto custo e complexidade são os principais fatores que tornam as redes *crossbar* raramente usadas. O barramento por outro lado, se não for considerado o seu alto potencial para se tornar um "gargalo", é bastante popular graças ao seu baixo custo e facilidade de conexão de outros módulos a um sistema já existente. As redes multiestágios por sua vez, vêm demonstrando ser uma opção bastante viável em relação às redes *crossbar* devido às suas características de custo e complexidade menores; essas propriedades têm justificado, inclusive, o desenvolvimento de projetos a níveis comerciais, como é o caso dos sistemas BBN Buterfly e IBM RP3 [ReTh86].

Tabela 3.1: Características dos vários tipos de redes de interconexão

Característica	*Barramento*	*CrossBar*	*Multiestágios*
Desempenho	baixo	alto	alto
Custo	baixo	alto	moderado
Complexidade	baixa	alta	moderada
Tolerância à Falhas	baixa (boa nos barramentos múltiplos)	moderada	moderada (boa com *hardware* adicional)

É possível concluir que muitas pesquisas e testes comparativos ainda têm que ser feitos antes de se chegar a uma rede ideal, capaz de superar os problemas de comunicação entre os módulos de um multiprocessador de memória compartilhada. O exposto nesta seção não pretende esgotar o assunto — não é esse o objetivo deste trabalho —, e sim esclarecer, sob o ponto de vista da comunicação, os problemas a que estão sujeitos os sistemas multiprocessados de memória compartilhada e, por conseguinte, os protocolos de coerência de cache. Abordagens aprofundadas, com análises e dados comparativos de desempenho, sobre diversas redes de interconexão podem ser encontradas, além das referências citadas, em [Feng81], [Pate81] e [Sieg85].

3.2 Esquemas de Diretórios

Nesta seção serão apresentadas diversas soluções propostas para o desenvolvimento de protocolos de coerência usando diretórios. Essas soluções podem ser divididas em soluções centralizadas e distribuídas. Nas soluções centralizadas, o diretório reside, normalmente, na memória e mantém informações relacionadas aos blocos de memória, enquanto nas soluções distribuídas, as informações no diretório correspondem aos blocos nos caches e se encontram armazenadas juntamente com eles nos próprios caches. Para cada solução serão apresentadas e discutidas as suas principais características e contribuições. Como uma das maiores preocupações atuais dos projetistas de protocolos em relação aos diretórios, diz respeito ao *overhead* de espaço ocupado por eles, esse aspecto será discutido com mais profundidade em cada caso; o gráfico da Figura 3.13, no final desta seção, compara o *overhead* de espaço imposto por diversas soluções considerando uma configuração de sistema multiprocessado padrão.

3.2.1 Diretórios Totalmente Mapeados

Diretório Réplicas dos Diretórios de Cache

O primeiro esquema de coerência de cache usando diretório relatado na literatura foi proposto por Tang [Tang76] em meados dos anos 70. Nessa época, pouca ou quase nenhuma informação existia sobre a utilização de memórias cache em sistemas multiprocessados de memória compartilhada. Segundo Tang, uma possível forma de controlar a coerência da informação nos caches de um sistema multiprocessado consiste na duplicação, em memória, de todos os diretórios dos caches do sistema, formando assim, um diretório central contendo a informação da distribuição dos blocos de memória por todo o sistema. Além da estrutura padrão dos ambientes monoprocessados (*tag* de endereço e *bit* de registro de modificação do bloco), tanto os diretórios nos caches quanto o diretório central contam ainda com um *bit* extra que indica o tipo do bloco em relação aos outros caches: compartilhado ou privado. A Figura 3.4 descreve graficamente a inter-relação dos dois tipos de diretórios.

O diretório central é gerenciado por um controlador próprio, que troca comandos de consistência com os controladores de cada cache no sistema; qualquer operação que provoque alteração no *status* do diretório do cache (ausências na leitura ou escrita, substituição de blocos, modificação do tipo do bloco, etc.) deve ser informada ao controlador do diretório central e vice–versa. Em uma ausência na escrita de um bloco compartilhado, por exemplo, o controlador

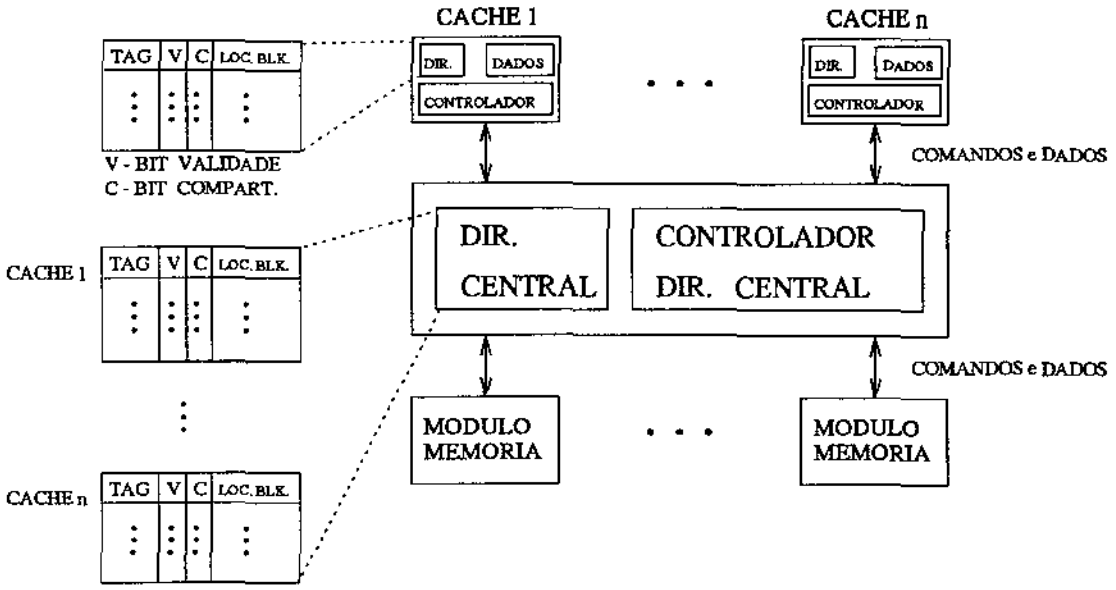

Figura 3.4: Inter-relação dos diretórios nos caches e o diretório central no esquema de Tang.

do cache envia um comando de leitura de um bloco privado para o controlador do diretório central que, por sua vez, avisa a todos os caches que compartilham o bloco, via informação obtida no diretório central e um comando correspondente, que eles devem invalidar suas cópias — isso é também registrado no diretório central.

Não é difícil perceber que a solução de Tang está diretamente associada à realidade tecnológica da época em que ela fora proposta: sistemas com o número de processadores restrito a poucas unidades. Diversos fatores impedem a sua aplicação em sistemas maiores [ArBa84]: o controlador do diretório central teria que ter um enorme poder de processamento, já que, todas as cópias dos diretórios dos caches devem ser pesquisadas, paralelamente, para se determinar, por exemplo, qual cache possui um bloco modificado durante uma requisição desse bloco por outro cache; além disso, seria necessário que todos os caches fossem organizados de maneira similar e com baixa associatividade preferencialmente, condições consideradas muito restritivas. Outro fator desfavorável é que o projeto da unidade controladora/diretório central dependeria do número de processadores do sistema, o que impediria a possibilidade de expansão do sistema. Finalmente, mudanças nos caches ocasionando o envio de comandos para o controlador do diretório central, responsável também por operações de pesquisa e atualizações, o transformariam, sem dúvida, em um "gargalo" para o sistema.

Diretório Mapa de Bits

Um esquema mais adequado a sistemas com um número maior de processadores, considerado por muitos autores como o primeiro representante viável na prática para o desenvolvimento de protocolos usando diretórios foi proposto por Censcier e Feautrier [CeFe78]. Ao invés de duplicar os diretórios como na solução anterior, esse esquema associa a cada bloco existente em um módulo de memória, um vetor de *bits* de "presença" ou validade, com tantas posições quantas forem os caches existentes no sistema. Cada elemento desse vetor (*bit*) indica a existência (ou não) daquele bloco no cache correspondente. Uma vez que um *bit* é necessário para indicar a relação de presença do par bloco–cache, o termo "totalmente mapeado" foi atribuído a esse tipo de diretório e, apesar de existirem muitas variações — algumas delas apresentadas mais adiante —, ele é considerado o esquema totalmente mapeado original. Um *bit* extra para cada bloco indica se ele fora modificado ou não (Figura 3.5). Dessa forma, um bloco não referenciado tem o seu vetor de presença preenchido com o valor 0, um bloco referenciado (apenas lido) por um ou mais processadores é indicado pelo valor 1 em um ou mais elementos do vetor, e um bloco modificado em um dos caches possui o *bit* extra ligado e um, e apenas um, dos elementos no vetor com o valor 1.

Uma vez que se pode ter acesso direto ao vetor de presença (através do endereço do bloco), é possível construir um controlador mais rápido que o da solução de Tang para gerenciar a informação armazenada no diretório. Além disso, se a memória for organizada de modo que um bloco resida completamente em um único módulo de memória, os mapas de *bits* podem ser distribuídos pelos módulos, cada um contendo um controlador responsável unicamente pelo mapeamento dos blocos de memória correspondente. Isso elimina o perigo potencial de "gargalo" de um controlador centralizado (Figura 3.5). Tanto a memória quanto o diretório podem, inclusive, ser organizados de forma entrelaçada para melhorar o desempenho.

A estrutura necessária à manutenção do mapa de *bits*, por outro lado, torna a solução totalmente mapeada cara em termos de espaço ocupado. Por exemplo, um sistema com 16 processadores e 64 *Mbytes* de memória global requer cerca de 12,5% a mais de memória para o armazenamento do diretório (considerando blocos de 16 *bytes*). Posto formalmente, devido ao fato do tamanho de uma entrada associada a cada bloco no diretório ser proporcional ao número de processadores/caches ($\Theta(N)$) e o tamanho do próprio diretório ser proporcional ao número de blocos na memória ($\Theta(M)$) e considerando que a quantidade de memória global do sistema cresce juntamente com o número de processadores, o *overhead* de espaço ocupado pelo diretório é proporcional ao produto dessas duas grandezas, ou seja, $\Theta(MN)$ uma função quadrática.

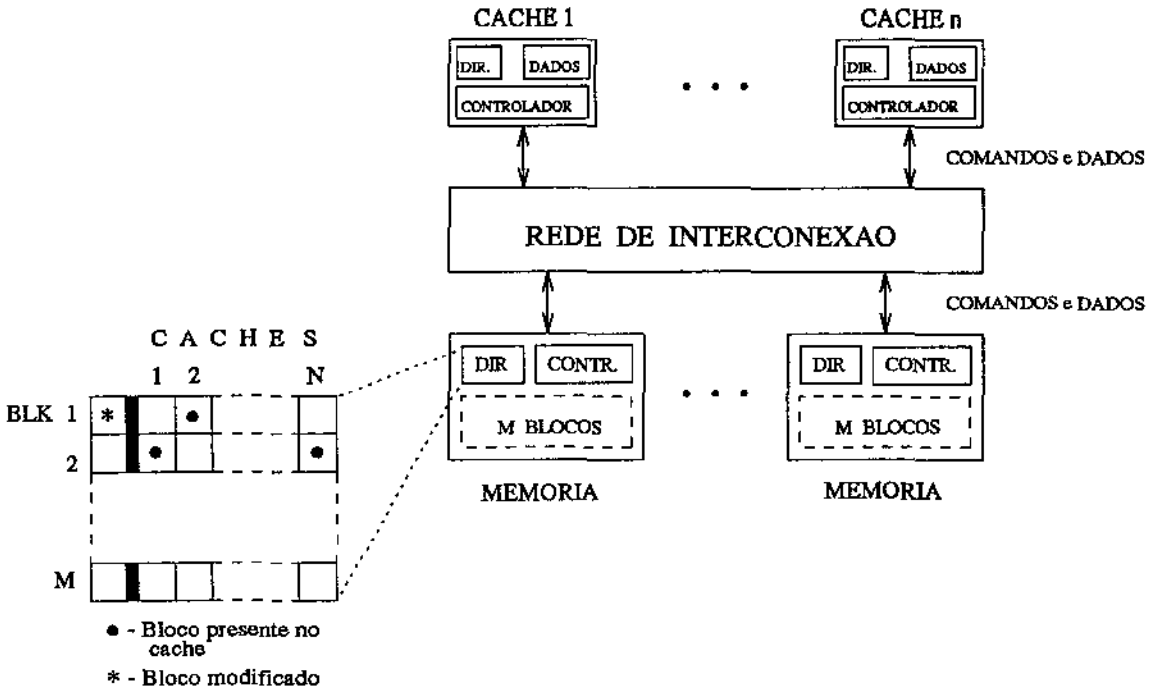

Figura 3.5: Organização de um Diretório Totalmente Mapeado

Além disso, o tamanho do vetor de *bits* definido na fase de projeto é um fator limitante do grau de expansão do sistema, já que, ele determina o número máximo de processadores que o sistema pode comportar.

Uma extensão para essa solução foi proposta por Yen e Fu [YeFu82] (cf. [ArBa84]) e consiste em adicionar um *bit* a cada bloco residente no cache para indicar se a cópia do bloco é exclusiva ou não. Com isso, escritas a blocos privados podem ser feitas sem consulta prévia ao diretório na memória (o envio de um sinal para atualizá-lo, porém, ainda é necessário). Apesar de melhorar o desempenho, essa extensão introduz problemas de sincronização relacionados à manutenção da consistência entre os estados nos caches e a informação no diretório, o que pode ocasionar um aumento significativo da complexidade do protocolo de coerência envolvido.

Archibald e Baer [ArBa84] propõem eliminar totalmente o vetor de *bits* de presença, que passa a ser composto unicamente por 2 *bits* por entrada, para indicar um dos quatro estados possíveis para cada bloco: *ausente* (não presente em nenhum dos caches); *exclusivo* (presente em apenas um cache); *presente** (presente em mais de um cache); *presenteM* (presente em um cache e modificado).

A principal diferença entre esse esquema, conhecido como solução *2–bits*, e o enfoque totalmente mapeado é que a informação indicando que caches possui cópias de um bloco não

está disponível. Isso implica que, quando é necessário enviar um sinal para um cache, em conseqüência de uma operação iniciada por um outro cache, se tem que recorrer a um esquema de difusão (*broadcast*) como nas soluções orientadas a redes tipo barramento, apresentadas no capítulo anterior. A maior vantagem da solução *2-bits* é que o diretório torna-se significativamente reduzido, com entradas de tamanho fixo, o que facilita a expansão do sistema onde ele é implementado. O *overhead* de espaço ocupado pelo diretório passa a ser o produto de uma constante pelo aumento de memória disponível, que continua sendo proporcional ao número de processadores, ou seja, $\Theta(N)$. A eficiência, contudo, está diretamente associada à porcentagem de blocos compartilhados sujeitos à operação de escrita; modelagens matemáticas demonstram que essa porcentagem deve ser extremamente reduzida (1% ou menos) para que o *overhead* imposto pela difusão dos comandos de consistência pela rede de interconexão não comprometa o desempenho, em sistemas com um número razoável de processadores (com 1% de taxa de compartilhamento de blocos sujeitos à escrita, o número máximo de processadores é da ordem de 64, segundo análise dos autores [ArBa84]).

Diretórios de Superblocos e Diretórios em Caches

Dois outros esquemas que visam reduzir o *overhead* de espaço imposto pelo esquema de Censcier e Feautrier são propostos por Kafka e Newton [OkNe90]. O primeiro deles consiste em agrupar os blocos da memória em superblocos, associando um elemento do vetor de *bits* de presença a cada superbloco, ao invés de a cada bloco como no esquema original. No cache, *bits* de validade e de modificação registram o estado local de cada bloco. Além do vetor de *bits* por superbloco, cada entrada do diretório na memória é composta ainda, pelo endereço do cache que contém a versão mais atualizada do bloco – o *dono* do bloco – e por um *bit* de indicação de modificação para cada bloco de que um superbloco é constituído, como mostra a Figura 3.6.

A idéia básica por trás dessa estrutura é reduzir o vetor de *bits* por módulo de memória, sem contudo, aumentar o número de ausências nos caches ocasionadas pela invalidação de um superbloco devido à modificação de apenas uma pequena parte dele (um bloco). Resumindo, o vetor de *bits* de um superbloco terá tantas posições ligadas, quantas forem os caches que possuírem cópias de um ou mais blocos daquele superbloco.

Segundo os autores, esse esquema é uma generalização do esquema de Censcier e Feautrier, já que, nesse último, os blocos podem ser vistos como superblocos constituídos de um único bloco.

O *overhead* de espaço ocupado pelo diretório continua sendo proporcional a $\Theta(MN)$.

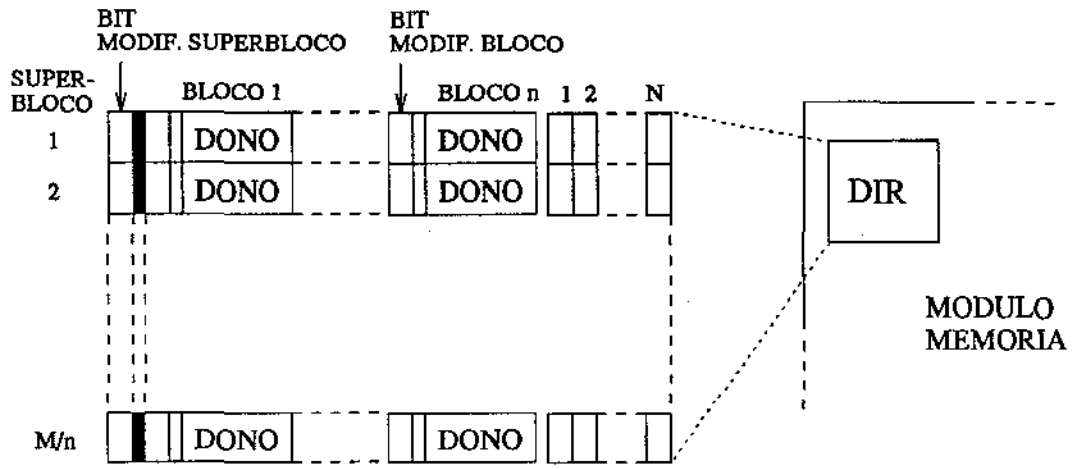

Figura 3.6: Estrutura de um Diretório de Superblocos.

Contudo, na prática, pode-se ter um número muito maior de processadores e uma quantidade maior de memória, com o diretório ocupando um espaço muito menor. Ou seja, o esquema original requer $N + 1$ *bits* para o vetor de presença por bloco, onde N representa o número de processadores; o esquema proposto, por sua vez, requer $\frac{N+1}{n} + \log N + 1$ *bits* por bloco (Figura 3.6) para o mesmo número de processadores, sendo n o número de blocos em um superbloco. Exemplificando, para um sistema com 512 processadores e 32 blocos por superbloco, o esquema de Censcier e Feautrier requer 513 *bits* por bloco enquanto que esse esquema requer apenas 26.

O preço pago pela redução no tamanho do diretório é o *overhead* de comunicação. Sinais de invalidação devem ser enviados a todos os caches que retêm parte de um superbloco, quando este sofre uma operação de escrita; alguns desses caches, contudo, podem não conter o bloco envolvido – fenômeno conhecido como *falso compartilhamento* (discutido no capítulo 2 deste trabalho). Resultados obtidos por Kafka e Newton através de simulações confirmam essa observação: à medida que se aumentou o tamanho do superbloco ou do próprio cache, percebeu-se um aumento moderado no tempo de acesso ao cache e acentuado no tráfego da rede, devido à necessidade de envio de um número maior de sinais de invalidação.

O número de posições no vetor de *bits* de presença continua limitando o número máximo de processadores no sistema.

O outro esquema sugerido por Kafka e Newton equivale à união de duas outras soluções que partem da mesma observação: na matriz formada pelo conjunto dos vetores de bits de presença do esquema Censcier e Feautrier (Figura 3.5), apenas um número fixo de bits fica ligado em um determinado intervalo de tempo; esse número corresponde, no máximo, à quantidade

de blocos comportados pelos caches do sistema. Logo, é possível visualizar a aplicabilidade das próprias memórias cache em substituição aos diretórios, com grande redução do espaço ocupado. Esse cache–diretório[3] teria funcionalidade semelhante aos TLB's (*Translaction Lookasside Buffer*) existentes nos sistemas que suportam gerenciamento de memória virtual [4], com uma diferença: não existe a necessidade de armazenamento secundário; caso um bloco seja referenciado e não exista nenhuma entrada no cache–diretório correspondente a ele, uma das entradas deve ser liberada, segundo uma política preestabelecida, sendo as cópias do bloco que anteriormente ocupava a entrada invalidadas.

Kafka e Newton propõem a utilização de dois tipos de cache–diretório. O primeiro deles grande, com cada entrada (linha) sendo capaz de armazenar o endereço de um pequeno número de caches que contêm cópias do bloco que está ocupando a entrada em um dado instante, como mostra a Figura 3.7a [5]; o segundo, menor que o anterior, com cada entrada armazenando todo o vetor de bits de presença (como no esquema original) do bloco associado à entrada naquele instante, como mostra a Figura 3.7b.

Quando um bloco é referenciado pela primeira vez, uma entrada para ele é alocada no primeiro tipo de cache–diretório. Quando o número de cópias desse bloco exceder o número de endereços suportados pela entrada, uma entrada no segundo tipo de cache–diretório é alocada e as informações do bloco transferidas do primeiro para o segundo tipo de cache–diretório — a entrada no primeiro cache–diretório fica liberada.

Na realidade, essa solução é a união de duas outras propostas feitas por Archibald e Baer [ArBa84] e por Gupta *et al.* [GWM90] (cf. [MPT91]). Archibald e Baer propõem o primeiro tipo de cache–diretório como melhoria de sua solução *2–bits* apresentada anteriormente; antes de recorrer à difusão na hora de efetuar invalidações, o controlador de memória buscaria se o bloco associado possui uma entrada no cache–diretório e, caso encontrasse, enviaria o sinal somente aos caches com cópia do bloco em questão, efetuando a difusão para os casos não encontrados. Gupta *et al.* baseiam-se na proposta original de Censier e Feautrier e sugerem a utilização de um diretório reduzido cujo número de entradas é, no mínimo, igual ao número de blocos que os caches do sistema comportam. O grau de redução é dado pela razão entre o

[3]Essa designação será usada doravante, como forma de diferenciar os caches substitutos do diretório dos outros caches do sistema.

[4]Um TLB é um pequeno cache, cuja função é armazenar as entradas mais recentemente utilizadas de uma tabela de páginas, evitando a busca freqüente dessas entradas na memória durante o processo de tradução de um endereço virtual para real.

[5]Os fatores que justificam essa forma de organização são os mesmos relacionados à classe de soluções conhecida como *Diretórios Limitados*, apresentados e discutidos mais adiante.

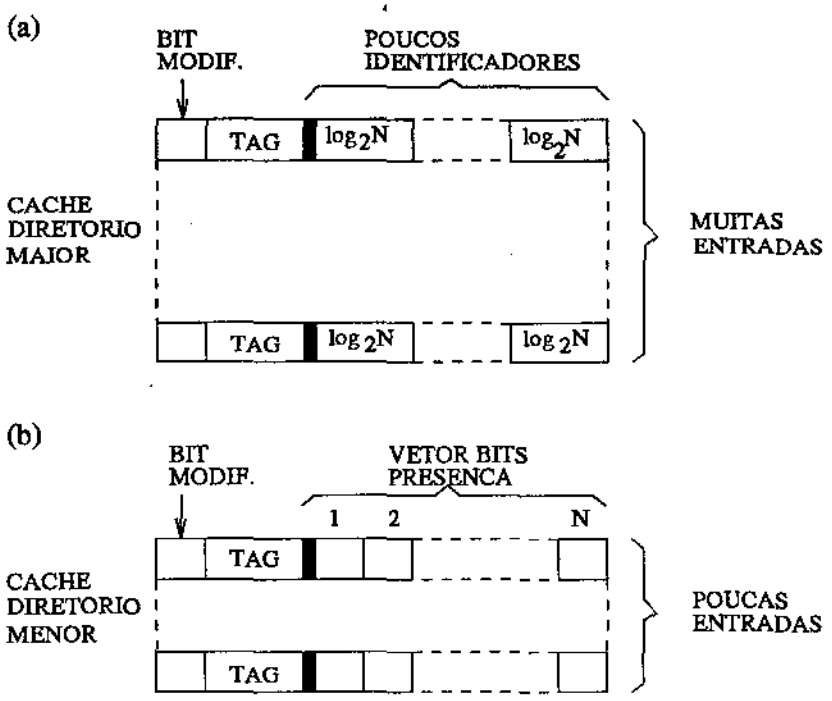

Figura 3.7: Estrutura dos Diretórios em Cache.

número de blocos da memória e o número de entradas no diretório. Dessa forma, o diretório reduzido pode ser implementado de modo semelhante ao segundo tipo de cache–diretório da solução de Kafka e Newton.

A economia alcançada com a utilização de diretórios na forma de caches pode ser medida pelo número de *bits* necessários à sua implementação. No primeiro tipo de cache discutido acima, considerando um sistema com N caches e M blocos de memória, o número de *bits* em um cache diretamente mapeado é dado por $\frac{M}{K}(n \log_2 N + \log_2 K + 1)$, sendo n o número máximo de caches endereçados em cada entrada e K o grau de redução do diretório. No segundo tipo, gasta-se $\frac{M}{K}(N + \log_2 K + 1)$. Por exemplo, em um sistema com 64 processadores, caches de 32 *Kbytes* (blocos de 16 *bytes*) e memória global 128 *Mbytes*, gasta-se cerca de 400 *Kbytes* e 1200 *Kbytes* (para $K = 64$ e $n = 4$) para o primeiro e segundo tipos de diretórios respectivamente, contra 64000 *Kbytes* caso se implementasse um diretório com uma entrada para cada bloco de memória. Por outro lado, deve-se considerar, para efeito de comparação, o fato de se estar utilizando memórias cache (mais caras) na implementação dos diretórios.

Apesar do aumento da capacidade do *hardware* do controlador de memória devido à inclusão de caches–diretório, os resultados obtidos por Kafka e Newton em simulações indicam um desempenho em termos de tempo de acesso e tráfego induzido na rede desse tipo de solução bem próximo do obtido na solução ótima de Censier e Feautrier.

Diretório Hierárquico

Nesse esquema, para efeito de construção do diretório, os processadores são agrupados em conjuntos de n elementos ($\log_2 N$ inteiro) com n desses conjuntos formando um elemento ou grupo de um outro conjunto de nível hierarquicamente superior [MPT91]. Assim, um sistema com 64 processadores pode ser organizado como 8 grupos, cada um contendo 8 processadores, ou como 4 grupos, cada um contendo 4 subgrupos com 4 processadores cada, como mostra a Figura 3.8. O objetivo dessa forma de organização é dividir o diretório em subdiretórios, alocados a cada grupo, com o vetor de *bits* de presença passando a indicar a existência de blocos válidos em grupos hierarquicamente inferiores. Dessa forma, um *bit* ligado no vetor correspondente a um determinado bloco, no diretório de um grupo antecessor, indica a existência daquele bloco no grupo posterior correspondente à posição ligada; um *bit* extra, por bloco, indica a existência de cópias do bloco em grupos não descendentes, evitando com isso o envio de sinais de invalidação desnecessários. Obviamente, caso o diretório fosse construído estritamente como exposto, cada subdiretório deveria possuir entradas para todos os blocos residentes nos grupos descendentes; isso ocasionaria um crescimento linear do diretório à medida que se caminhasse dos níveis superiores para os níveis inferiores na hierarquia, até o número máximo correspondente a todos os blocos de caches do sistema. Considerando, contudo, que, a cada instante, apenas uma pequena parte dos blocos de um grupo antecessor é realmente utilizada nos grupos descendentes, os subdiretórios podem ser implementados sob a forma de cache–diretório como na solução de Kafka e Newton [OkNe90].

O *overhead* de espaço total da organização hierárquica é dado por:

$$C\{\sum_{i=1}^{m} \frac{1}{K_i}[n + \log_2 K_i + 1]\} + \frac{M}{K_0}(n + \log_2 K_0)$$

onde n indica o número de elementos por grupo e K_i o grau de redução do subdiretório no nível i. A partir de uma comparação numérica é possível visualizar quão reduzido esse *overhead* pode ser em relação às outras soluções. Admitindo-se um sistema com a mesma configuração do exemplo da solução com cache–diretórios anterior, requer-se, para o esquema hierárquico 8×8, um total de espaço para o diretório de 2,6 *Mbytes*, contra 16,5 *Mbytes* do segundo tipo de diretório da solução anterior e 72 *Mbytes* da solução original.

O diretório hierárquico pode ser distribuído como parte dos elementos da própria rede de interconexão, como, por exemplo, das chaves em uma rede multiestágio ou nos nodos de outros

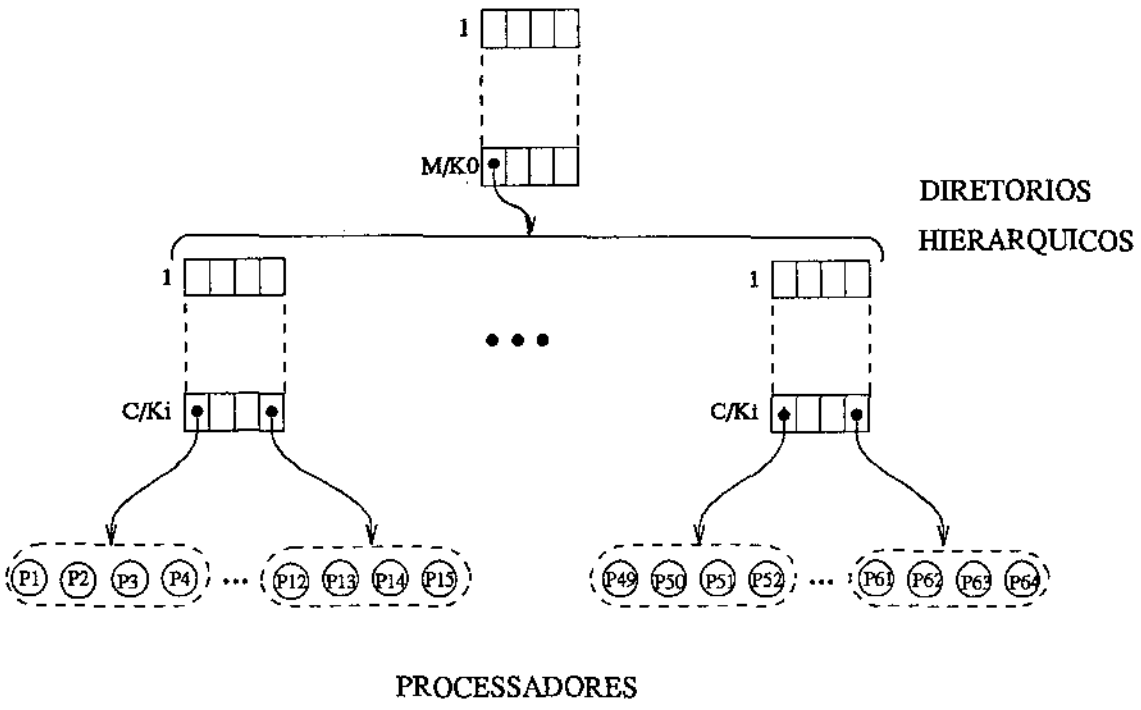

Figura 3.8: Estrutura de um diretório hierárquico, com 4 elementos por grupo.

tipos de rede que apresentem estrutura hierárquica. De acordo com os resultados obtidos em simulações, o desempenho dos diretórios hierárquicos é similar ao da solução mapa de *bits*, com ligeira vantagem em aplicações onde se observa localidade geométrica (onde grande parte das variáveis compartilhadas encontram-se em um mesmo grupo de processadores, dispensando o envio de sinais de coerência pela rede).

3.2.2 Diretórios Limitados

A maior restrição imposta pelos diretórios totalmente mapeados é, como visto, o *overhead* de espaço resultante de seu armazenamento. Uma solução imediata para reduzir esse *overhead* consiste em limitar à uma constante, o número máximo de cópias simultâneas em caches dos blocos de memória. Com isso, reduz-se o espaço necessário em cada entrada do diretório para armazenar a identificação dos caches com cópias dos blocos. Diretórios estruturados dessa forma são conhecidos como diretórios limitados [ASHH88]. A Figura 3.9 exibe a estrutura básica dos diretórios limitados. O vetor de *bits* dos diretórios totalmente mapeados é substituído por um vetor de apontadores (menor em termos de espaço) na versão limitada.

Quando um determinado cache desejar compartilhar um bloco cuja entrada já esteja completamente preenchida, um dos apontadores da entrada deve ser liberado — invalidando-se

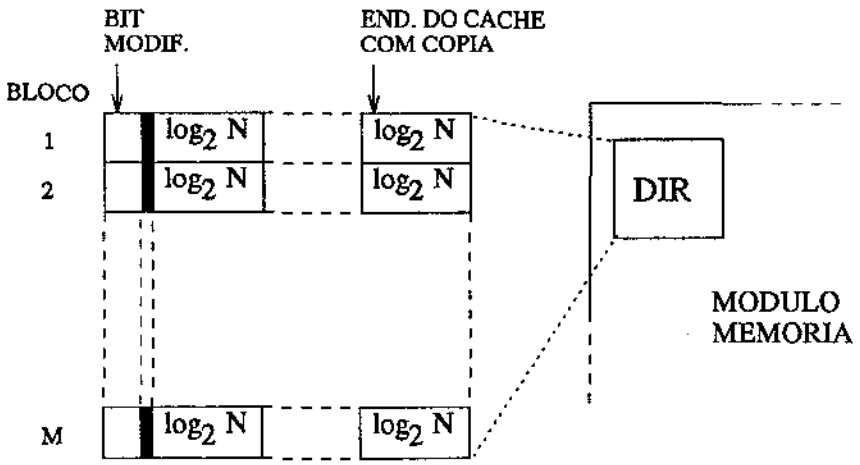

Figura 3.9: Estrutura de um diretório limitado.

a cópia do cache correspondente — para em seguida armazenar o endereço do novo cache, antes de permitir o acesso desse cache ao bloco.

Além do *overhead* de espaço reduzido, o principal fator que motiva a utilização dos diretórios limitados em protocolos de coerência está associado ao comportamento peculiar que as aplicações paralelas apresentam com relação às variáveis compartilhadas. Estatísticas obtidas a partir da seqüência de referências feitas por algumas aplicações na fase de execução indicam que, normalmente, apenas uns poucos processadores fazem referência à determinadas posições antes de elas serem modificadas [ASHH88], [GuWe92], [SH91]. Portanto, é de se esperar que as entradas nos diretórios, constituídas de poucos apontadores, sejam capazes de, na maioria dos casos, comportar o endereço dos caches que possuem uma cópia de um bloco, antes do envio de sinais de invalidação.

A economia no espaço utilizado pelos diretórios limitados pode ser sentida, a exemplo dos diretórios totalmente mapeados, pela análise do seu *overhead*. Dado que cada apontador na estrutura limitada requer $\log_2 N$ bits para armazenar o endereço de um processador em um sistema com N processadores, e considerando que a quantidade de memória cresce linearmente com o número de processadores, o tamanho de um diretório limitado cresce como $\Theta(N \log N)$, já que, é alocado um número pequeno e fixo de apontadores por entrada no diretório. Esse *overhead*, aproximadamente linear, muito menos severo, portanto, que o observado nos esquemas totalmente mapeados ($\Theta(N^2)$), torna os diretórios limitados bastante atrativos para a utilização em protocolos de coerência de cache em sistema multiprocessados sujeitos a grande expansão.

Os protocolos que implementam o esquema de diretórios limitados, utilizam-se dos en-

dereços armazenados nos apontadores existentes em cada entrada para enviar sinais de invalidação para os processsadores que compartilham um bloco prestes a ser modificado. Esses protocolos podem ser classificados como $Dir_iNB, i < N$ (N = número de processadores), já que, os sinais de invalidação são enviados a um grupo restrito e preestabelecido (i) de processadores (NB significa *non-broadcast* [ASHH88]). A maioria das variações que surgiram a partir desse modelo básico tentam estendê-lo, alterando, principalmente, as ações executadas no momento em que não existem mais apontadores disponíveis em uma entrada e ocorre uma nova requisição do bloco correspondente. Algumas dessas variações são discutidas a seguir.

Diretórios limitados com invalidação por difusão

Nessa variação, um *bit* associado a cada entrada no diretório é ligado no momento em que ocorre a falta de apontadores para a entrada. Isso indica que, a partir desse momento, a entrada não tem mais controle sobre quais caches possuem cópias do bloco e o protocolo é obrigado a admitir o pior: todos os caches possuem cópia do bloco em questão e algum mecanismo de difusão (*broadcasting*) deve ser usado no momento em que for necessário invalidá-las. Notacionalmente, esse esquema é representado por Dir_iB (B significa *broadcast*).

Um ponto desfavorável a esse tipo de extensão está associado à rede de interconexão usada no sistema; redes tipo ligação ponto a ponto (como as descritas na seção 3.1) não provêem um mecanismo eficiente para operações de difusão dos sinais, além de ser difícil determinar se todos os caches receberam e processaram os sinais de invalidação [CFKA90].

Simoni e Horowitz [SH91] desenvolvem um modelo de análise para diretórios limitados e o utiliza no estudo do comportamento de diretórios tipo Dir_iB e Dir_iNB e na definição do número ideal de apontadores necessários para manter toda a informação de compartilhamento no diretório em sistemas com milhares de processadores. Seus resultados demonstram que, mesmo sendo infreqüente, quando ocorre a falta de apontadores e o protocolo recorre ao esquema de difusão de invalidações, a degradação do desempenho da rede decorrente das invalidações "extras" (aquelas enviadas a caches que não possuem cópias do bloco) pode chegar a níveis inaceitáveis.

Diretórios em árvore

Para sanar o problema da falta de apontadores, sem ter que sacrificar as cópias já existentes ou tampouco recorrer a técnicas como a difusão, Maa *et al.* [MPT91] propoem um esquema de

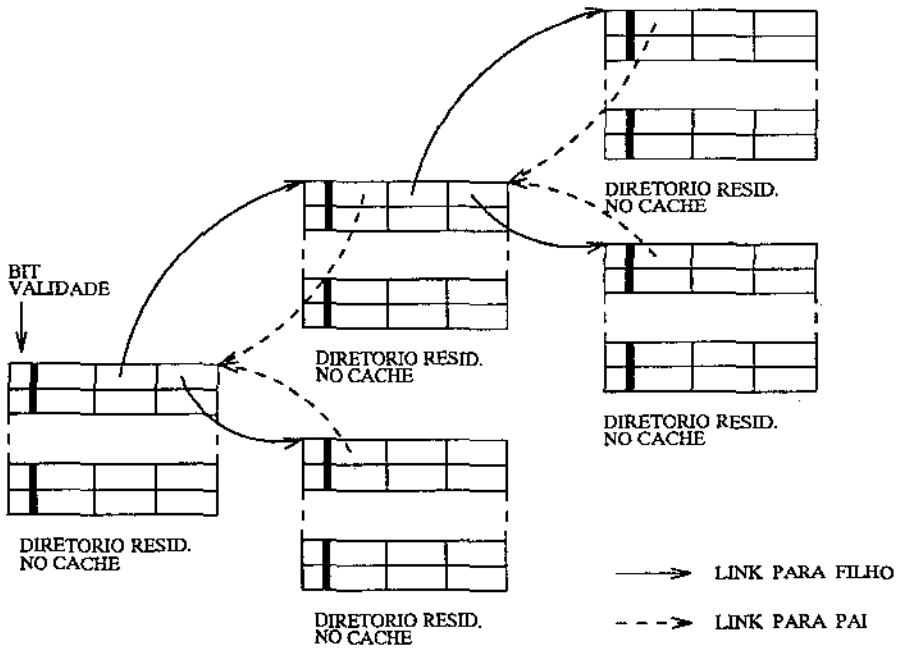

Figura 3.10: Estrutura de um diretório limitado em árvore.

diretórios limitados, segundo o qual, os apontadores para caches com cópias, se tornam nodos de uma árvore caso ocorra uma falta desses apontadores. Enquanto não ocorre a falta, esse esquema se comporta de forma similar ao esquema limitado original. Quando a falta é sentida, um *bit* extra associado a cada apontador é ligado, indicando que aquela entrada passa a apontar para uma subárvore cuja primeira entrada aponta para o nodo pai e o restante aponta para caches com cópias do bloco, como mostra a Figura 3.10. Essa configuração se estabelece até que uma nova falta seja sentida, desencadeando novamente o processo. Como uma das entradas é usada para apontar para o nodo pai, a árvore formada é do tipo $(i-1)$-ária — onde i representa o número de entradas inicialmente disponíveis — e o esquema é notacionalmente representado por $Dir_{(i-1)}*$.

O diretório é organizado de forma distribuída pelos caches; ou seja, a cada bloco no cache pode estar associado um nodo da árvore — provavelmente alocado em uma área especial de armazenamento existente para esse fim em cada cache do sistema. Além disso, cada bloco na memória possui o endereço do cache onde se inicia a árvore do bloco (raiz). O *overhead* de espaço ocupado é dado, portanto, por $(iC+M)\log_2 N + iC$, com C sendo o total de blocos nos caches, M o total de blocos na memória, N o número de processadores e i o número de entradas por nodo.

É esperado que o gerenciamento da estrutura se torne muito complexo caso a árvore associada a um bloco cresça muito. Além disso, os autores não deixam claro como a estrutura

se comporta, no caso de substituição de um bloco ao qual está associado um nodo intermediário. Por outro lado, os fatores que motivam essa solução são os mesmos que os da solução limitada original; ou seja, em aplicações usuais, o número de cópias de blocos compartilhados não é muito grande, fazendo com que a estrutura em árvore seja disparada apenas no tratamento de exceções.

3.2.3 Diretórios Encadeados

Além do *overhead* de espaço, os protocolos de coerência baseados nos modelos de diretórios já discutidos, herdam destes uma outra característica que pode representar um fator de degradação do desempenho: a serialização no acesso ao diretório central [6]. Se durante o atendimento a um processador o controlador do diretório receber uma requisição de outro processador, mesmo que seja para um bloco diferente, essa requisição ou é armazenada em uma fila (*buffer*) para atendimento posterior, ou é rejeitada e o processador deve emitir uma nova requisição, retardando a execução da aplicação e aumentando o tráfego na rede.

Visando sanar essa restrição, surgiram modelos de diretórios que adotam um enfoque mais distribuído. A idéia básica desse modelo consiste em suprimir o diretório do módulo de memória, distribuindo as informações nele contida pelos próprios caches do sistema. A seguir serão discutidas algumas soluções que se enquadram nesse modelo.

Diretórios encadeados, ou diretórios em lista ligada, oferecem a capacidade de expansão, com o mesmo *overhead* de espaço das melhores soluções centralizadas, através do encadeamento, via apontadores, das cópias compartilhadas de um bloco. Thapar e Delagi [ThDe90] propõem que os blocos na memória e suas respectivas cópias nos caches sejam encadeados usando uma lista ligada simples, começando pelo bloco na memória. Cada bloco, na memória e nos caches deve, portanto, dispor de um campo extra para armazenar o apontador para o próximo cache com cópia do bloco.

A Figura 3.11 demonstra a formação da lista ligada, à medida que vão sendo requisitadas cópias de um bloco. Primeiramente, o cache A sente a ausência do bloco e envia um comando de leitura para a memória; considerando que essa é a primeira leitura do bloco, ou seja, nenhum outro cache possui cópia do bloco, o apontador do bloco na memória é preenchido com o endereço de A e a própria memória envia uma cópia do bloco, que assume o estado *exclusivo*

[6] Mesmo quando a memória global está distribuída pelos diversos módulos do sistema, as informações relacionadas aos blocos de memória de um determinado módulo encontram-se centralizadas no diretório daquele módulo.

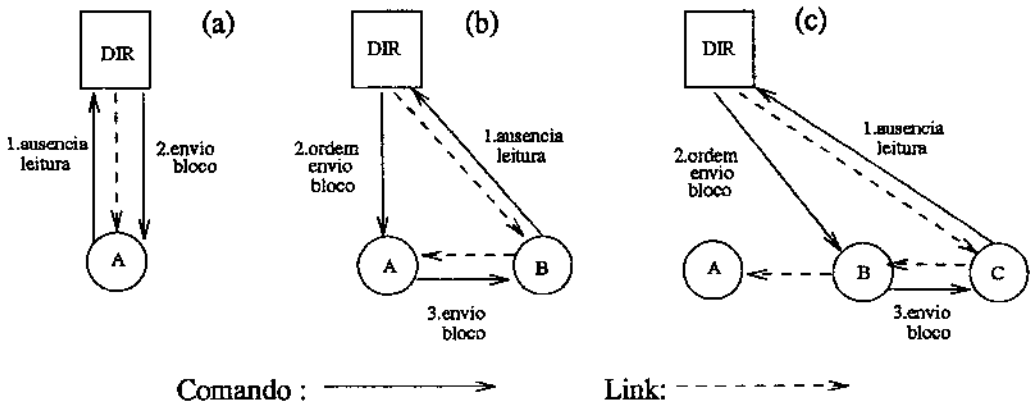

Figura 3.11: Estabelecimento de uma lista ligada simples em um diretório encadeado.

em A (Figura 3.11a). Em seguida, o cache B também envia uma requisição do bloco; agora, a memória, ao perceber que já existem outras cópias do bloco, envia o pedido (e o endereço de B) para que ele seja atendido por A, já que, ele pode ter alterado o bloco localmente [7]. Ao receber o comando, A envia uma cópia do bloco e atualiza o apontador de sua cópia para B — a memória, atualiza o apontador do bloco na memória para B (Figura 3.11b). O processo se repete para C (Figura 3.11c).

No processo de atendimento a uma requisição para escrita, além do envio da cópia do bloco — caso o processador requisitante já não a tenha — é necessário percorrer a lista para efetuar invalidações. A Figura 3.12 mostra a seqüência de eventos que ocorrem quando vários caches compartilham a cópia de um bloco e uma ausência durante a escrita do bloco é sentida no cache D. O controlador do diretório remete a requisição para o cache topo da lista C, a partir do seu endereço disponível no apontador do bloco na memória — que passa a apontar para D; quando C recebe a requisição, ele invalida sua cópia e envia um sinal para que B faça o mesmo. Além disso, C envia a cópia do bloco para D. Quando B recebe o pedido de invalidação, ele invalida sua cópia e repassa o sinal para A que, sendo o fim da lista (não aponta para nenhum outro cache ou não possui cópia do bloco devido a uma substituição), invalida sua cópia e envia um outro sinal para D indicando que as invalidações alcançaram o fim da lista. Só após receber o sinal do topo e do último elemento da lista, D tem a certeza de que a escrita pode ser feita de forma consistente. Operações de escrita a blocos que não estão ausentes e que ja fazem parte de uma lista são executadas de forma semelhante [ThDe90].

[7] A transferência da requisição do bloco da memória para o cache topo da lista tem ainda, nos casos de bloco compartilhado, o desejado efeito colateral de liberar rapidamente o controlador de memória para o atendimento a outras requisições.

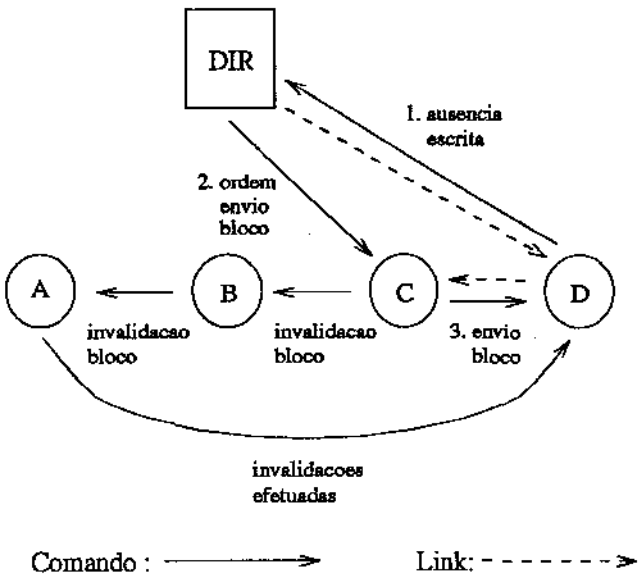

Figura 3.12: Seqüência de operações durante uma ausência para escrita em um diretório encadeado.

A substituição de um bloco que faz parte de uma lista é efetuada mediante a invalidação dos últimos elementos da lista, a partir do elemento substituído. Por esse motivo, o fim da lista pode também ser um cache que não possui cópia de um bloco.

A latência no atendimento a uma ausência pode ser um motivo de preocupação, já que, a lista de blocos compartilhados deve ser percorrida seqüencialmente. Segundo os autores, porém, se escritas a um bloco ocorrerem com freqüência, o número de caches a serem invalidados entre escritas será reduzido. Adicionalmente, os mesmos argumentos que justificam os diretórios limitados podem suportar a afirmação de que a lista ligada tende a ser pequena.

3.3 Análise de Desempenho

Chaiken *et al.* [CFKA90] apresentam os resultados de um estudo comparativo do desempenho de alguns dos esquemas de diretório discutidos neste capítulo. Seus resultados foram obtidos a partir de simulações dirigidas por referência que representavam o comportamento de diversas aplicações paralelas. A metodologia de análise adotada consistiu de três etapas: 1. coleta das referências geradas pelas aplicações (usando três ferramentas distintas); 2. simulação dos protocolos de coerência usando as referências obtidas na etapa anterior, produzindo taxas médias de requisições não atendidas pelo cache e/ou comandos de consistência a serem enviados pela rede; 3. simulação de uma rede de interconexão tipo multiestágio, que recebe os dados gerados

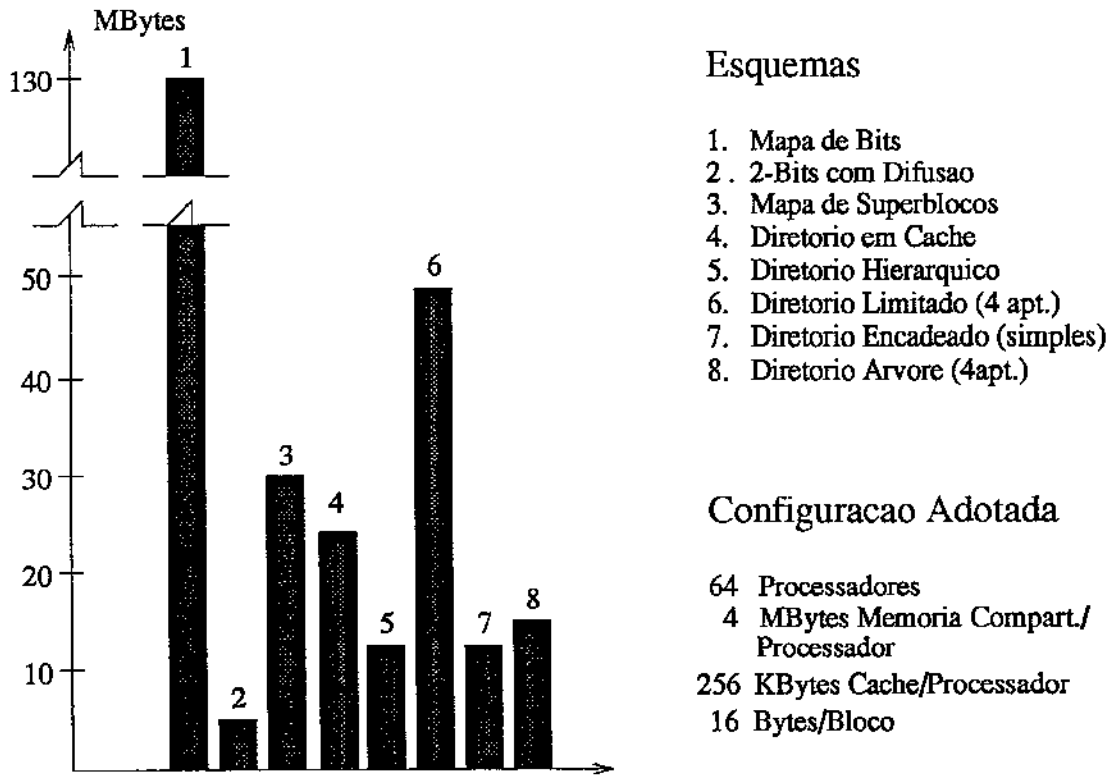

Figura 3.13: *Overhead* de espaço imposto por diversas esquemas de diretório para uma configuração de sistema multiprocessado específico.

na etapa 2 e produz a taxa média de utilização dos processadores, considerando a latência dos nodos da rede.

Os gráficos da Figura 3.14 demonstram como a utilização dos processadores (eixo horizontal) é afetada pelos protocolos avaliados (eixo vertical) em três das aplicações escolhidas devido ao comportamento distinto entre si; a aplicação *Weather* particiona a atmosfera em uma grade tridimensional e usa o método das diferenças finitas para resolver o conjunto de equações que descrevem o estado do sistema; a aplicação *Speech* compreende o estágio de decodificação léxica de um sistema de entendimento da linguagem falada baseado em fonemas; a aplicação *P-Thor* é um simulador paralelo de lógica. Todos os protocolos analisados baseiam-se no envio de sinais de invalidação durante a escrita dos blocos compartilhados. A barra maior, no final de cada gráfico, indica a execução das simulações sem a intervenção dos protocolos de coerência; apesar de não representar a execução correta das aplicações, já que, tanto blocos compartilhados como não compartilhados são tratados como não compartilhados, essa medida indica como a utilização do processador é afetada unicamente pela taxa de ausências sentidas na execução da aplicação, servindo, dessa forma, como um limite superior para o desempenho dos protocolos de coerência. A barra superior, por outro lado, indica o desempenho de um esquema que não permite um compartilhamento em cache de blocos sujeitos à modificação e, na maioria das vezes, pode ser visto como um limite inferior para o desempenho [8].

Como esperado, o desempenho do esquema Mapa de *bits* é o que mais se aproxima do ótimo, sendo extremamente melhor que os esquemas que só permitem a existência em cache de blocos privados, demonstrando assim, a validade do compartilhamento de dados e da utilização de protocolos de coerência, mesmo quando as aplicações não são escritas ou compiladas considerando essa forma de execução. Os esquemas limitados por sua vez, demonstraram quão dependentes eles são em relação ao grau de compartilhamento (na aplicação *Speech* verificou-se um vasto compartilhamento de uma estrutura de dados que, apesar não ser sujeita a modificações, provocou a execução de substituições desnecessárias de blocos para a liberação de apontadores) e ao método de sincronização adotado (na aplicação *Weather* foram usadas primitivas de sincronização comuns — *spin-locks*, barreiras, etc. — que, também, provocaram um número grande de substituições para a liberação de apontadores); na aplicação *P-Thor*, escrita visando a mínima necessidade de comunicação entre processadores via pontos de sincronização e o mínimo número de processos que lêem posições compartilhadas, os diretórios limitados

[8] Evidentemente, o comportamento dessa medida é dependente da quantidade de referências a blocos compartilhados que uma aplicação apresenta, e pode, em alguns casos, apresentar um desempenho superior ao dos outros esquemas.

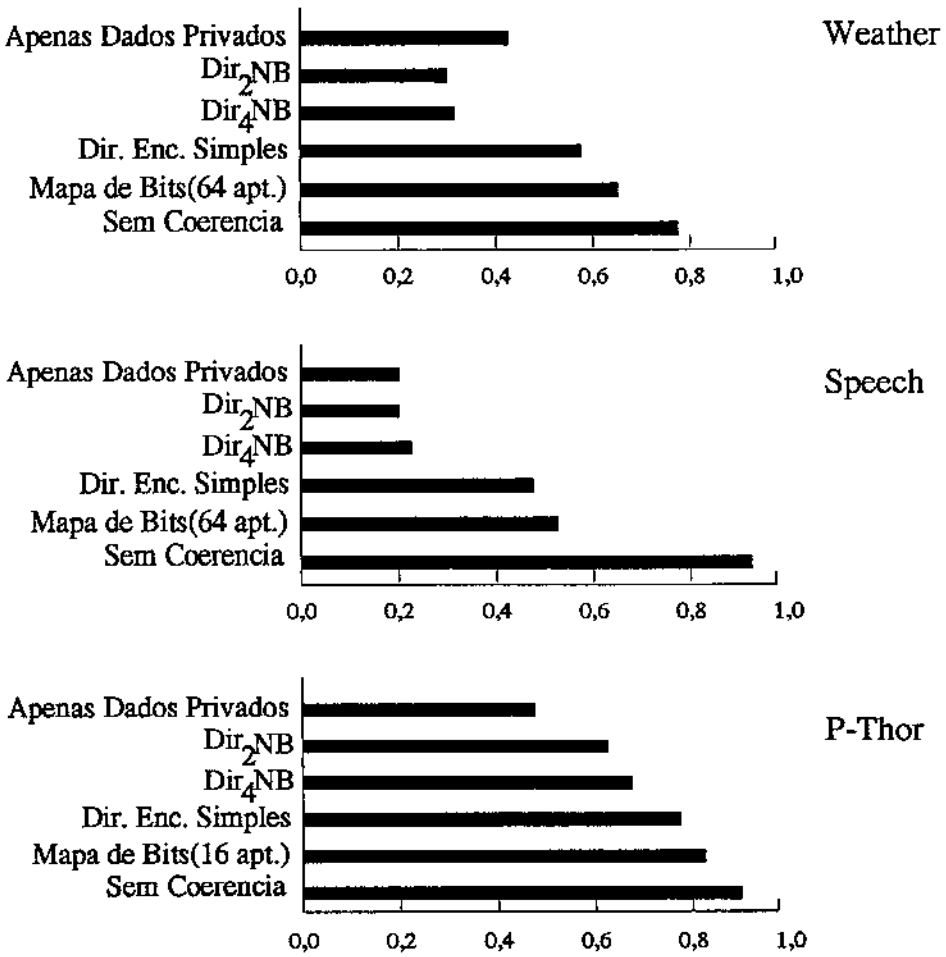

Figura 3.14: Comparação do desempenho de alguns esquemas de diretórios em três aplicações com comportamentos distintos (Directory–Based Cache Coherence in Large Scale Multiprocessors, *IEEE Computer, Jun. 90, pg. 55).*

apresentaram desempenho satisfatório. Finalmente, os diretórios encadeados apresentaram desempenho próximo aos esquemas Mapa de *bits*; apesar desse comportamento melhor que o dos diretórios limitados, não se deve deixar de considerar sua maior complexidade e latência mais alta nas operações de escrita a blocos compartilhados devido à serialização dos sinais de invalidação.

3.4 Exemplos de Implementação

A implementação prática de sistemas que utilizam diretórios ainda é restrita. Os exemplos que podem ser citados são, em sua maioria, protótipos que ainda estão em fase de testes. A seguir são apresentados alguns destes exemplos.

3.4.1 Multiprocessador Dash — Directory Architecture for Shared Memory

Essa máquina foi desenvolvida na Universidade de Stanford a partir de um outro sistema multiprocessado, comercialmente disponível, que funciona como módulo básico, ou nodo, na estrutura do Dash; trata-se da POWER Station 4D/240 da Silicon Graphics, um sistema constituído de 4 processadores de alto desempenho, cada um com seu próprio sistema de caches, interligados por um barramento *pipeline* e usando o protocolo Illinois (vide capítulo 2) na manutenção da coerência [Leno90], [Leno92]. No protótipo, um total de até 16 nodos podem ser conectados, através de uma rede tipo ponto a ponto; cada nodo dispõe ainda de uma parte da memória global e de um controlador de diretório que faz a interface com a rede de interconexão.

A coerência de cache inter-nodos é obtida a partir das informações armazenados nos diretórios, estruturados segundo uma configuração de Mapa de *Bits*, onde cada posição do vetor de presença de um bloco indica a sua existência em um dos nodos do sistema; o tamanho desse vetor é, portanto, no máximo igual a 16. Cada diretório/controlador é responsável por armazenar os vetores de *bits* referentes aos blocos da parte da memória global associada ao nodo e atender às requisições de blocos ou comandos de consistência enviados por outros nodos.

3.4.2 Multiprocessador Alewife

A máquina Alewife, um projeto de sistema multiprocessado desenvolvido no MIT [CKA91], é composta por módulos interligados por uma rede tipo ponto a ponto. Cada módulo é constituído de processador, unidade de ponto flutuante, memória cache e parte da memória globalmente compartilhada — o diretório encontra-se distribuído na memória dos diversos módulos do sistema. O protocolo de coerência utilizado é uma extensão proposta para o esquema de diretório limitado, conhecida como *LimitLess* (*Limited directory Locally Extended through Software Suport*). No LimitLESS, quando uma entrada do diretório esgota o número de apontadores disponíveis, o módulo de memória interrompe o processador local, que passa a emular um esquema tipo diretório totalmente mapeado para o bloco que causou a interrupção. Essa emulação por *software* se torna viável graças à existência, em um mesmo módulo processador, de uma unidade de controle de memória e de um processador dotado de mecanismos bastante rápidos de interrupção.

A vantagem desse esquema é que a emulação em *software* provê uma grande flexibilidade no gerenciamento de situações infreqüentes, sem contudo, necessitar de *hardware* extra — a depender do comportamento da aplicação, o *software* pode tomar diferentes tipos de ações que não sejam simplesmente a emulação de um esquema totalmente mapeado. Em contrapartida, com a tecnologia atual, o *overhead* imposto pela emulação completamente em *software* do diretório é significante. Uma possível solução, porém, seria, em lugar de interromper o processador, dotar o diretório de um controlador (em um único *chip*) e alguma memória RAM dedicada à gerência dos casos onde há esgotamento de apontadores disponíveis [SH91].

3.4.3 Scalable Coherent Interface

O padrão IEEE P1596 define uma interface (Scalable Coherent Interface — SCI) que provêm serviço de comunicação através da transmissão de pacotes via uma rede de ligações ponto a ponto [Gust92]. Ela resulta da experiência obtida no desenvolvimento de dois outros padrões para redes tipo barramento: o FastBus (IEEE960) e o FutureBus (IEEE896.X) — vide capítulo 2 — que, apesar de fornecerem vários recursos para a aplicação em sistemas multiprocessados, rapidamente demonstraram o inevitável "gargalo" inerente às redes tipo barramento (apenas uma transmissão por vez), e a velocidade limitada devido às características físicas desse tipo de rede (ocasionando transmissões imperfeitas à medida que a velocidade cresce). Como o grupo de trabalho do SCI especificou como objetivo de projeto um alto desempenho da interface (1

Gigabyte/segundo/nodo), a idéia de utilizar redes tipo barramento foi descartada e substituída pela utilização de ligações ponto a ponto unidirecionais.

Os principais objetivos a serem alcançados pelo padrão e que acabaram por emprestar as iniciais para formar seu nome foram:

- Expansibilidade (Scalable) – um mesmo mecanismo deveria poder ser usado em equipamentos monoprocessados tão pequenos quanto os microcomputadores de mesa, bem como em sistemas multiprocessados com centenas ou milhares de processadores (próxima geração de computadores);
- Coerência (Coherent) – para suportar o uso eficiente de memórias cache em sistemas de memória compartilhada distribuída;
- Interface (Interface) – para oferecer uma arquitetura aberta que permitisse que produtos de diferentes fabricantes pudesse ser interligados em um mesmo sistema.

A seguir será descrito de que forma a SCI dá suporte à coerência de cache. Maiores informações sobre a interface como um todo podem ser encontradas em [Gust92].

O diretório definido pela SCI tem estrutura semelhante ao esquema de diretório encadeado descrito anteriormente, com algumas poucas diferenças [JLGS90]. A lista que une os blocos compartilhados é duplamente ligada; ou seja, todos os blocos nos caches dispõem de dois campos extras que apontam para os caches próximo e anterior na lista, sendo que o bloco na memória dispõe de apenas um campo apontador para o primeiro elemento da lista. O estabelecimento da lista e o atendimento a requisições de blocos para escrita com invalidações, também são feitos de forma semelhante ao diretório em lista ligada simples, sendo que, na SCI pode-se tirar vantagem do fato da lista formada pelos blocos ser duplamente encadeada. Uma diferença a ser salientada, contudo, é a de que, no processo de adição de um elemento na lista, ao invés do controlador de memória transferir a requisição para o cache topo da lista, ele retorna ao cache requisitante o endereço do cache topo com o qual ele refaz a requisição atualizando os apontadores mutuamente; apesar de aumentar o número de mensagem para 4 em lugar de 3 como no esquema encadeado simples, essa decisão evita a possibilidade de *deadlocks* quando as filas de armazenamento dos sinais encontram-se cheias.

Opcionalmente, os processadores podem dispensar o controle do protocolo de coerência para os blocos que não são sujeitos à modificação — nesse caso, nenhuma lista de compartilha-

mento é formada e os blocos são trazidos diretamente da memória permanecendo no cache até que sejam substituídos.

3.5 Observações Finais

O surgimento de uma grande quantidade de esquemas empregando diretórios nos permite afirmar que a solução eficiente da coerência de cache a partir dessas estruturas, se constitui em um dos maiores desafios para a próxima geração de sistemas multiprocessados de memória compartilhada de alto desempenho. Neste capítulo, procuramos cobrir uma ampla faixa de propostas de organização de diretórios, apresentando tanto enfoques centralizados (diretórios totalmente mapeados e limitados) quanto distribuídos (diretórios encadeados). Apesar de existirem outras propostas não apresentadas, elas não passam, em sua maioria, de variações dos esquemas discutidos neste trabalho.

Dentre as soluções apresentadas, os diretórios limitados destacam-se por apresentarem uma boa relação custo/benefício e estarem diretamente sintonizados com o comportamento da execução das aplicações paralelas em sistemas de memória compartilhada. Entretanto, estudos comparativos demonstram um desempenho desse esquemas inferior ao de outras soluções, devido principalmente às ações executadas quando o número de apontadores nas entradas do diretório se mostram insuficientes — um evento de ocorrência pouco freqüente, mas imprevisível. No próximo capítulo propomos um esquema que visa proporcionar um tratamento eficiente da falta de apontadores e elevar o desempenho dos diretórios limitados a níveis compatíveis com os obtidos em esquemas como os totalmente mapeados.

Capítulo 4

Uma Proposta de Diretório Limitado Eficiente

Como visto no capítulo anterior, diversas são as formas de organização do diretório em protocolos de coerência de cache. Poucas, contudo, são passíveis de serem implementadas na prática. O diretório limitado é um exemplo de solução viável. Além de ser flexível à expansão do sistema, sua lógica de controle é simples, podendo ser executada pelos controladores de memória sem causar aumento significativo na latência do acesso à memória.

A maior restrição para a utilização dos diretórios limitados está relacionada com o correto dimensionamento do número de apontadores. Os efeitos da má escolha do número de apontadores podem ser melhor observados através da análise das duas formas básicas de tratamento da ocorrência da falta de apontadores: a *não difusão* e a *difusão* de invalidações. Nos protocolos que não usam a difusão de sinais de invalidação, o subdimensionamento provoca uma espécie de "dança" dos blocos, fenômeno que ocorre quando um bloco é freqüentemente requisitado por um número de processadores maior que o número de apontadores disponíveis. Se essas requisições forem apenas para leitura, o fenômeno ocasionará execução de invalidações e transferências desnecessárias do bloco – um exemplo claro desse fenômeno está presente em boa parte das aplicações paralelas, onde, durante a fase de inicialização, um mesmo bloco pode vir a ser requisitado por todos os processadores do sistema. Nos protocolos que utilizam a difusão de invalidações, todos os processadores podem requisitar um bloco sem a ocorrência da "dança" do bloco. Em caso de necessidade da invalidação desse bloco, contudo, a rede de interconexão é intensamente requisitada; por ser o único meio de comunicação entre os módulos do sistema, ela sofre um grande aumento em seu tráfego. Nos dois tipos de solução o superdi-

mensionamento acarreta um desperdício de espaço, já que, os apontadores em excesso podem ficar sem utilização a maior parte do tempo.

Chaiken [CFKA90] mostra que modificando o código da aplicação paralela utilizando técnicas que eliminam a existência de variáveis com alto grau de compartilhamento ou que são muito referenciadas (usando árvores de combinação [MeSc91], por exemplo, na confecão de primitivas de sincronização), é possível obter-se um desempenho dos diretórios limitados compatível com os diretórios *mapa de bits*. Essa estratégia, todavia, requer um grande esforço por parte do programador na identificação dessas variáveis, na escolha da técnica correta para cada variável e na verificação de que a aplicação se tornou realmente mais eficiente após todas as modificações. Além disso, esse enfoque obriga a revisão, quando não reprogramação, das aplicações já existentes, o que vai contra uma das maiores motivações para o desenvolvimento de sistemas multiprocessados de memória compartilhada: a possibilidade de utilização quase que imediata de aplicações inicialmente desenvolvidas para sistemas monoprocessados.

Com base nessas observações, é possível visualizar a necessidade, nos diretórios limitados, de mecanismos eficientes para a gerência da falta de apontadores que não imponham *overheads* de espaço no armazenamento do diretório nem *overheads* de comunicação no envio de sinais de invalidação, e que não obriguem os programadores a se envolverem com detalhes de arquitetura da máquina para desenvolverem aplicações paralelas eficientes. Neste capítulo apresentamos uma proposta de protocolo para diretório limitado com invalidação por difusão que adota um mecanismo eficiente para o tratamento da falta de apontadores. A descrição do nosso protocolo é precedida de uma revisão do tipo de diretório no qual ele se baseia, o diretorio limitado com invalidação por difusão, onde procuramos identificar os reais responsáveis por suas ineficiências.

4.1 Revisão dos Diretórios Limitados com Invalidação por Difusão

De acordo com o exposto no capítulo anterior, em um diretório limitado com invalidação por difusão a falta de apontadores para um bloco é registrada através de um *bit* na entrada do diretório correspondente ao bloco. O bloco é suprido para tantos quantos forem os processadores que o requisitarem para leitura. No momento em que ocorre uma requisição para escrita ou é recebido um pedido de invalidação das cópias de um bloco, o diretório é verificado e, se não houve falta de apontadores, um sinal de invalidação é enviado a cada um dos processadores indicados pelos apontadores; caso tenha ocorrido a falta, o sinal é enviado pela rede a todos os

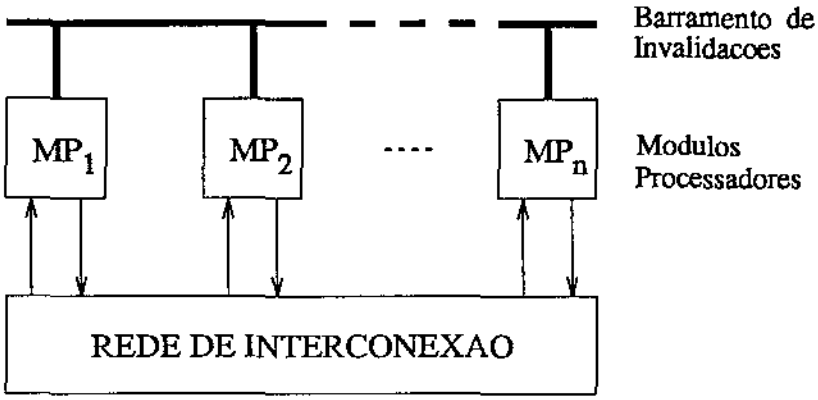

Figura 4.1: Localização do barramento de invalidações e suas respectivas conexões em um sistema multiprocessado.

processadores, independentemente do fato deles possuírem ou não cópia do bloco. Havendo ou não a falta de apontadores, todos os processadores que receberam um comando de invalidação devem enviar para o emissor do comando, um sinal de reconhecimento da invalidação para que, com isso, se tenha a garantia de que todas as cópias foram invalidadas.

Em uma análise imediata dos protocolos de diretório limitado com invalidação por difusão percebe-se que eles são extremamente eficientes no atendimento às requisições para leitura de blocos com falta de apontadores (não há *overheads* de comunicação com sinais de invalidação e há um *overhead* de espaço insignificante decorrente do *bit* de difusão). Em contrapartida, a explosão de sinais que é sentida na rede no momento da difusão das invalidações revela o lado extremamente ineficiente dessa solução. Os sinais enviados a processadores que realmente possuem cópias do bloco são justos e inevitáveis, fazendo parte de qualquer uma das soluções com diretórios apresentadas neste trabalho; já os sinais de invalidação extras, aqueles enviados a processadores sem cópia do bloco, além de desnecessários, competem com os outros sinais válidos do sistema por um recurso extremamente caro como é a rede de interconexão.

O verdadeiro responsável pela ineficiência do processo de difusão, entretanto, é a própia rede de interconexão que não oferece um meio adequado a este tipo de operação. Redes tipo barramento, por outro lado, são bastante eficazes no tratamento de difusões.

Propomos, portanto, um protocolo baseado em diretórios limitados dotado de um barramento especial com funções específicas para o tratamento dos sinais de difusão. O detalhamento deste barramento especial ou barramento de invalidações, assim como os argumentos que justificam a sua adoção são apresentados e discutidos na próxima seção. Suporemos, por hora, que o barramento de invalidações é um dispositivo perfeitamente adequado às necessidades do

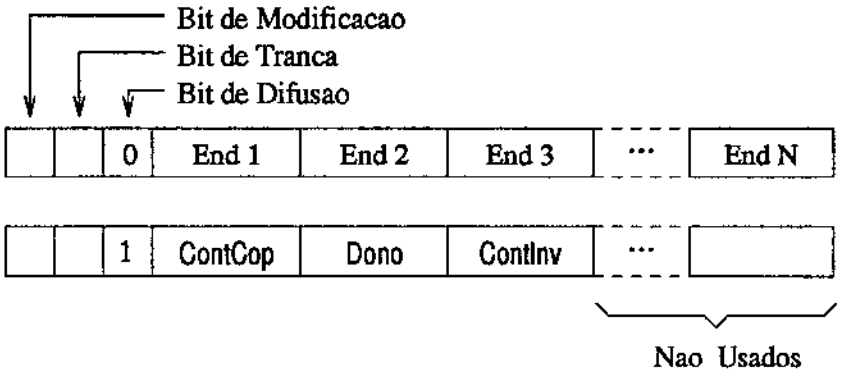

Figura 4.2: Redefinição dos campos no diretório DLE

nosso protocolo. O protocolo proposto é descrito a seguir.

Para efeito de clareza da apresentação, consideraremos que o sistema multiprocessado que usamos como referência é composto por módulos contendo processador, cache e parte da memória global compartilhada dotada de controlador de memória e diretório com entradas referentes aos blocos de memória do módulo. De fato, esse enfoque reflete a tendência observada na organização das atuais arquiteturas multiprocessadas — nada impede, todavia, a utilização do protocolo em arquiteturas onde a memória é organizada em bancos separados dos módulos processadores. A Figura 4.1 localiza o barramento de invalidações nesse modelo de sistema multiprocessado.

4.2 Descrição do Protocolo DLE — Diretório Limitado Eficiente

4.2.1 Organização do Diretório

Em cada banco de memória existe um diretório contendo informações referentes aos blocos de memória do banco. Para cada bloco de memória existe uma entrada no diretório constituída, basicamente, de 1 *bit* de validade, 1 *bit* de difusão, 1 *bit* de "tranca" e 3 ou mais campos com *bits* suficientes para endereçar qualquer um dos caches do sistema. O significado do conteúdo dos campos varia de acordo com o valor do *bit* de difusão: se o *bit* de difusão estiver desligado, os campos endereçam os únicos caches com cópia do bloco (End1, End2, End3, ..., EndN); se o *bit* de difusão estiver ligado, o primeiro campo é um contador que indica quantas cópias do bloco foram requisitadas pelos caches do sistema (CntCop), o segundo campo fica reservado para

armazenar, no caso de uma requisição de alteração do bloco, o endereço do cache requisitante (Dono) e o terceiro campo é também um contador que indica o número de vezes que um mesmo sinal de invalidação foi enviado pelo barramento de invalidações (CntInv). Outros campos na entrada (End4, ..., EndN), se existirem, não são usados quando o *bit* de difusão estiver ligado. A redefinição dos campos em uma entrada do diretório é mostrada na Figura 4.2.

4.2.2 Algoritmo DLE

Localmente, o bloco no cache pode estar em um dos seguintes estados:

- *Inválido* - a cópia da memória ou a de outro cache é a mais atual;
- *Válido* - consistente, podem existir outras cópias do bloco em outros caches;
- *Modificado* - o cache contém a versão mais atual do bloco; a memória e/ou outros caches possuem cópias inválidas.

A mudança de estados pode ocorrer em função de uma operação do processador local (leitura ou escrita) ou induzida por comandos enviados pelo controlador do diretório em resposta a requisições feitas por outros processadores. A Figura 4.3 apresenta o diagrama de estados de um bloco no cache segundo o protocolo DLE.

As ações a seguir são efetuadas a depender da operação que está sendo executada sobre o bloco no cache (leitura, escrita ou substituição) e seu respectivo estado.

1. **LEITURA** (bloco *válido* ou *modificado* no cache)

   A operação de leitura é atendida localmente.

2. **LEITURA** (bloco *ausente* ou *inválido* no cache)

   Uma requisição do bloco para leitura é feita ao banco de memória correspondente. Duas situações podem ocorrer:

   (a) *Bit* de modificação desligado

   O bloco de memória está consistente. Mais uma vez, duas situações podem ocorrer:

i. *Bit* de difusão desligado

Possivelmente existe apontador disponível na entrada do diretório correspondente ao bloco. O controlador de memória envia uma cópia do bloco, que toma o estado *válido* no cache requisitante, e aloca um dos apontadores para apontar para esse cache. Caso não exita mais apontador disponível, o *bit* de difusão é ligado e CntCop é inicializado com o número de apontadores que a entrada pode suportar adicionado de 1.

ii. *Bit* de difusão ligado

O controlador de memória envia uma cópia do bloco e incrementa CntCop na entrada do diretório correspondente ao bloco. A cópia toma o estado *válido* no cache requisitante.

(b) *Bit* de modificação ligado

O bloco de memória está inconsistente. Na entrada do diretório correspondente ao bloco, Dono indica qual o último cache que requisitou autorização para modificação (algoritmo de escrita a seguir) que, por conseguinte, contém a versão mais atual do bloco. O controlador de memória envia então um sinal para Dono atualizar o bloco na memória e enviar uma cópia do bloco para o cache requisitante. Tanto a cópia do bloco em Dono quanto no cache requisitante assumem o estado *válido*. No diretório, o *bit* de modificação é desligado e o primeiro apontador é atualizado com o endereço do cache requisitante (o segundo apontador já contém o endereço de Dono); os apontadores restantes ficam liberados.

3. **ESCRITA** (bloco *modificado* no cache)

A operação de escrita é efetuada imediatamente na cópia local do bloco.

4. **ESCRITA** (bloco *válido*, *ausente* ou *inválido* no cache)

Antes de modificar o bloco, é necessário obter uma autorização. Um sinal é enviado ao banco de memória correspondente indicando o desejo de alteração do bloco por parte do cache; caso o bloco no cache esteja *ausente* ou *inválido*, um pedido de envio de cópia é associado ao pedido de autorização. Ao final da operação, o bloco no cache assume o estado *modificado*. Quando a requisição da autorização chega no controlador do diretório, duas situações podem ocorrer:

(a) *Bit* de modificação desligado

O bloco de memória está consistente. Mais uma vez, duas situações podem ocorrer:

i. *Bit* de difusão desligado

Um sinal de invalidação é enviado pela rede de interconexão para cada um dos caches endereçados pelos apontadores da entrada do diretório correspondente ao bloco. O *bit* de difusão é ligado e CntCop é atualizado com o número de caches para os quais foram enviados os sinais de invalidação; adicionalmente, Dono passa a conter o endereço do cache que requisitou a autorização. A partir desse momento, o controlador diretório é liberado para atender a pedidos referentes a outros blocos. Nos caches que receberam o sinal de invalidação, a cópia do bloco envolvido assume o estado *inválido* e um sinal de reconhecimento/execução da invalidação é remetido para o banco de memória que solicitou a invalidação. Para cada sinal remetido, CntCop é decrementado e, ao alcançar o valor zero, o *bit* de modificação é ligado sendo o sinal de autorização enviado para Dono; caso uma cópia do bloco tenha sido requerida, ela é enviada juntamente com a autorização.[1]

ii. *Bit* de difusão ligado

O diretório não tem controle de quais caches possuem cópias do bloco, apenas de quantos eles são através de CntCop. O endereço do bloco é colocado em um *buffer* de invalidações para ser difundido juntamente com o endereço do banco de memória pelo barramento de invalidações. Dono passa a conter o endereço do cache que requisitou a autorização. A partir desse momento, o controlador do diretório é liberado para atender a pedidos referentes a outros blocos. O pacote enviado pelo barramento de invalidações alcança todos os processadores e é armazenado em um *buffer* de chegada (a seção Barramento de Invalidações a seguir, detalha como são tratadas as exceções, ou seja, quando nem todos os módulos do sistema recebem o pacote de invalidação devido a situações como *buffer* de chegada cheio, por exemplo). Os caches que efetivamente possuem cópias do bloco, efetuam a invalidação e remetem um sinal para o banco de memória que solicitou a invalidação (seu endereço fora enviado juntamente com o pacote de invalidação) através da rede de interconexão indicando o seu reconhecimento/execução. Para cada sinal de reconhecimento/execução recebido, CntCop é decrementado e, ao alcançar o valor zero, o *bit* de modificação é ligado sendo o sinal de autorização enviado para Dono; caso uma cópia do bloco tenha

[1] Uma variação possível, visando aliviar a carga de mensagens a serem tratadas pelo controlador do diretório, consiste em enviar, junto com as invalidações, o endereço do futuro dono do bloco; assim, as confirmações das invalidações seriam remetidas diretamente para serem tratadas pelo próprio dono, aproveitando o tempo que ele está ocioso, já que não pode prosseguir antes que todas as cópias sejam invalidadas.

sido requerida, ela é enviada juntamente com a autorização. O *bit* de difusão é desligado.

(b) *Bit* de modificação ligado

Na entrada do diretório correspondente ao bloco, Dono indica qual o último cache que requisitou autorização para modificação. Um sinal é então enviado a Dono para que ele invalide sua cópia e envie o bloco para o cache requisitante; ao receber a cópia, o cache requisitante avisa o controlador que atualiza Dono para o novo possuidor da versão mais atual.

5. **SUBSTITUIÇÃO** (bloco *inválido* no cache)

O bloco no cache está inconsistente e pode ser substituído imediatamente, dispensando qualquer operação extra.

6. **SUBSTITUIÇÃO** (bloco *válido* ou *modificado* no cache)

Como o bloco substituído deixará de fazer parte do elenco de blocos no cache, o diretório deve ser atualizado. Duas situações podem ocorrer:

(a) *Bit* de modificação desligado

Não há necessidade de atualizar a memória. Mais uma vez, duas situações podem ocorrer:

i. *Bit* de difusão desligado

O apontador correspondente ao cache que está efetuando a substituição é liberado.

ii. *Bit* de difusão ligado

CntCop é decrementado e, caso tenha alcançado zero, o *bit* de difusão é desligado.

(b) *Bit* de modificação ligado

O bloco no cache é enviado para a memória a fim de atualizar o bloco de memória (*copy-back*). Em seguida, o *bit* de modificação é desligado e os apontadores liberados.

4.2.3 Sincronização

O protocolo DLE foi especificado de forma que, em nenhum instante, o controlador do diretório fica impedido de atender a outras requisições enquanto está aguardando a execução de operações que, apesar de estarem relacionadas com o atendimento de uma requisição em andamento,

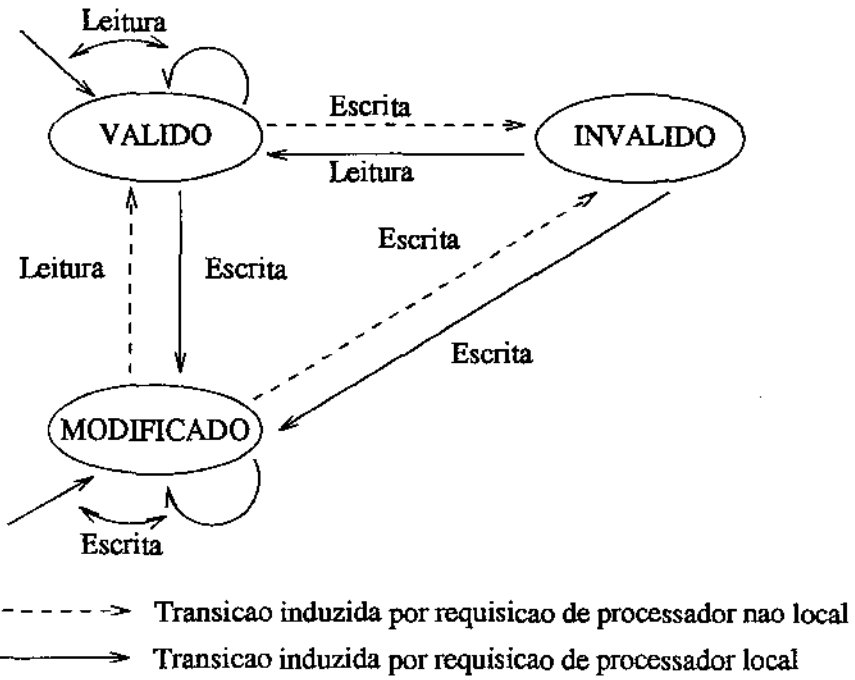

Figura 4.3: Diagrama de estados de um bloco no cache segundo o protocolo DLE

independem dele; é o caso, por exemplo, das operações envolvidas no processo de invalidação das cópias de um bloco em uma requisição de escrita. Por um lado, essa decisão de projeto torna possível a obtenção do desempenho máximo por parte do controlador do diretório, uma vez que, ele fica habilitado a atender a diversas requisições de maneira sobreposta; além disso, ela se adequa perfeitamente à forma de operação, também assíncrona, das redes com chaves multiestágios — onde normalmente não existe o estabelecimento de conexões, mas sim a troca de mensagens na forma de pacotes. Por outro lado, essa forma de operação requer mecanismos eficientes de sincronização que garantam a execução completa, consistente e sem conflitos de cada uma das requisições recebidas. No protocolo DLE, esses mecanismos estão representados basicamente por dois objetos: o *bit* de tranca e o contador de invalidações ContInv.

O *bit* de tranca tem como função evitar que duas requisições distintas atuem simultaneamente sobre um mesmo bloco. Ele é ligado antes do controlador do diretório efetuar qualquer operação sobre um bloco (leitura, escrita, tratamento de pedidos de invalidação, etc.) para indicar que uma requisição sobre aquele bloco está pendente e que outras requisições sobre o mesmo bloco devem ser inibidas até que esse *bit* seja desligado. O *bit* de tranca resolve, portanto, a maioria dos problemas resultantes de requisições conflitantes, como é o caso de requisições de leitura a um bloco que está sofrendo invalidações decorrentes de uma requisição para escrita anterior. As requisições conflitantes podem ser tratadas de duas formas:

1. Toda requisição direcionada a blocos com o *bit* de tranca ligado é rejeitada, obrigando o cache requisitante a refazer a requisição;

2. O controlador armazena localmente as requisições que podem ser atendidas assim que o *bit* de tranca seja liberado, como é o caso do exemplo acima[2] e rejeita as que são realmente inviáveis, como é o caso de duas requsições simultâneas para escrita — nesse caso, uma é atendida e a outra é rejeitada.

O contador de invalidações (ContInv) por sua vez, tem por objetivo auxiliar na detecção e correção de possíveis falhas na operação do barramento de invalidações. Considere, por exemplo, a situação na qual uma difusão pelo barramento de invalidações não ocorreu de maneira perfeita e nem todos os processadores registraram o comando de inavalidação. Isso pode acontecer por razões que vão desde a falta de espaço no *buffer* de chegada de um processador que, coincidentemente, possui uma cópia do bloco a ser invalidado — uma situação pouco provável, porém possível — até uma falha de operação do próprio barramento. Nesses casos, o aviso de reconhecimento/execução da invalidação não é remetido, o contador de cópias (ContCop) do bloco nunca alcança o valor zero e o processador que efetuou a requisição para escrita do bloco nunca recebe a autorização; além disso, o controlador fica impossibilitado de atender a qualquer outra requisição direcionada a esse bloco.

Para evitar esse tipo de *deadlock*, o processador que enviou a requisição de alteração deve, decorrido um certo período de tempo sem o retorno da autorização por parte do controlador, ressubmeter a requisição. Dessa forma, na tentativa de atender a esse novo pedido, o controlador reenviará o comando de invalidação incrementando ContInv. O processo se repete até que a requisição seja atendida, atestando o sucesso das invalidações, ou que ContInv alcançe um valor predeterminado, forçando o controlador a executar ações emergenciais que vão desde a difusão de comandos de invalidação pela própria rede de interconexão, até a indicação de uma possível falha de *hardware* do barramento de invalidações. Nesse último caso, é possível obter-se um bom nível de tolerância a falhas, se, a partir do momento da detecção da falha, o controlador do diretório iniciar a execução de um protocolo alternativo do tipo *não difusão* até que o problema no barramento de invalidações seja sanado.

[2]Um *bit* auxiliar na entrada do bloco pode indicar a existência de operações aguardando a liberação.

4.3 Barramento de Invalidações

Trata-se de um barramento que segue os mesmos princípios de funcionamento de um barramento padrão, com as vias de sinais e circuitos lógicos de controle habituais, mas que é voltado exclusivamente para a difusão de invalidações. A princípio, o comportamento serializado com que o acesso a um barramento é feito, pode levar à conclusão de que ele rapidamente alcança o ponto de saturação com um número reduzido de dispositivos conectados, como aliás já fora afirmado neste trabalho. Entretanto, diversas razões nos levam a crer que, se utilizado da forma especializada como está sendo proposta, o ponto de saturação do barramento, apesar de existir, está muito além dos registrados quando o barramento é utilizado como rede de interconexão, onde ele é obrigado a suportar uma gama variada de formas de operação e requisições. Algumas dessas razões estão relacionadas a seguir:

- Diferentemente de um barramento padrão, o barramento de invalidações, como o nome indica, é usado exclusivamente para o envio de sinais específicos de invalidação, operação que pode ser efetuada em um número fixo e reduzido de ciclos. Em um barramento padrão, por outro lado, um acesso para uma leitura ou *copy-back* de um bloco, por exemplo, requer um ciclo de endereçamento seguido de tantos ciclos quantos forem neccesários para a transferência do bloco (durante esse período, o barramento fica inabilitado para o atendimento às requisições de outros componentes do sistema).

- A necessidade do envio de sinais de invalidação pelo barramento está restrita aos casos em que houve falta de apontadores no diretório e não a todos os casos de escrita a blocos compartilhados. Como já discutido nesse trabalho, espera-se que essas sejam situações infreqüentes.

- A difusão do sinal de invalidação é feita de forma unidirecional, do controlador do diretório para os caches do sistema; a acusação de recebimento e execução da invalidação é feita através da rede de interconexão. Em um barramento padrão, o processo de comunicação é normalmente feito de forma bidirecional, com uma requisição sendo enviada em um sentido e respondida em outro.

- Como o barramento de invalidação é usado de forma específica — para a difusão de um pacote de *bits* de tamanho predefinido —, ele pode ser projetado para otimizar esse tipo de operação fazendo uso, por exemplo, de *buffers* nos *drivers*, operação *pipeline*, etc.

A seguir, desenvolvemos uma expressão que pode ser usada na previsão do ponto de saturação do barramento de invalidações a partir da sua taxa de transferência e da velocidade dos processadores a ele conectados.

4.3.1 Estimativa de Saturação

A taxa de transferência (*bandwidth*) fixa transforma o barramento em um recurso limitante do desempenho em sistemas que o utilizam de forma compartilhada. Na subseção anterior afirmamos que o ponto onde um barramento especializado começa a limitar o desempenho está muito além dos observados em barramentos não especializados. A questão que surge, por conseguinte, é qual seria o ganho obtido com a utilização especializada? Especificamente no caso do barramento de invalidações, qual a estimativa de expansão do sistema, em número de processadores, antes que se estabeleça o congestionamento dessa via? O desenvolvimento de uma expressão a partir do resultado de estudos recentes dos padrões de invalidação observados em aplicações paralelas reais pode nos dar uma idéia da resposta a essas questões.

Primeiramente, vamos desenvolver uma expressão que nos informe o número médio de requisições à memória que ocasionam a necessidade do uso do barramento de invalidações. Usaremos como referência um sistema com N processadores, dotado de diretórios limitados com 4 apontadores por entrada[3].

Seja ω a probabilidade de um processador qualquer efetuar uma referência à memória para escrita a um bloco compartilhado no início de um ciclo de memória e seja β a probabilidade de que uma dessas escritas envolve um bloco em cuja entrada o *bit* de difusão encontra-se ligado, ou seja, o número de cópias do bloco em caches do sistema é maior que 4. O número esperado de sinais enviados pelo barramento é dado, portanto, por $\rho \times \omega \times \beta$, onde ρ é o desempenho máximo do sistema, dado por N multiplicado pelo número máximo de instruções que cada processador pode executar por unidade de tempo. Logo, a seguinte relação, entre a taxa de tranferência do barramento de invalidações $\mathcal{TT}$ e a expressão acima, deve ser válida para obter-se o máximo desempenho:

$$\mathcal{TT} \geq \rho \times \omega \times \beta \tag{4.1}$$

[3]Esse parece ser um número razoável que atende bem ao compromisso "espaço ocupado pelo diretório × probabilidade de falta de apontadores" discutido no início deste capítulo e que vem sendo indicado como adequado por estudos como [CFKA90], [SH91] e [GuWe92].

Tabela 4.1: Valores típicos de ω e β obtidos por [GuWe92] a partir de aplicações paralelas reais.

Aplicação	Num. CPU	Data Ref. milhões	Escrita Comp. milhões	%(ω)	Invalidações (Apt > 4) %(β)
MaxFlow	32	25,5	2,2	8	3
Water	32	48,0	1,0	2	1
PTHOR	32	16,6	0,9	5	4
LocusRoute	32	18,5	1,0	5	6

A Tabela 4.1 traz os valores típicos de ω e β obtidos por Gupta e Weber [GuWe92] a partir das referências à memória geradas em simulações de 4 aplicações paralelas reais. Apesar dos dados dessa tabela estarem relacionados a um sistema com 32 processadores, os autores afirmam que o número de invalidações manteve-se constante quando a simulação foi repetida para 8 e 16 processadores; adicionalmente, eles não acreditam que esses dados sofram mudanças significativas em sistemas com um número elevado de processadores.

Para calcularmos ρ na inequação 4.1 usaremos o maior produto $\omega \times \beta$ da Tabela 4.1, dado pela aplicação *LocusRoute* ($\omega = 0,05$ e $\beta = 0,06$) e tomaremos como referência para o barramento de invalidações as características do FutureBus+, um líder na tecnologia atual em termos de desempenho [Andr89]. O FutureBus+ é capaz de fornecer uma taxa de transferência da ordem de 100 MTransf./seg. (64 *bits*/transf.) em seu modo pacote; nesse modo, a necessidade do estabelecimento de conexão (*handshaking*) antes do envio dos dados é eliminada, obtendo-se, com isso, altas taxas de transferência[4].

Temos portanto:

$$100M \geq \rho \times 0,05 \times 0,06$$

$$\rho \leq 33.333M \tag{4.2}$$

Para calcular o número máximo de processadores que obedece a relação 4.2, vamos tomar o desempenho máximo, da ordem de 2,5 MIPS [Taba90], dos processadores usados em alguns sistemas multiprocessados com rede de interconexão multiestágios, como é o caso do BBN

[4]Esse modo de operação se adequa perfeitamente à forma de operação do barramento de invalidações, onde um pacote contendo o enderço do bloco a ser invalidado é difundido pelo barramento para todos os outros processadores do sistema; considerando-se pacotes de 64 *bits* tem-se uma capacidade de endereçamento de 2^{64} blocos.

Butterfly (68020) e do IBM RP3 (IBM ROMP – um processador próprio). Além disso, vamos admitir que a execução de cada instrução gera uma referência à area de dados. Assim,

$$2,5M \times N \leq 33.333M$$

$$N \leq 13.333$$

Esse valor máximo de N ultrapassa, em mais de uma ordem de grandeza, o número máximo de processadores projetados para aqueles sistemas, 256 para o BBN Butterfly e 512 para o IBM RP3. Apesar desse resultado não ser conclusivo, ele nos dá uma boa indicação da viabilidade da utilização do barramento de invalidações nas futuras gerações de sistemas multiprocessados.

4.3.2 Restrições de Comprimento, Alimentação e Arbitragem

A conexão de um número muito grande de dispositivos a um único barramento encontra dois fatores de limitação: a extensão ou comprimento desse barramento e o nível de corrente do sinal que irá circular por ele. Por um lado, um barramento muito extenso, operando a uma freqüência muito alta, está sujeito à captação de ruído e a outros efeitos eletromagnéticos que causam distorções e atrasos no sinal que está sendo propagado; por outro lado, o nível de corrente necessário para que o sinal seja percebido por todos os pontos de conexão pode chegar a valores que inviabilizam a construção prática de barramentos densamente compartilhados [Gust84]. Além desses dois fatores, um outro que deve ser igualmente considerado é a complexidade dos circuitos de arbitragem — responsáveis por decidir quem tem o direito de colocar o sinal no barramento a cada instante — que aumenta linearmente com o número de conexões.

A tecnologia na área de barramentos tem evoluído na direção de soluções eficazes para os dois primeiros problemas. Uma descrição aprofundada do enfoque adotado no FutureBus na solução desses problemas pode ser encontrada em [Bala84].

Já o problema da arbitragem não parece ter uma solução viável para um número elevado de conexões. A maioria dos barramentos recentes, incluindo o FutureBus, adota um mecanismo de decisão combinacional que basea-se em prioridades mas que restringe o número máximo de dispositivos que podem ser conectados. Felizmente, graças à forma assíncrona com que o algoritmo DLE trata o envio de invalidações pelo barramento, é possível projetar uma estrutura de barramento com sub-barramentos que viabiliza a existência de um grande número de pontos de conexão. Um exemplo dessa estrutura poderia ser obtido através da divisão dos processadores

em grupos, com cada grupo conectado a um barramento particular e cada barramento particular sendo conectado a um barramento principal conforme a Figura 4.4. Todo sinal de invalidação enviado pelo barramento particular alcançaria todos os outros processadores a ele conectados e também o ponto de conexão com o barramento principal; a partir do barramento principal o sinal seria então distribuído para todos os outros processadores. Essa solução além de sanar os 3 problemas discutidos acima, permite que o sistema parta de uma configuração inicial e possa ser expandido gradativamente.

4.3.3 Utilizações Anteriores

A idéia de utilizar um barramento com funções específicas de invalidação não é nova. De fato, ela fora originalmente usada na entitulada **solução clássica** para o problema da coerência de cache, mencionada no capítulo 2. De acordo com Censier e Feautrier [CeFe78], essa solução é encontrada nos primeiros biprocessadores que adotavam política *write-through* para a atualização da memória [5].

> *"Para garantir a coerência, cada cache é conectado à uma via de sinais auxiliar, sobre a qual todas as outras unidades enviam o endereço do bloco a ser modificado. Cada cache monitora permanentemente essa via e executa o algoritmo de busca no diretório de cache para cada endereço recebido. Para os endereços encontrados, o* bit *de validade do bloco correspondente é desligado."*

O envio de sinais de invalidação à cada operação de escrita de um processador em conseqüência da política *write-through*, contudo, confina a aplicação da solução clássica a sistemas com um número reduzido de processadores. Além disso, existe uma pequena probabilidade de ocorrência de inconsistências, caso operações de leitura sejam atendidas pelo cache entre o momento em que uma modificação é feita e a invalidação é efetuada no bloco envolvido.

Apesar da semelhança funcional, diversos fatores diferenciam os barramentos de invalidações do protocolo DLE e da solução clássica. A restrição do uso do barramento apenas nos casos de blocos com falta de apontadores reduz, como visto, enormemente o tráfego induzido por cada processador nessa via. O emprego de tecnologia recente na área de barramentos

[5] Entre os sistemas comerciais que adotaram a solução clássica estão os biprocessadores IBM 370/168 e o 3033, além dos sistemas monoprocessados, como o VAX11/780, onde ela é usada para garantir a coerência nos acessos à memória entre o processador central e os processadores de I/O (cf. [ArBa84]).

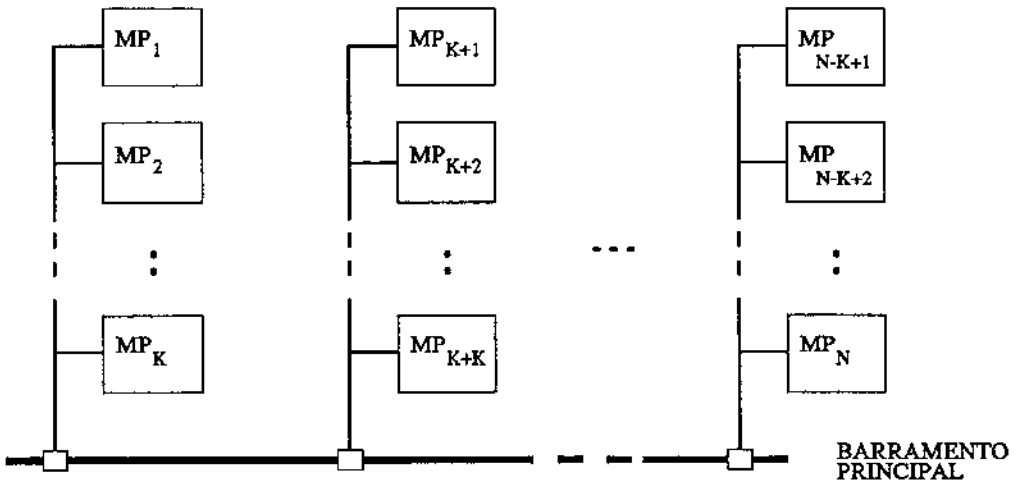

Figura 4.4: Organização sugerida para o barramento de invalidações como forma de solução para as restrições de Comprimento, Alimentação e Arbitragem.

também contribui para um desempenho esperado acentuadamente melhor do protocolo DLE. Finalmente, a existência do diretório na intermediação de uma operação de escrita a blocos compartilhados elimina a probabilidade de ocorrência de inconsistências; a serialização das operações é garantida, já que, só é dada a autorização a um cache para prosseguir com a modificação de um bloco quando todos os outros confirmarem a execução da invalidação de suas cópias locais.

Por outro lado, algumas características de implementação permanecem comuns às duas soluções. Entre elas, a inevitável necessidade de duplicação dos diretórios dos caches (para que o algoritmo de monitoração do barramento de invalidações não cause interferências no atendimento do cache às requisições dos processadores) e, também, de pequenos *buffers* em cada ponto de interconexão com o barramento para acomodar instantes de pico no tráfego dos sinais de invalidação.

4.4 Observações Finais

O resultado líquido de se ter uma via auxiliar para o envio dos sinais de invalidação no protocolo DLE é manter limitada a carga que esse tipo de operação impõe na rede de interconexão independentemente do comportamento da aplicação. Se em uma aplicação específica, o número máximo de cópias compartilhadas de cada bloco nunca atinge a quantidade de apontadores disponíveis no diretório, tem-se garantido que o *overhead* de comunicação para o tratamento

de invalidações é mantido a níveis mínimos. Nas aplicações que, por outro lado, esse número tende a ultrapassar a quantidade de apontadores, o mecanismo do barramento de invalidações entra em ação amortecendo essa variação de comportamento e, mais uma vez, garantindo níveis mínimos de *overhead* na rede.

O uso de trancas (*locks*) em operações de sincronização também são beneficiadas, já que, o bloco que contém a variável de tranca pode ser compartilhado por tantos quantos forem os processadores que desejarem disputá-la, sem nenhum *overhead* para a rede.

Pode-se ainda dizer que, atualmente, a tecnologia dos barramentos encontra-se bastante desenvolvida. Sua utilização nos futuros sistemas multiprocessados, porém, está restrita ao desempenho de funções especializadas, não mais como principal via de comunicação; a difusão de sinais, por exemplo, é uma das funções que o barramento desempenha melhor que qualquer outro tipo de rede de interconexão.

Neste capítulo, apresentamos a proposta do protocolo DLE e discutimos, com justificativas, porque o consideramos viável. Nos próximos capítulos descreveremos um ambiente de simulação, onde procuraremos validar o protocolo DLE e obter resultados de desempenho a partir de comparações com outras propostas.

Capítulo 5

Ambiente de Simulação

A fim de avaliar o desempenho do protocolo DLE apresentado no capítulo anterior juntamente com outros protocolos baseados em diretórios discutidos no capítulo 3, foi desenvolvido um ambiente de simulação onde procurou-se reproduzir parte do subsistema de gerenciamento de memória de um sistema multiprocessado de memória compartilhada típico. A figura 5.1 mostra os principais componentes deste ambiente e como eles estão interligados. Neste capítulo serão discutidos os pontos mais relevantes desse ambiente de simulação — ou, simplesmente, simulador — sendo apresentado também as decisões que influenciaram o seu desenvolvimento.

Na figura 5.1, cada módulo processador é constituído de uma unidade de processamento, um controlador de cache, um controlador de diretório e parte da memória global compartilhada. Os módulos se comunicam através de uma rede de interconexão do tipo chaves multiestágios via um servidor de mensagens embutido no módulo. Cada um desses componentes [1] é criado dinamicamente durante a inicialização do simulador. Todo o sistema funciona sob a gerência de um configurador global que, além de estabelecer as características de funcionamento dos componentes, assegura a correta interligação — interna e externa — dos módulos processadores.

[1] Por componente entenda-se objetos que participam ativamente do processo de simulação efetuando e/ou atendendo a requisições de outros componentes, de acordo com seus estados correntes. Esse termo será bastante utilizado com esse significado no decorrer deste capítulo.

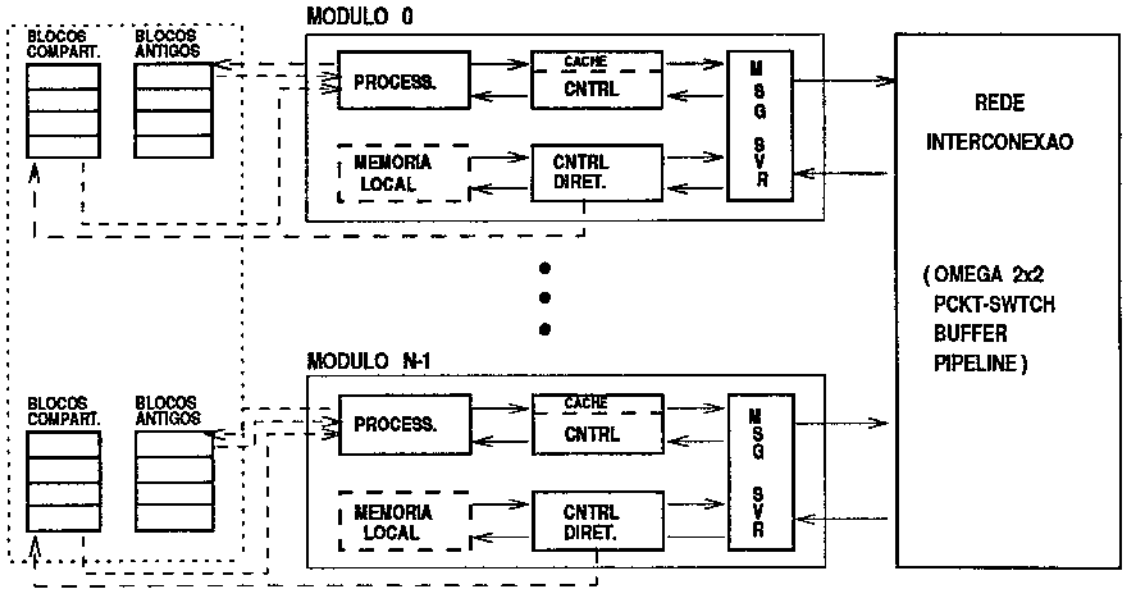

Figura 5.1: Diagrama do ambiente de simulação

5.1 Modelo de Simulação e Ambiente de Implementação

As características e forma de funcionamento dos diversos componentes do simulador apontaram para o modelo de simulação *discreto orientado a eventos* como o mais indicado para basear o seu desenvolvimento. De forma resumida, a simulação discreta aplica-se a modelos onde alterações no estado das variáveis das quais ele depende se dá discretamente em pontos específicos do tempo simulado (ao contrário da simulação contínua onde as variáveis dependentes podem variar continuamente ao longo do tempo). Em sua forma orientada a eventos, essas variações são ocasionadas, ainda, por eventos que ocorrem no decorrer do tempo simulado; entre eventos, o estado das variáveis não se altera [Neel87], [Soar90]. No simulador em questão, por exemplo, a execução de um acesso à memória — referência — feito por um processador a um endereço não disponível no cache é um evento que dá origem a outros eventos — no caso, a execução de um pedido do bloco ausente à memória efetuado pelo controlador de cache.

O tratamento de um evento além de ocasionar mudanças no estado corrente do componente, está sujeito a um atraso que é característico do componente e que é responsável pelo consumo do tempo simulado. Cada unidade do tempo simulado pode corresponder à qualquer unidade do tempo real. No simulador, uma unidade de tempo simulado corresponde a um ciclo de máquina e os atrasos em cada componente são múltiplos dessa unidade.

A implementação do simulador foi inteiramente feita na linguagem C++, com o auxílio de uma biblioteca de corrotinas que fornece facilidades para o desenvolvimento de simulações discretas orientadas a eventos, disponível em ambiente UNIX e estações de trabalho SUN[SUN89]. A Tabela 5.3 no final deste capítulo traz a relação dos programas que compõem o simulador.

Segundo essa biblioteca, cada corrotina é vista como uma tarefa e diversas tarefas podem estar executando simultaneamente, compartilhando um mesmo espaço de endereçamento. Um relógio e um escalonador internos fazem o gerenciamento de quais tarefas devem ser executadas a cada instante do tempo simulado. Quando em execução, porém, somente a própria tarefa pode liberar o processador para a alocação de outras tarefas (modelo não preemptivo). Uma tarefa pode ser posta em execução em tempos predeterminados ou pela ocorrência de um evento pelo qual ela estava aguardando.

Cada vez que uma tarefa libera o processador, o escalonador verifica se existe mais alguma tarefa para ser executada, antes de avançar o relógio para o instante mais próximo no futuro onde existam tarefas escalonadas. O processo se repete até que as tarefas alcancem um estado final ou que sejam forçadas, por uma outra tarefa, ao estado de término. [2]

Cada componente do simulador está implementado como uma tarefa. Dessa forma, a depender do número de módulos processadores configurado no simulador [3], milhares de tarefas podem ser escalonadas a cada avanço do tempo simulado. As mensagens trocadas entre os componentes permanecem armazenadas em estruturas de filas até o momento de serem tratadas. Essas filas formam a estrutura básica de comunicação entre os componentes. A chegada de uma mensagem, a depender do estado em que se encontra o componente que a recebe, pode ser um evento que dispara a sua execução. Os formatos das mensagens assim como o modelo de funcionamento dos principais componentes do simulador são descritos a seguir.

[2] Maiores informações sobre a biblioteca — *task library* — e sobre outras plataformas onde ela se encontra disponível podem ser encontradas na referência citada.

[3] Atualmente o simulador comporta configurações entre 2 e 256 módulos (em incrementos de potência de 2) sendo que essa limitação pode ser facilmente estendida a até 65536.

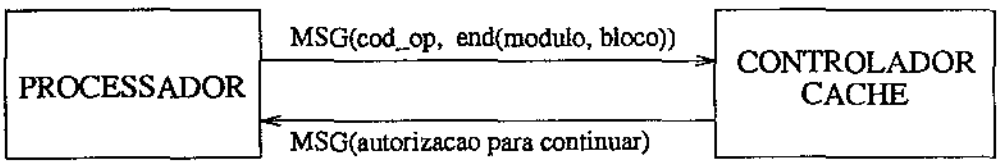

Figura 5.2: Formato das mensagens da CPU

5.2 Descrição dos Componentes

5.2.1 Processador

O processador é o elemento gerador dos eventos que irão ocasionar todos os outros eventos do processo de simulação, ou seja, as referências a endereços de memória que serão enviadas ao controlador de cache. Cada referência corresponde a uma mensagem cujo formato está detalhado na figura 5.2.

Basicamente o conteúdo da mensagem é formado pelo código da operação (leitura ou escrita) e o endereço requisitado. O endereço, por sua vez, é subdividido em dois campos: módulo de memória e bloco na memória. As referências geradas pelos processadores são sintéticas e o processo de geração está descrito na seção 5.3.

A cada mensagem enviada pelo processador corresponde uma outra enviada como resposta pelo controlador de cache autorizando a emissão de uma nova mensagem, simulando assim, o atendimento à mensagem anterior. O somatório dos intervalos de tempo em que o processador fica parado, impossibilitado de efetuar trabalho útil, aguardando a resposta do controlador de cache será a principal medida usada nas avaliações de desempenho dos protocolos de coerência avaliados.

5.2.2 Controlador de Cache

O controlador de cache tem três funções básicas no simulador:

1. gerencia a alocação e o estado (*válido/inválido*, *modificado/não modificado*) dos blocos no cache;

2. recebe e trata as mensagens enviadas pelo processador efetuando a busca, armazenamento e substituição (se necessário) quando a cópia solicitada não se encontra no cache local;

3. recebe e trata as mensagens enviadas por outros componentes diferentes do processador; um exemplo dessa classe de mensagens é a invalidação de uma cópia no cache local enviada pelos controladores de diretórios.

O cache adotado no simulador é do tipo mapeamento direto, com política de atualização de memória *copy-back*.

A cada instante o controlador do cache pode estar em um dos seguintes estados que ocorrem em função das solicitações do processador:

- □ **OCIOSO** - nenhum pedido está sendo atendido no momento; qualquer mensagem será tratada imediatamente após a sua chegada;
- □ **LENDO** - houve uma ausência e uma busca do bloco para leitura está em execução;
- □ **ESCREVENDO** - houve uma ausência e uma busca do bloco para escrita está em execução;
- □ **INVALIDANDO** - houve um pedido para escrita e uma cópia válida do bloco está presente no cache; a fim de manter a coerência, o controlador fica aguardando que as outras cópias do bloco sejam invalidadas.

O pedido do processador só é atendido quando o controlador muda de qualquer um dos estados para o estado **OCIOSO**, indicando que nenhuma operação de leitura, escrita ou invalidação se encontra pendente. Estando em um desses últimos estados, porém, o controlador está apto a tratar mensagens oriundas dos outros componentes do simulador. Dessa forma, é possível atender a um pedido de invalidação de um bloco no cache local enquanto o controlador do cache está aguardando um atendimento a um pedido de leitura.

O formato e o conteúdo das mensagens trocadas entre os controladores de cache e os controladores de diretório está especificado a seguir, juntamente com a descrição do componente servidor de mensagens.

5.2.3 Controlador de Diretório

Semelhantemente ao controlador de cache, o controlador do diretório é o responsável pela gerência dos estados dos blocos contidos no módulo de memória ao qual ele está associado.

Como visto no capítulo 3, a estrutura que armazena os estados dos blocos é conhecida como diretório. Atualmente, o simulador está habilitado a operar com três tipos de estruturas de diretórios: *totalmente mapeado*, *limitado padrão* e o DLE (proposto no capítulo anterior). A parte comum a essas três estruturas é a existência, em cada entrada do diretório, de dois *bits*, um de "tranca", que sinaliza que nenhuma operação pode ser efetuada sobre o bloco enquanto ele estiver ligado, e um outro que indica se a cópia do bloco na memória está atualizada ou não.

Os blocos de memória associados às entradas nos diretórios são na realidade fictícios. Não existe nenhum componente no simulador que represente a memória local em cada módulo. Para efeito de simulação só nos interessa estabelecer os atrasos envolvidos na leitura de blocos da memória. Esse atraso é adicionado ao tempo de tratamento das mensagens enviadas ao controlador de diretório que ocasionam acesso à memória (pedidos de leitura de blocos, por exemplo).

O controlador de diretório opera, a cada ciclo, tratando mensagens que se constituem em solicitações de leitura ou escrita de blocos ou em respostas a solicitações enviadas pelo próprio controlador a outros componentes. Por ser um componente que gerencia informações centralizadas (os blocos na memória e o próprio diretório), optou-se, no desenvolvimento do simulador, pela inexistência de estados internos ao controlador que induzissem à existência de pendências. Ou seja, tomando-se que toda mensagem de pedido envolve um bloco, todas as ações relacionadas com o seu atendimento ou são executadas em um único ciclo do controlador (leitura de um bloco atualizado na memória, por exemplo), ou se estabelece um processo que envolve a troca de várias outras mensagens de serviço entre o controlador de diretório e os outros componentes (leitura de um bloco não atualizado na memória, por exemplo); no intervalo entre o envio de uma dessas mensagens de serviço e o recebimento de uma provável resposta, o controlador fica livre para tratar mensagens que envolvam outros blocos de memória. Da mesma forma, pedidos relacionados a blocos que estão sofrendo ações de consistência são recusados e devem ser ressubmetidos. O *bit* de "tranca" é usado para indicar, quando ligado, os blocos para os quais as mensagens devem ser recusadas.

O conjunto de mensagens que são trocadas entre os controladores de cache e os controladores de diretórios constituem o protocolo de comunicação que é comum às três estruturas de diretórios disponíveis no simulador. O protocolo de comunicação implementado no simulador é apresentado na seção 5.4 a seguir.

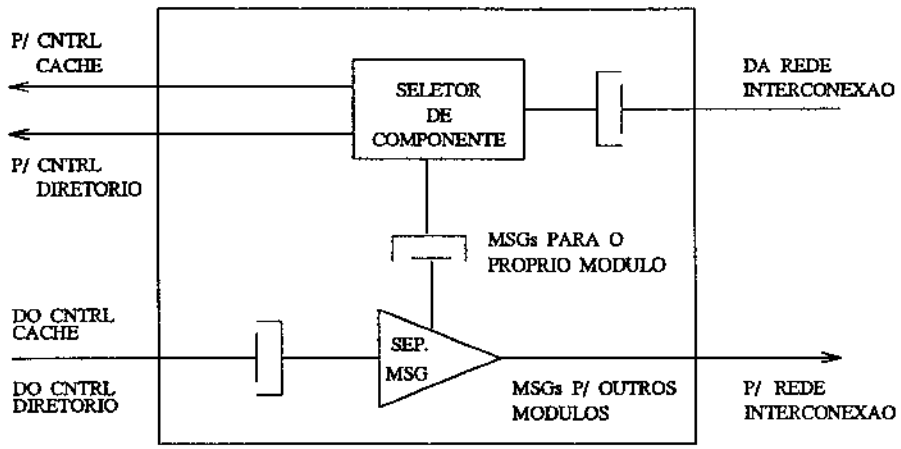

Figura 5.3: Diagrama do servidor de mensagens

5.2.4 Servidor de Mensagens

Existem duas funções básicas associadas ao servidor de mensagens:

- serve como *interface* com a rede de interconexão recebendo, classificando e encaminhando as mensagens que são enviadas para o controlador do diretório e para o controlador do cache;
- serve como *interface* com o módulo, recebendo mensagens dos controladores de cache e diretório do módulo, enviando pela rede as mensagens destinadas a outros módulos e desviando para o próprio módulo as mensagens trocadas entre os controladores de cache e diretório do próprio módulo, evitando-se como isso, atrasos e uso da rede desnecessários.

A figura 5.3 mostra a organização interna do servidor. Mensagens enviadas pelos controladores de cache e diretório do módulo são armazenadas em uma única fila de entrada. A partir daí, um elemento separador verifica as mensagens cujo destino é o próprio módulo, armazenando-as na fila interna e enviando para a rede as que se destinarem a outros módulos. As mensagens que chegam da rede também são armazenadas em uma fila de entrada de mensagens externas. Um separador de mensagens verifica a existência de mensagens pendentes nas filas interna e externa e faz a transmissão para controlador ao qual ela se destina, sempre alternando o atendimento entre as duas filas para evitar bloqueios.

O formato das mensagens trocadas entre o controlador de cache e o controlador de diretório dos diversos módulos é o mesmo, só variando o comprimento. Como mostra a figura 5.4, uma mensagem completa é dividida em pacotes. O tamanho de cada pacote é dependente do

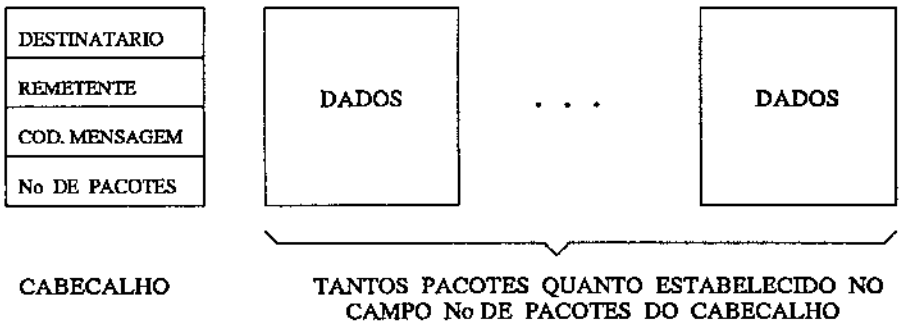

Figura 5.4: Formato das mensagens trocadas entre controladores de cache e de diretório.

comprimento do canal de comunicação da rede; no simulador, a rede está configurada com um canal de 32 *bits*.

O primeiro pacote de cada mensagem corresponde ao cabeçalho e está subdividido em 4 campos de 8 *bits*, sendo eles: identificador do módulo de destino, identificador do módulo que enviou a mensagem,[4] código da mensagem e número de pacotes de dados que seguem o cabeçalho e completam a mensagem.

5.2.5 Rede de Interconexão

A rede de interconexão disponível atualmente no simulador é do tipo Omega com chaves multi-estágios conforme mostra a figura 5.5. Podem ser destacadas como características básicas dessa rede a utilização de chaves 2×2, canal de transmissão de 32 *bits*, *buffer* nas entradas das chaves (garantindo que mensagens destinadas a um canal de saída ocupado não sejam descartadas, mas sim aguardem bloqueadas a liberação da saída desejada), funcionamento baseado na troca de pacotes, controle distribuído (a decodificação do identificador do destino é efetuada em cada chave por onde a mensagem passa), garantia de entrega da mensagem completa e na ordem em que fora enviada e, finalmente, garantia de que uma mensagem bloqueada devido a uma ocupação do canal de saída seja a próxima mensagem a utilizar o canal.

O entendimento do funcionamento de toda a rede pode ser resumido ao entendimento do funcionamento de uma única chave da rede. O detalhe na figura 5.5 mostra a estrutura interna das chaves. As mensagens que chegam são armazenadas nas filas A e B. A cada ciclo da

[4] O tamanho desses campos identificadores (8 *bits*) é o fator que limita o simulador atualmente a um número máximo de 256 processadores. Entretanto, através da utilização de técnicas de combinação desses dois campo, é possível elevar o número máximo de módulos a até 65536.

chave os canais de saída 0 e 1 podem estar ocupados transmitindo uma mensagem (um pacote por ciclo) ou livres. Cada vez que um canal de saída se torna livre, a lógica de controle da chave verifica se existe alguma mensagem bloqueada nas filas de entrada aguardando tal saída e inicia a sua transmissão em caso afirmativo. Os dois canais de saída funcionam em paralelo. A lógica de controle da chave utiliza os campos identificador de destino e número de pacotes no cabeçalho da mensagem para descobrir qual canal de saída será utilizado e quando uma mensagem chegou ao fim respectivamente.

Cada chave da rede é implementada no simulador como uma tarefa como descrito no início do capítulo. Com isso, o funcionamento das chaves se dá de forma paralela entre si e em relação aos demais componentes do simulador.

5.2.6 Barramento de Invalidações

O barramento de invalidações é usado exclusivamente na configuração que simula o protocolo DLE. Assim como os outros componentes, o barramento de invalidações é implementado como uma tarefa que executa paralelamente às outras, com formas e ciclos de operação bem definidos. Uma única fila de entrada está disponível para ser usada pelos controladores de diretório que necessitarem enviar invalidações pelo componente. Cada mensagem de invalidação difundida pelo barramento é constituída por dois pacotes: o primeiro deles contém o endereço (módulo de memória + bloco) do bloco a ser invalidado; o segundo, o endereço do controlador de cache para onde deve ser enviada a confirmação da invalidação. Junto a cada controlador de cache existe uma fila onde o barramento de invalidações coloca uma réplica desses pacotes que serão consumidos no processo de atendimento da invalidação.

Ao todo são gastos três ciclos de barramento para o envio de uma mensagem de invalidação. O primeiro deles representa o tempo gasto pelo controlador de diretório para obter o controle do barramento; os dois ciclos restantes são usados no envio dos pacotes.

A simulação do processo de arbitragem para indicar o próximo controlador de diretório a ter o controle do barramento é obtida através da fila de entrada das invalidações que, como dito, é única, fornecendo um mecanismo justo, servindo as requisições na ordem em que elas chegam.

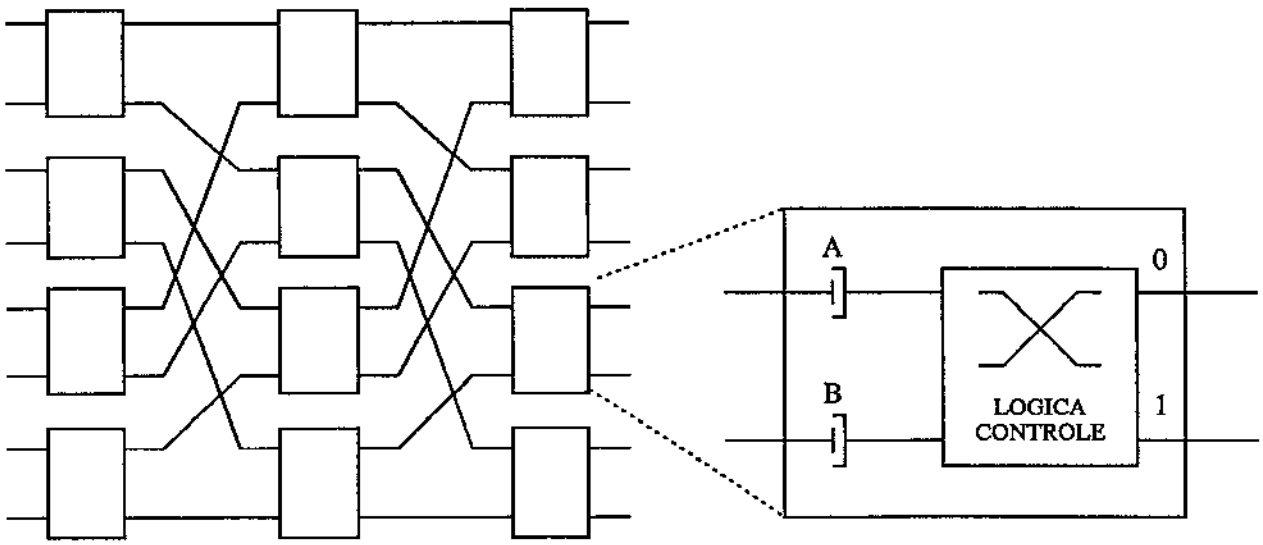

Figura 5.5: Diagrama interno do modelo de implementação das chaves da rede de interconexão.

5.3 Gerador de Referências

Tradicionalmente, a avaliação de desempenho de sistemas de memória antes da sua construção tem sido feita a partir da técnica conhecida como simulação dirigida por seqüência de referências (*trace driven simulation*). Uma seqüência de referências — ou, simplesmente, *trace* — como o nome indica, corresponde ao registro de todas as referências feitas pelo processador ao sistema de memória, na ordem em que elas foram feitas, quando executando uma determinada aplicação. Após coletados, os *traces* podem ser usados para simular a execução da aplicação em um simulador do sistema sob testes visando a obtenção de dados de desempenho e de como eles são afetados à medida que são variados os parâmetros do simulador.

Em sistemas monoprocessados, a obtenção e utilização de *traces* tem sido largamente aplicada e aceita como eficaz há um muito tempo [Smit82]. Em sistemas multiprocessados de memória compartilhada a mesma técnica tem sido adotada, sendo que a sua validade tem sido posta em dúvida em ambientes onde se deseja um alto grau de fidelidade das avaliações. Dentre os principais fatores para tal desconfiança podem ser citados: a existência de diversos processos executando em paralelo, influenciando uns aos outros, torna os *traces* altamente dependentes da arquitetura onde eles são gerados; o tamanho e velocidade de geração do *trace* cresce proporcionalmente com o número de processadores, dificultando o processo de obtenção e forçando a adoção de técnicas como, por exemplo, a fragmentação dos *traces*; o próprio processo de obtenção tende a induzir distorções na ordem em que as referências são feitas que

não existiriam caso a execução se desse sem a geração de *traces*.[SJF91]

Alternativamente à utilização de *traces* reais, alguns pesquisadores têm se baseado no comportamento estatístico apresentado por aplicações seqüenciais no acesso à memória associado à escolha de alguns parâmetros que caracterizam as aplicações paralelas — como a taxa de compartilhamento de dados — para gerar *traces* sintéticos. Essa estratégia encontra maior aceitação principalmente quando o tipo de avaliação que se deseja fazer é comparativa. Archibald e Baer [ArBa86b], por exemplo, desenvolvem um modelo probabilístico onde parâmetros como taxa de operações de leitura, número de blocos compartilhados, taxa de referências a blocos compartilhados, além de outros, são variados no processo de geração de *traces* sintéticos usados na avaliação de protocolos de coerência de cache em sistemas que utilizam o barramento como rede de interconexão.

O modelo para geração de *traces* adotado no nosso simulador é semelhante ao usado por Archibald e Baer. Entretanto, algumas modificações visando tornar os *traces* mais próximos da realidade foram introduzidas. A principal delas é a utilização de informações mais apuradas, recentemente divulgadas em trabalhos como [GuWe92] e não disponíveis na época dos estudos de Archibald e Baer, relacionadas à forma com que os dados (blocos) são compartilhados em aplicações paralelas reais. A seguir estão relacionadas algumas das informações que devem ser passadas ao simulador para que ele tenha um padrão a ser seguido na geração dos *traces* [5].

- Taxa de operações de leitura (a blocos privados e compartilhados);
- Taxa de operações de escrita (a blocos privados e compartilhados);
- Distribuição das operações compartilhadas em função do número de processadores que compartilham os blocos (ex. taxa de operações sobre blocos compartilhados por 1, 2, 3, ou mais processadores);
- Taxa de operações sobre blocos já referenciados anteriormente e sobre blocos nunca referenciados pelos processadores;
- Distribuição das referências entre os módulos de memória (atualmente estão disponíveis a distribuição uniforme e a exponencial — no caso da distribuição uniforme pode-se ainda impor uma taxa de referências direcionadas à memória local do módulo processador.)

[5] Os valores usados na simulação dos protocolos de coerência em estudo serão apresentados juntamente com os resultados obtidos no próximo capítulo

A geração dos *traces* é feita pelo componente processador e em tempo de simulação. Essa decisão apresenta duas vantagens em relação à utilização de *traces* gerados em uma fase anterior à simulação: dispensa o armazenamento dos *traces* (que requer um enorme espaço) e viabiliza a utilização do estado dos blocos na memória (possibilitando a obtenção, por exemplo, de informações do tipo o bloco está sendo compartilhado ou não, quantos processadores detêm cópia do bloco, etc.) no momento da produção de cada *trace*. Em contrapartida, a simulação se torna certamente mais lenta.

A geração dos *traces* é função exclusiva do componente processador embora, como mencionado, ele obtenha informações do estado dos blocos na memória (a partir das estruturas internas dos controladores de diretório) para auxiliar o processo de geração. Os *traces* são gerados a cada ciclo do processador, contanto que a operação (leitura ou escrita) induzida pelo *trace* anterior tenha sido atendida pelo controlador de cache correspondente.

A fim de tornar mais próximo o comportamento dos *traces* da realidade, um número limitado das referências mais recentemente efetuadas é mantido e, periodicamente (a uma taxa predefinida), essas referências são repetidas para simular a existência de *loops* no trace.

5.4 Protocolo de Comunicação

O protocolo de comunicação define o conjunto de mensagens e a ordem em que elas devem ser trocadas entre os componentes controladores de cache e diretórios a fim de atender a pedidos de cópias de blocos de memória e de manter a coerência dessas cópias nos caches do sistema. As Tabelas 5.1 e 5.2 reunem as características e descrevem a função de cada uma das mensagens do protocolo de comunicação definido para o simulador. A Tabela 5.1 relaciona as mensagens recebidas pelos controladores de diretórios e a Tabela 5.2 as mensagens recebidas pelos controladores de cache. A primeira coluna dessas tabelas traz o nome do código da mensagem, a segunda, o seu tamanho em pacotes — desprezando o cabeçalho — juntamente com a indicação do conteúdo desses pacotes; finalmente, a terceira coluna traz uma breve descrição da função da mensagem.

O protocolo de comunicação é comum aos três tipos de diretórios estudados. Com isso, evita-se que fatores externos, no caso diferenças no protocolo de comunicação, influenciem o desempenho dos diretórios.

A seguir estão relacionadas algumas das principais características impostas aos protocolos

de coerência simulados que são suportadas pelo protocolo de comunicação.

- Todos os protocolos de coerência simulados são baseados na invalidação das cópias desatualizadas como forma de manutenção da coerência dos caches;
- A cada mensagem de invalidação corresponde uma mensagem de confirmação da invalidação;
- O controlador de diretório fica livre para tratar as mensagens que chegam até ele, mesmo quando as "negociações" para a manutenção da coerência de um bloco estão em andamento (conforme explicado anteriormente no detalhamento deste componente);
- Solicitações de escrita a blocos compartilhados desencadeiam o envio de mensagens de invalidação emitidas pelo controlador de diretório correspondente ao bloco, sendo a confirmação da invalidação enviada diretamente ao controlador de cache que requisitou a escrita. Dessa forma, a carga de mensagens que devem ser tratadas pelos controladores de diretórios é aliviada;
- Nenhuma operação de escrita é feita antes da garantia de que a cópia do bloco que está sendo alterado é a única cópia válida em cache do bloco. Com isso, os protocolos estudados obedecem ao modelo de consistência de memória *seqüencial* como definido por Lamport [Lamp79] (apresentado no capítulo 1);
- Qualquer operação (leitura ou escrita) sobre um bloco atendida pelo controlador do diretório tem a garantia de ser efetuada ao menos uma vez, antes que uma outra operação emitida por um outro controlador de cache envolvendo o mesmo bloco venha a impossibilitá-la. Assim, a operação de escrita sobre um bloco compartilhado, por exemplo, se atendida, tem a garantia de ser executada antes que uma outra operação de escrita oriunda de um controlador de cache distinto do primeiro a interrompa (essa segunda escrita pode ocorrer, por exemplo, enquanto o primeiro controlador estiver aguardando as confirmações de invalidação das outras cópias do bloco no sistema);
- Operações de substituição de cópias modificadas no cache que ocasionam a atualização do bloco na memória (política *copy-back*) são executadas paralelamente à operação de solicitação do bloco que irá tomar o lugar do bloco substituído. A cópia em cache só é efetivamente descartada, contudo, quando a confirmação da atualização do bloco na memória é recebida.

- Pedidos de leitura ou de escrita a blocos modificados são remetidos ao controlador de cache que detém a cópia mais atual; esse controlador se incumbe de enviar a cópia para o cache requisitante, atualizando também a memória, no caso de leitura, ou informando a transferência da posse ao controlador do diretório, no caso de escrita.

Tabela 5.1: Conjunto de mensagens do protocolo de comunicação direcionadas aos controladores de diretório.

MENSAGEM	TAMANHO - CONTEÚDO	DESCRIÇÃO
PRIV_BLK_RD_REQ	1 - endereço do bloco	Requisição de um bloco garantidamente privado[a] para o processador requisitante. É usado tanto para leitura quanto para escrita.
BLK_RD_REQ	1 - endereço do bloco	Requisição de um bloco compartilhado.[b]
BLK_RD_EXCL_REQ	1 - endereço do bloco	Requisição para escrita de um bloco compartilhado (requisição de cópia exclusiva).
DRT_BLK_MEM_UPD	1 - endereço do controlador de cache que recebeu uma cópia do bloco + endereço do bloco N - cópia do bloco[c]	Atualização da cópia da memória que se encontra desatualizada; ocorre em função de um pedido de leitura de um bloco que está modificado em um dos caches do sistema.
DRT_BLK_RD_EXCL_ACK	1 - endereço do controlador de cache que recebeu uma cópia do bloco + endereço do bloco	Confirmação de que a cópia mais atual do bloco fora transferida para o controlador de cache que a requisitou para escrita; enviada pelo último controlador a modificar o bloco.
DIR_INV_REQ	1 - endereço do bloco	Requisição de invalidação de outras cópias de um bloco; a cópia local no controlador de cache requisitante está válida.
PRIV_BLK_RPL_REQ	1 - endereço do bloco N - cópia do bloco	Requisição de substituição de um bloco privado.

[a] Para garantir a existência de blocos privados a cada processador na memória local de cada módulo, o gerador de *traces* divide o espaço de endereçamento das memórias em áreas privada e compartilhada. A área privada, por sua vez, é subdividida em áreas privadas a cada módulo processador do sistema.

[b] Apesar de passível de compartilhamento, o bloco requisitado pode, no instante da requisição, não está sendo compartilhado por nenhum processador.

[c] O número exato de pacotes depende do tamanho do bloco e do tamanho do pacote; para um bloco de 16 *bytes* em um sistema com canal de transmissão de 32 bits (4 *bytes*/pacote) tem-se $N = 4$.

Tabela 5.1: Conjunto de mensagens do protocolo de comunicação direcionadas aos controladores de diretório(continuação).

MENSAGEM	TAMANHO - CONTEÚDO	DESCRIÇÃO
RPL_REQ	1 - endereço do bloco N - cópia do bloco	Requisição de substituição de um bloco compartilhado.
CLR_ENTRY	1 - endereço do bloco	Requisição de atualização do diretório em função da substituição da cópia não alterada do bloco no cache.
RLS_DIR_ENTRY	1 - endereço do bloco	Requisição de liberação do *bit* de "tranca" da entrada correspondente ao bloco; indica que um processo de manutenção da coerência de várias fases chegou ao fim.

Tabela 5.2: Conjunto de mensagens do protocolo de comunicação recebido pelos controladores de cache.

MENSAGEM	TAMANHO - CONTEÚDO	DESCRIÇÃO
PRIV_BLK_RD_ASW	N - cópia do bloco	Resposta a um pedido de leitura de um bloco privado.
CLR_BLK_RD_ASW	N - cópia do bloco	Resposta a um pedido de leitura de um bloco compartilhado; a cópia enviada estava atualizada na memória.
DRT_BLK_RD_ASW	N - cópia do bloco	Resposta a um pedido de leitura de um bloco compartilhado; a cópia fora enviada pelo controlador de cache que a modificou por último. A cópia da memória estava desatualizada.
DIR_UPDATED	0	Quando é feito um pedido de leitura a um bloco desatualizado na memória, o pedido é retransmitido para o controlador de cache que detém a cópia mais atual que a envia para o cache requisitante e para a memória a fim de atualizá-la. DIR_UPDATED indica ao controlador de cache requisitante que o diretório já se encontra atualizado e que ele pode enviar mensagens como, por exemplo, DIR_INV_REQ.
CLR_BLK_RD_EXCL_ASW	1 - número de sinais INV_ACK que devem ser aguardados N - cópia do bloco	Resposta a um pedido para escrita de um bloco que encontrava-se atualizado na memória; juntamente com a cópia do bloco é enviado o número de mensagens de confirmação da invalidação (enviadas pelos outros controladores de cache que compartilham o bloco) que devem ser aguardadas antes que a modificação possa ser feita.
DRT_BLK_RD_EXCL_ASW	N - cópia do bloco	Resposta a um pedido para escrita de um bloco que encontrava-se desatualizado na memória; a cópia fora enviada pelo controlador de cache que o modificou por último.

Tabela 5.2: Conjunto de mensagens do protocolo de comunicação recebido pelos controladores de cache (continuação).

MENSAGEM	TAMANHO - CONTEÚDO	DESCRIÇÃO
DRT_BLK_OWN_GRT	0	Indicação de que o diretório já transferiu a posse do bloco para o controlador de cache que recebeu ou irá receber um DRT_BLK_RD_EXCL_ASW; o controlador de cache só está autorizado a executar a alteração na cópia recebida após receber esse sinal.
INV_ASW	1 - número de confirmações de invalidação (INV_ACK) que devem ser aguardadas	Indica que o controlador de diretório aceitou um pedido de invalidação (DIR_INV_REQ) de um bloco que está para ser modificado no cache local; deve-se aguardar todas as confirmações antes de prosseguir com a modificação.
INV_ACK	0	Confirmação de invalidação enviadas pelos controladores de cache que compartilhavam o bloco.
INV_REQ	1 - endereço do bloco + endereço do controlador de cache para onde enviar a confirmação (INV_ACK)	Requisição de invalidação, enviada pelo controlador de diretório, de uma cópia de bloco compartilhado existente no cache local. Após a invalidação deve ser enviada uma confirmação (INV_ACK) para o controlador de cache cujo endereço fora enviado junto com a mensagem.
INV_NAK	0	O pedido de invalidação temporariamente não pode ser atendido pelo controlador de diretório e deve ser ressubmetido.
DRT_BLK_RD_REQ	1 - endereço do bloco + endereço do controlador de cache para onde enviar a cópia do bloco	Requisição de envio da cópia mais atual, mantida no cache local, para um outro controlador de cache; adicionalmente, deve ser enviada uma cópia para atualizar a memória; a cópia local permanece válida.
DRT_BLK_RD_EXCL_REQ	1 - endereço do bloco + endereço do controlador de cache para onde enviar a cópia do bloco	Requisição de transferência da cópia mais atual do bloco mantida no cache local; a cópia local deve ser invalidada.

Tabela 5.2: Conjunto de mensagens do protocolo de comunicação recebido pelos controladores de cache (continuação).

MENSAGEM	TAMANHO - CONTEÚDO	DESCRIÇÃO
BLK_RD_NAK	0	O pedido de cópia do bloco para leitura temporariamente não pode ser atendido e deve ser ressubmetido.
BLK_RD_EXCL_NAK	0	O pedido de cópia do bloco para escrita temporariamente não pode ser atendido e deve ser ressubmetido.
RPL_ACK	0	O pedido de atualização da cópia do bloco na memória foi atendido e a cópia no cache local pode ser descartada.
RPL_NAK	0	O pedido de atualização da cópia na memória temporariamente não pode ser atendido e deve ser ressubmetido.
BLK_RD_EXCL_NAK	0	O pedido de cópia para escrita não pode ser temporariamente atendido e deve ser ressubmetido.
FAULT_PTR_INV[a]	1 - endereço do bloco	Solicitação de invalidação de uma cópia do bloco no cache local devido a uma substituição no apontador correspondente ao controlador de cache na entrada do bloco em um diretório limitado.

[a] Mensagem utilizada apenas nos diretório limitados.

5.5 Observações Finais

Uma vez que o objetivo principal do simulador é o estudo comparativo de protocolos de coerência distintos, o enfoque adotado no seu projeto e implementação foi o de torná-lo independente de qualquer dos protocolos avaliados. Assim, sua característica mais forte é a capacidade de ser modificado ou estendido, visando a adequação à alguma avaliação específica, sem a necessidade de um grande esforço de reprogramação. Componentes inteiros podem ser substituídos sem que alterações tenham necessariamente que ser feitas nos componentes restantes — obviamente para que isso seja válido, o componente introduzido deve respeitar a *interface* definida para a parte do simulador com a qual ele irá interagir (essa limitação restringe-se, em grande parte, ao formato das mensagens); os controladores dos diretórios avaliados, por exemplo, foram implementados e intercambiados sem que componentes como processador, controlador de cache, servidor de mensagens, etc, tivessem conhecimento desse intercâmbio.[6]

A utilização dos conceitos da orientação a objetos e o conseqüente aumento da capacidade de abstração experimentado na concepção e desenvolvimento de cada componente foi, em grande parte, responsável por essa capacidade de adaptação do simulador a novos ambientes. C++, por sua vez, forneceu a flexibilidade que, achávamos, seria necessária desde o início do projeto — a opção de trabalhar com a área de alocação dinâmica e com objetos criados em tempo de execução mostrou ser de importância fundamental para a construção de um ambiente de simulação razoavelmente eficiente e capaz de ser configurado de diversas formas, sem torná-lo extremamente complexo.

Por outro lado, a biblioteca de corrotinas, mencionada no início do capítulo, além de gerenciar a execução em paralelo das tarefas, forneceu estruturas de dados básicas e tornou-se a plataforma comum sobre a qual todos os componentes do simulador foram montados.

Como principais ausências sentidas no ambiente de desenvolvimento, entretanto, podem ser citadas a inexistência de um depurador, embutido talvez na própria biblioteca de corrotinas, que fosse voltado para a forma paralela com que as tarefas executam — o que nos obrigou a desenvolver rotinas básicas para a verificação de cada componente do simulador — e, também, a inexistência de facilidades para a geração das informações estatísticas sobre as quais são efetuadas as avaliações dos resultados das simulações.

No próximo capítulo serão apresentados os parâmetros e as configurações que foram uti-

[6] No caso específico do protocolo DLE, o componente controlador de cache teve que sofrer alguma modificação devido à introdução do componente barramento de invalidações.

lizados na simulação dos protocolos de coerência em estudo, juntamente com os resultados obtidos.

Tabela 5.3: Relação de programas que compõem o ambiente de simulação.

PROGRAMA	DESCRIÇÃO
PROCESSOR.CC	Efetua, a cada ciclo de processador, uma referência a um endereço de memória que pode ser uma leitura ou escrita a um dado compartilhado ou privado, de acordo com uma distribuição predefinida informada como parâmetro de entrada. A geração das referências a dados compartilhados é ainda influenciada pelo padrão de compartilhamento (dado compartilhado por 0, 1, 2, etc. processadores) também informado como entrada. As referências geradas são enviadas ao componente controlador de cache.
CACHE.CC	Simula a operação de um cache de mapeamento direto com política de atualização de memória *copy-back*. A gerência da informação armazenada fica a cargo do componente controlador de cache.
CACHECNTRL.CC	Recebe as referências geradas pelo componente processador e verifica se pode ser atendida pelo cache local. Caso não possa, envia uma mensagem ao controlador de diretório correspondente ao endereço solicitado, estabelecendo com esse componente um processo de manutenção de coerência quando necessário. Paralelamente, atende a pedidos de outros componentes, diferentes do processador, recebidos pela rede.
DIRFM.CC DIRLM.CC DIRDLE.CC	Simulam os componentes controladores de diretório para os protocolos *full map*, limitado e DLE respectivamente. As estruturas de diretórios são construídas apenas para os dados compartilhados. Atende ou recusa os pedidos enviados pelos controladores de cache a cada ciclo, sem permanecer em estados de pendência.
MSGSVR.CC	Recebe as mensagens dos componentes controladores de cache e de diretórios locais a um módulo, enviando pela rede as que não se destinarem ao próprio módulo. Recebe e distribui entre esses componentes as mensagens que chegam pela rede enviadas por componentes externos ao módulo.
MIN.CC	Simula uma rede multiestágios tipo Omega com chaves 2×2 com *buffer* em cada chave, controle distribuído, operação *pipeline* e troca de pacotes. Garante a entrega dos pacotes que compõem a mensagem na ordem de envio, sem quebrá-la. Cada ciclo da chave transfere um pacote de suas entradas para a saída indicada pela lógica de controle.
PROBE.CC INSPECT.CC	Ferramentas para depuração.

Capítulo 6

Configuração e Resultados das Simulações

Conforme estabelecido no capítulo anterior, as referências que alimentam o simulador são sintéticas, geradas a partir do comportamento de aplicações paralelas reais divulgado recentemente por Gupta e Weber [GuWe92]. Das cinco aplicações apresentadas naquele trabalho, três foram escolhidas, considerando os seus comportamentos distintos entre si com relação, principalmente, ao grau de compartilhamento dos dados. São elas *Maxflow*, *PTHOR* e *Locus-Route*. O padrão de comportamento dessas aplicações é apresentado na próxima seção junto com a configuração básica definida para o simulador. Esses dados formam a "janela experimental" sobre a qual as simulações dos protocolos com diretórios totalmente mapeado, limitado e DLE foram executadas. Na seção seguinte, os resultados dessas execuções são apresentados e discutidos.

6.1 Aplicações Simuladas

Antes de apresentar a configuração e os parâmetros de entrada adotados, essas aplicações serão brevemente descritas de acordo com a referência citada.

Maxflow

Maxflow encontra o fluxo máximo em um grafo orientado. Esse algoritmo tem larga aplicação em pesquisa operacional e em outras áreas. A maior parte do tempo de execução de sua versão paralela é gasta escolhendo nodos ativos do grafo, ajustando o fluxo ao longo

das arestas que chegam e que saem do grafo e, então, ativando os nodos sucessores. *Maxflow* explora o paralelismo a um grau fino. A escolha dos processadores que irão operar cada nodo é feita dinamicamente.

PTHOR

PTHOR é um simulador de portas e circuitos lógicos desenvolvido na Universidade de Stanford. As estruturas de dados básicas usadas no simulador representam elementos lógicos (*gates*, *flip-flops*, etc.), as redes (fios que interligam os elementos) e as filas de tarefas que contém os elementos ativos. Cada elemento tem uma fila de tarefas preferencial com o objetivo de aumentar a localidade de dados.

LocusRoute

LocusRoute é um roteador global para células VLSI que tem sido usado em projetos reais de circuitos integrados fornecendo roteamentos de alta qualidade. Ele explora o paralelismo roteando várias conexões concorrentemente. A estrutura central no *LocusRoute* é uma matriz de custos que é usada para registrar a presença de conexões já realizadas em cada ponto. O congestionamento de conexões, por sua vez, é usado como uma função de custo que guia o roteamento de novas conexões.

6.2 Configuração e Parâmetros adotados

A Tabela 6.1 apresenta a configuração básica definida para os componentes do simulador nas simulações efetuadas. Para os três tipos de protocolos avaliados o sistema foi configurado com 8 módulos processadores (cada módulo como especificado no capítulo anterior). Cada ciclo de operação de um componente é um múltiplo da unidade do tempo simulado.

Os controladores de cache e diretório operam com dois tipos de ciclos. Se o atendimento a uma mensagem evolver transferência de blocos (como, por exemplo, leitura de bloco na memória, no caso do controlador do diretório, e um pedido de transferência de um bloco modificado localmente, no caso do controlador de cache) uma quantidade maior de ciclos é gasta. Do contrário, a quantidade mínima de ciclos é usada. Para o controlador de diretório essa decisão é adequada e compatível com a realidade. Para o controlador de cache, entretanto, essa decisão afeta diretamente o componente processador, já que, este pode ter seu tempo de espera pelo atendimento a uma referência aumentado devido à ocupação do controlador de cache com o tratamento a uma mensagem externa envolvendo tranferência de blocos.

Tabela 6.1: Configuração básica do simulador.

COMPONENTE	PARÂMETRO	VALOR
Processador	Prob. de referenciar um bloco já referenciado antes	0.8
	Prob. de referenciar um bloco nunca referenciado antes	0.2
	Ciclos de operacão	2
Memória	Espaço privado alocado para cada módulo	1000 blocos
	Espaço compartilhado por todos os módulos em cada módulo	4000 blocos
	Tam. do endereço	32 *bits*
Cache	Tam. por módulo	256 *KBytes*
	Tipo de Associatividade	Mapeamento Direto
	Tamanho do bloco	16 *bytes*
	Política de atualização	*copy-back*
Controlador Cache	Ciclos de operação sem transferência de blocos	1
	Ciclos de operação com transferência de blocos	4
	Política de manutenção da coerência	Invalidação
Controlador Diretório	Ciclos de operação sem acesso à memória	2
	Ciclos de operação com acesso à memória	6
	Política de substituição de apontador (diretório limitado)	Aleatória
	Número de apontadores (diretório limitado/DLE)	3
Rede de Interconexão	Tam. canal	32 *bits*
	Tam. pacote	32 *bits*
	Ciclos de operação por chave	1
Servidor de Mensagens	Ciclos de operação	1
Bar.de Invalidação	Ciclos de arbitragem	1
	Ciclos para envio invalidação	2

Tabela 6.2: Padrão de comportamento das aplicações simuladas.

Aplicação	Ref. Compart.(%)		Ref. Privadas(%)		Níveis de Compartilhamento (%) – 8 proc.								Inv.
	Leitura	Escrita	Leitura	Escrita	+0	+1	+2	+3	+4	+5	+6	+7	Média
MaxFlow	49	8	24	19	0,5	90	5	2,4	0,8	0,4	0,3	0,6	1,17
PTHOR	43	5	38	14	38	55	5	0,7	0,4	0,3	0,2	0,4	0,77
LocusRoute	47	5	36	12	14	65	13	5	2	0,6	0,3	0,1	1,19

A Tabela 6.2 traz os padrões de comportamento das aplicações escolhidas.

As quatro primeiras colunas da Tabela 6.2 são auto-explicativas. As oito colunas que seguem indicam o grau de compartilhamento dos blocos registrados no momento em que invalidações eram efetuadas. Da primeira para a última, seus valores podem ser lidos como: taxa de blocos compartilhados por mais 0 processadores (ou seja, blocos existentes no cache de apenas 1 processador), taxa de blocos compartilhados por mais 1 processador e assim por diante. A última coluna indica o número médio de invalidações necessárias sempre que um bloco era modificado. Esses valores foram usados no simulador para guiar a geração das referências tanto para leitura quanto para escrita.

6.3 Resultados Obtidos

Cada aplicação foi simulada para cada um dos protocolos estudados perfazendo um total de 9 simulações. Em cada execucão foi solicitado que os processadores executassem cerca de 15 milhões de ciclos cada antes da geração dos dados estatísticos.

Antes de apresentar e discutir esses resultados, é importante verificar como o simulador respondeu ao padrão das aplicações na fase de geração das referências. As Tabelas 6.3 e 6.4 trazem essa informação. As discrepâncias mais relevantes entre as respostas do simulador e os parâmetros das aplicações são sentidas nos níveis de compartilhamento mais altos da Tabela 6.4 (particularmente as colunas +5, +6 e +7). Essas diferenças provavelmente decorrem do fato de o simulador não ser sempre capaz, no instante da geração da referência, de identificar na memória um bloco que satisfaça esses graus de compartilhamentos. É provável que em simulações mais longas essas diferenças tendam a diminuir. Note que o susposto bom desempenho dos protocolos limitados na identificação de blocos altamente compartilhados é falso; o que ocorre, na realidade, é que, como não existem blocos compartilhados por mais de três processadores (número máximo de apontadores definidos para a simulação desses protocolos), quando o gerador de referências

Tabela 6.3: Resposta do gerador de referencias do simulador aos padrões de comportamento das aplicações estudadas – operações por referência (8 processadores).

Aplicação	Protocolo Coerência	Ref. Compart.(%)		Ref. Privadas(%)	
		Leitura	Escrita	Leitura	Escrita
MaxFlow	Tot. Map.	48,42	6,09	25,38	20,09
	Limitado	48,53	7,1	24,72	19,57
	DLE	48,01	7,07	25,06	19,84
PTHOR	Tot. Map.	42,96	4,97	38,03	14,01
	Limitado	42,97	4,99	38,01	14,01
	DLE	42,96	4,97	38,03	14,01
LocusRoute	Tot. Map.	46,98	4,99	36,01	12,01
	Limitado	46,98	4,99	36,00	12,00
	DLE	46,98	4,99	36,01	12,01

busca um bloco compartilhado por mais de três processadores ele considera qualquer bloco compartilhado por três processadores como adequado.

6.3.1 Desempenho dos Processadores

Para obter o desempenho dos processadores em relação a cada um dos protocolos avaliados, foram registrados, no decorrer das simulações, o número de ciclos que os processadores permaneciam parados, aguardando a resposta dos controladores de cache, impossibilitados, portanto, de efetuar tarefa útil. O número máximo de ciclos de processador simulados diminuído dessa medida resulta no desempenho alcançado por cada processador. Como para todas as simulações, a única variação imposta ao ambiente de simulação foi a substituição do protocolo de coerência usado, o desempenho dos processadores resultante está diretamente associado a cada um desses protocolos. Os resultados médios de desempenho dos processadores para as três aplicações escolhidas está apresentado graficamente na Figura 6.1.

Nas três aplicações os desempenhos dos protocolos ficaram muito próximos. A razão principal para esse resultado é que em todas as aplicações mais de 90% das referências efetuadas foram destinadas a blocos com menos de 3 apontadores, favorecendo diretamente os protoco-

Tabela 6.4: Resposta do gerador de referencias do simulador aos padrões de comportamento das aplicações estudadas – níveis de compartilhamento durante escritas (8 processadores).

Aplicação	Protocolo Coerência	Níveis médios de compartilhamento (%)								Inv. Média
		+0	+1	+2	+3	+4	+5	+6	+7	
Maxflow	Tot. Map.	0,68	87,33	6,57	3,26	1,04	0,44	0,30	0,34	1,21
	Limitado	0,56	88,52	5,79	2,70	0,92	0,45	0,34	0,67	1,21
	DLE	0,57	89,09	5,7	2,77	0,9	0.37	0,25	0,32	1,18
PTHOR	Tot. Map.	38,38	55,19	4,95	0,71	0,40	0,24	0,07	0,03	0,70
	Limitado	38,08	54,93	5,01	0,70	0,40	0,29	0,19	0,38	0,74
	DLE	38,26	55,27	5,01	0,70	0,39	0,24	0,07	0,03	0,70
LocusRoute	Tot. Map.	14,04	65,13	13,07	4,97	1,98	0,56	0,20	0,02	1,18
	Limitado	14,01	64,98	13,08	4,96	1,98	0,59	0,28	0.09	1,19
	DLE	14,04	65,11	13,09	4,96	1,98	0,57	0,19	0,02	1,18

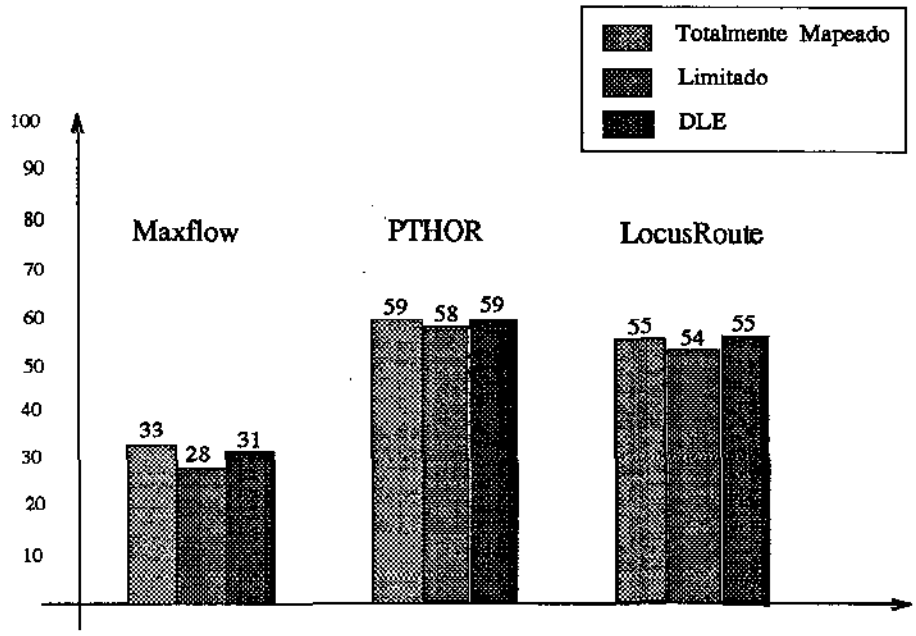

Figura 6.1: Desempenho médio dos processadores para cada um dos protocolos avaliados considerando aplicações distintas (% – ciclos úteis do processador / total de ciclos).

Tabela 6.5: Taxas médias de acerto, ausências e ausências por invalidação dos caches

Protocolo	Taxa Média	*Maxflow*	*PTHOR*	*LocusRoute*
Tot. Mapeado	acerto	88,78	95,03	94,28
	ausência	6,77	3,98	4,47
	ausência inv.	4,43	0,98	1,23
Limitado	acerto	85,30	94,88	94,10
	ausência	7,58	4,03	4,37
	ausência inv.	7,11	1,08	1,52
DLE	acerto	87,25	94,73	94,23
	ausência	6,88	4,13	4,37
	ausência inv.	5,85	1,13	1,38

los limitados. Além disso, como visto, o gerador de referências se mostrou pouco eficiente na geração de referências a blocos altamente compartilhados. Outro aspecto que deve ser considerado é que nos *traces* gerados, praticamente inexistiu a concorrência por blocos específicos, como comumente ocorre em aplicações paralelas reais com aqueles blocos que contêm variáveis tipo *lock*. Apesar dessas restrições, o protocolo DLE apresentou uma clara tendência de desempenho semelhante ao protocolo totalmente mapeado que apresentou o melhor desempenho em todas as aplicações; nas aplicações *PTHOR* e *LocusRoute* o desempenho foi igual, enquanto que na aplicação *Maxflow* esse desempenho ficou 2 pontos porcentuais abaixo e 3 pontos porcentuais acima do protocolo limitado.

6.3.2 Desempenho dos Caches

A Tabela 6.5 mostra as taxas de acerto, ausência e ausência por invalidação verificada nos caches durante as simulações. Novamente, o protocolo totalmente mapeado apresentou melhores taxas. Em duas das aplicações o protocolo DLE apresentou melhores resultados que o diretório limitado.

Tabela 6.6: Taxas médias de utilização da rede em número de pacotes enviados pelos 8 processadores por ciclo de processador.

Protocolo	*Maxflow*	*PTHOR*	*LocusRoute*
Tot. Mapeado	3,76	2,48	2,71
Limitado	4,01	2,50	2,75
DLE	3,89	2,56	2,72

6.3.3 Utilização da Rede

A fim de verificar o grau de utilização da rede de interconexão pelos protocolos foram registrados os números de pacotes que foram enviados pela rede por cada um dos servidores de mensagem dos 8 processadores configurados. A soma desses registros foi então dividida pelo número de ciclos de processador para compensar pequenas diferenças, já que, nem todas as simulações terminaram com um número igual de ciclos de processador decorridos. O resultado, que nos dá o número médio de pacotes que foram enviados pela rede pelos 8 processadores a cada ciclo de processador, está listado na Tabela 6.6. Mais uma vez, o protocolo totalmente mapeado se comportou melhor utilizando o minímo a rede. O protocolo DLE apresentou melhores resultados que o limitado em duas das aplicações. Note que a alta taxa de utilização da rede é comprovadamente o responsável pelo baixo desempenho dos processadores; isso pode ser percebido ao compararmos as informações da Tabela 6.6 com o gráfico de desempenho dos processadores na Figura 6.1.

6.3.4 Utilização do Barramento de Invalidações

Um dos pontos de maior interesse nos resultados das simulações do protocolo DLE está relacionado com o comportamento do barramento de invalidações. As principais informações registradas foram quantos ciclos do total do tempo simulado este componente ficou ocupado atendendo requisições e o número médio e máximo de mensagens aguardando atendimento. A Tabela 6.7 traz esses resultados.

O grau de utilização do barramento de invalidações verificado, apesar de variar bastante de uma aplicação para outra, foi bem reduzido. O numero máximo de mensagens aguardando envio foi de uma única mensagem para as três aplicações simuladas o que também garantiu

Tabela 6.7: Taxas médias de utilização do barramento de invalidações.

Aplicação	Ciclos ocupados (%)	Num. médio de msgs aguardando envio	Num. max. de msgs aguardando envio
Maxflow	1,8	0,0035	1
PTHOR	0,1	0,000086	1
LocusRoute	0,6	0,001	1

um número médio desprezível dessas mensagens ao longo do tempo simulado. Juntos esses resultados indicam que é viável que um grande numero de processadores podem ser conectados ao barramento antes que ele venha a ser motivo de contenção e passe a influir negativamente no desempenho de todo o sistema. Vale ressaltar que na simulação do barramento não se fez nenhum esforço de otimização, do tipo operação paralela do processo de arbitragam com o envio de mensagens (operação *pipeline*), o que certamente contribuiria para um desempenho ainda melhor desse componente.

6.4 Outros Resultados

A fim de avaliar o comportamento dos protocolos em aplicações que apresentam maior grau de compartilhamento, um outro experimento foi realizado. Dessa vez, o gerador de referências foi alimentado com dados de uma aplicação hipotética, na qual o grau de compartilhamento dos blocos fora obtido a partir de um gerador aleatório com distribuição exponencial e valor médio 6; para as operações de leitura e escrita compartilhadas, foram utilizados dados similares aos da aplicação *LocusRoute*, sendo que, buscando-se privilegiar tais operações, seus porcentuais foram aumentados em 5 pontos em relação aos valores originais daquela aplicação. A Tabela 6.8 concentra essas informações juntamente com a resposta fornecida pelo gerador para cada um dos protocolos. O resultado médio de desempenho dos três protocolos para a aplicação hipotética é apresentado na Figura 6.2.

A Figura 6.2 mostra a superioridade do protocolo DLE sobre o protocolo limitado quando o grau de compartilhamento dos blocos cresce. O principal fator responsável pelo desempenho inferior do protocolo limitado foi, como esperado, o aumento na taxa de ausências por invalidação no cache e conseqüente redução da taxa de acertos (a taxa de ausências manteve-se em

Tabela 6.8: Comportamento da aplicação hipotética e resposta do gerador de referência para cada protocolo simulado.

	Ref. Compart.(%)		Ref. Privadas(%)		Níveis de Compartilhamento (%) - 8 proc.								Inv.
Protocolo	Leitura	Escrita	Leitura	Escrita	+0	+1	+2	+3	+4	+5	+6	+7	Média
Entrada	52	10	31	7	8,1	13,8	11,8	18,9	15,3	12,8	11,1	8,2	3,43
Tot. Mapeado	51	8	33	8	10,5	18,1	15,5	23,9	18,2	10,9	2,6	0,2	2,65
Limitado	50	9	33	8	8,7	14,9	12,7	18,2	14,6	12,2	10,6	7,9	3,33
DLE	51	8	33	8	10,6	18,2	15,5	24,1	18,4	10,8	2,3	0,2	2,63

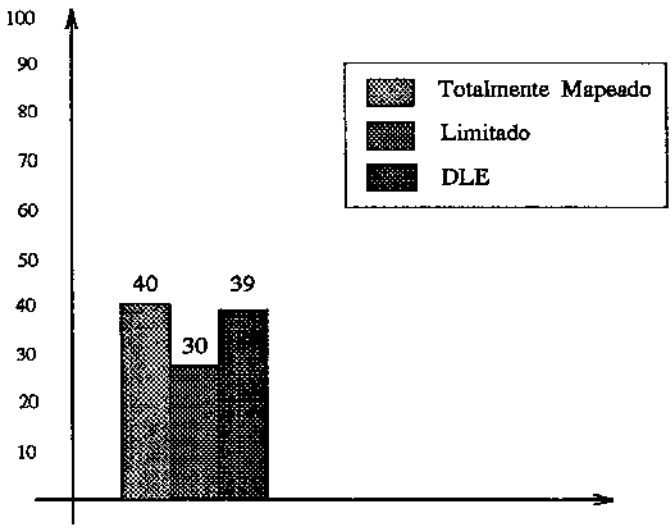

Figura 6.2: Desempenho médio dos processadores para cada um dos protocolos avaliados considerando uma aplicação hipotética (% - ciclos úteis do processador / total de ciclos).

Tabela 6.9: Taxas médias de acerto, ausências e ausências por invalidação dos caches na aplicação hipotética

Taxa Média	Tot. Mapeado	Limitado	DLE
acerto	90,9	86,6	90,0
ausência	4,1	4,2	4,3
ausência inv.	5,0	9,2	5,7

níveis próximos nos três protocolos); a Tabela 6.9 apresenta essas taxas.

A taxa de utilização do barramento de invalidações do protocolo DLE na aplicação hipotética foi de 4% com número máximo de mensagens aguardadndo o envio limitado a 2.

Capítulo 7

Conclusões e Trabalhos Futuros

7.1 Conclusões

Atualmente é possível afirmar que existe um número razoável de soluções para o problema da coerência de cache em grandes sistemas multiprocessados. Poucas delas, porém, oferecem uma relação compromisso espaço × complexidade tão boa quanto os diretórios limitados. A prática mostra que o comportamento da grande maioria das aplicações paralelas favorece fortemente as soluções limitadas, já que, o número médio de processadores que compartilham um bloco registrado em tempo de execução é normalmente baixo, indicando que um pequeno numero de apontadores é suficiente para comportar, a maior parte do tempo, o endereço de todos os caches com cópia do bloco. Por outro lado, um algoritmo simples, de gerenciamento do diretório responde por sua baixa complexidade.

Na busca por um aumento no desempenho dos protocolos baseados em diretòrios limitados, os esforços de pesquisa deveriam ser orientados para o desenvolvimento de mecanismos que tratem os casos de exceção, aqueles onde o número de processadores que compartilham um bloco excede o número de apontadores definidos na estrutura, mantendo a relação compromisso, mencionada acima, inalterada. A solução DLE, proposta neste trabalho, é uma tentativa neste sentido. O espaço ocupado pelo diretório na solução DLE não é alterado (considerando 3 como número mínimo de apontadores — um limite razoável para se obter um bom desempenho dos protocolos limitados, conforme os resultados apresentados no capítulo anterior). A complexidade, por outro lado, cresce um pouco. A existência de uma via paralela à rede de interconexão — o barramento de invalidações — por onde comandos de consistência são enviados faz com que o protocolo tenha que ser capaz de resolver situações como por exemplo: o que fazer quando um

processador recebe pelo barramento um comando de invalidação de um bloco que está sendo atualmente buscado na memória por ele mas cuja cópia ainda não fora enviada? Ou ainda, quando esse comando de invalidação está relacionado a um bloco cujo pedido de liberação da entrada no diretório (motivada por sua substituição no cache local) está em andamento. Esses são exemplos de situações que nos deparamos durante o processo de simulação, forçando-nos a tomar decisões que aumentaram a complexidade do protocolo. Note-se, entretanto, que situações semelhantes podem ocorrer com o diretório limitado simples em ambientes onde a rede de interconexão não garante a entrega das mensagens na ordem em que foram enviadas.

Especificamente com relação ao barramento de invalidações, acreditamos que, dada a forma especializada com que ele opera, ele pode ser projetado de modo a ser um dispositivo altamente eficiente, capaz de interligar um grande número de processadores antes que uma possível saturação possa ser detectada. Um ponto que deve ser levado em consideração em relação a essa saturação, entretanto, é que parte das invalidações enviadas pelo barramento serão referentes a blocos com menos cópias compartilhadas que o número de apontadores no diretório sempre que o *bit* de difusão esteja ligado (decorrente de uma falta de apontadores na entrada do diretório relativa ao bloco em seu passado).

Finalmente, concluímos que as avaliações efetuadas no capítulo 7, apesar de requererem um aprofundamento maior, nos mostra que a solução DLE é uma opção viável na construção de sistemas multiprocessados baseados em diretórios altamente eficientes.

7.2 Extensões e Trabalhos Futuros

As formas possíveis pelas quais o presente trabalho pode ser estendido estão descritas a seguir.

- Utilização do ambiente de simulação desenvolvido para obtenção de medidas de desempenho dos protocolos avaliados em configurações com um número maior de processadores;
- Repetição das simulações, dessa vez usando *traces* reais, objetivando com isso não só avaliar o desempenho dos protocolos estudados com esses *traces*, como também verificar em que medida, os *traces* sintéticos, na forma como são gerados atualmente no simulador, são capazes de substituir satisfatoriamente os *traces* reais.
- Desenvolvimento de um modelo analítico do barramento de invalidações a fim de comparar os resultados indicados pelo modelo com os resultados obtidos por simulação.

- Em uma extensão mais ambiciosa, projetar e implementar o barramento de invalidações com a funcionalidade descrita neste trabalho a fim de avaliar as dificuldades e demais problemas associados com a sua construção e torná-lo um protótipo passível de utilização na prática.

Bibliografia

[Andr89] Warren Andrew. 32–Bit Buses Contend for Designers' Attention. *Computer Design*, pp. 78–96, November 1, 1989.

[ArBa84] James Archibald e Jean-Loup Baer. An Economical Solution to the Cache Coherence Problem. *Proc. Int. Symp. on Computer Architecture*, pp. 355–362, 1984.

[ArBa86a] J. Archibald e Jean-Loup Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[ArBa86b] J. Archibald e Jean-Loup Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[ASHH88] Anant Agarwal, Richard Simoni, John Hennessy, e Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. *Proc. 15th. Int. Symp. on Computer Architecture*, 1988.

[Bala84] R. V. Balakrishnan. The Proposed IEEE 896 FutureBus — A Solution to the Bus Driving Problem. *IEEE Micro*, pp. 23–27, August 1984.

[Bril87] R. J. Bril. An Implementation Independent Approach to Cache Memories. *Computer Architecture News*, 15(3):17–24, June 1987.

[BYA89] Laxmi N. Bhuyan, Qing Yang, e Dharma P. Agrawal. Performance of Multiprocessor Interconnection Networks. *IEEE Computer*, 22(2):25–37, Feb. 1989.

[CeFe78] L. M. Censier e P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[CFKA90] David Chaiken, Craig Fields, Kiyoshi Kurihara, e Anant Agarwal. Directory–Based Cached Coherence in Large–Scale Multiprocessors. *IEEE Computer*, 23(6):49–58, 1990.

[CKA91] David Chaiken, John Kubatowicz, e Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. *Proc. of 4th ASPLOS*, pp. 224–234, 1991.

[EgKa89] S. J. Eggers e Randy H. Katz. Evaluating the Performance of Four Snooping Cache Coherency Protocols. *Proceedings of 16th Int. Symp. Computer Architecture*, pp. 2–15, 1989.

[Feng81] Tse-Yun Feng. A Survey of Interconnection Networks. *IEEE Computer*, 14(12):12–27, Dec. 1981.

[Fran84] S. J. Frank. Tightly Coupled Multiprocessor System Speeds Memory–Access Times. *Eletronics*, pp. 164–167, January 1984.

[Good83] J. R. Goodman. Using Cache Memory to Reduce Processor–Memory Traffic. *Proceedings of 10th Int. Symp. on Comp. Architecture*, pp. 124–131, June 1983.

[Gust84] David B. Gustavson. Computer Buses — A Tutorial. *IEEE Micro*, pp. 7–22, August 1984.

[Gust92] David B. Gustavson. The Scalable Coherent Interface and Related Standards Projects. *IEEE Micro*, pp. 10–22, Feb. 1992.

[GuWe92] Anoop Gupta e Wolf-Dietrich Weber. Cache Invalidation Paterns in Shared Memory Multiprocessors. *IEEE Transaction on Computers*, 41(7):794–810, 1992.

[GWM90] A. Gupta, W. D. Weber, e T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory–Based Cache Coherence Schemmes. *Proc. of Int. Conf. on Parallel Processing*, pp. 312–312, 1990.

[HwBr90] Kai Hwang e Faye Briggs. *Computer Architecture and Parallel Processing.* Prentice–Hall, 1990.

[JLGS90] David V. James, Anthony T. Laundrie, Stein Gjessing, e Gurindar S. Sohi. The Scalable Coherent Interface. *IEEE Computer*, 23(6):74–77, 1990.

[Katz85] Randy H. Katz et alli. Implementing a Cache Consistency Protocol. *Proceedings of 12th Int. Symp. on Comp. Architecture*, pp. 276–283, June 1985.

[Lamp79] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transaction on Computers*, C-28(9):241–248, September 1979.

[Leno90] Daniel Lenoski et alli. The Directory–Based Cache Coherence Protocol for the DASH Multiprocessor. *Proc. 17th Int. Symp. on Computer Architecture*, pp. 148–159, 1990.

[Leno92] Daniel Lenoski et alli. The Stanford DASH Multiprocessor. *IEEE Computer*, pp. 63–79, March 1992.

[MeSc91] John M. Mellor-Crummey e Michael L. Scott. Algorithms for Scalable Synchronization on Shared–Memory Multiprocessors. *ACM Transactions on Computers*, 9(1):21–65, February 1991.

[MHW87] Trevor N. Mudge, John P. Hayes, e Donald C. Winsor. Multiple Bus Architectures. *IEEE Computer*, 20(6):42–48, June 1987.

[MiBa92] Sang Lyul Min e Jean-Loup Baer. Design and Analisis of a Scalable Cache Coherence Scheme Based on Clocks and Timestamps. *Transactions on Parallel and Distributed Systems*, 3(1):25–45, January 1992.

[MPT91] Yeong-Chang Maa, Dhiraj K. Pradhan, e Dominique Thiebaut. Two Economical Directory Schemes for Large–Scale Cache Coherent Multiprocessors. *Computer Architectures News*, 19(5):10–18, 1991.

[Neel87] Francis Neelamkavil. *Computer Simulation and Modelling.* John Wiley and Sons, 1987.

[OkNe90] Brian W. O'Kafka e A. Richard Newton. An Empirical Evaluation of Two Memory–Efficient Directory Methods. *Proc. 17th Int. Symp. on Computer Architecture*, pp. 138–147, 1990.

[PaPa84] M. S. Papamarcos e J. H. Patel. A Low–Overhead Coherence Solution for Multiprocessors with Private Cache Memories. *Proceedings of 11th Int. Symp. on Comp. Architecture*, pp. 348–354, June 1984.

[Pate81] J. H. Patel. Performance of Processor–Memory Interconnections for Multiprocessors. *IEEE Transactions on Computers*, pp. 771–780, Oct. 1981.

[Rash87] Richard Rashid. Designs for Parallel Architectures. *UNIX Review*, pp. 36–43, April 1987.

[ReTh86] Randall Rettberg e Robert Thomas. Contention is No Obstacle to Shared–Memory Multiprocessing. *Communications of ACM*, 29(12):1202–1212, Dec. 1986.

[RuSe84] L. Rudolph e Z. Segall. Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. *Proceedings of 11th Int. Symp. on Comp. Architecture*, pp. 340–347, June 1984.

[SH91] Richard Simoni e Mark Horowitz. Modelling the Performance of Limited Pointers Directories for Cache Coherence. *Proc. 18th Int. Symp. on Computer Architecture*, pp. 309–318, 1991.

[Sieg85] H. J. Siegel. *Interconnection Networks for Large–Scale Parallel Processing.* Lexington Books, Lexington, MA, 1985.

[SJF91] Craig B. Stumkel, Bob Janssens, e W. Kent Fuchs. Address Tracing for Parrallel Machines. *IEEE Computer*, January, 1991.

[Smit82] Alan Jay Smith. Cache Memories. *Computing Surveys*, 14(3):473–530, September 1982.

[Soar90] Luiz Fernando Soares. *Modelagem e Simulação Discreta de Eventos.* VII Escola de Computação - IME-USP, 1990.

[Sten90] Per Stenstrom. A Survey of Cache Coherence Schemes for Multiprocessors. *IEEE Computer*, 23(6):12–24, June 1990.

[Ston87] Harold S. Stone. *High–Performance Computer Architecture.* Addison-Wesley Publishing Company, 1987.

[SUN89] *AT&T C++ Language System Release 2.0, Library Manual.* Sun Microsystems, 1989.

[SwSm86] P. Sweazey e A. J. Smith. A Class of Compatible Cache Consistency Protocols and their Support by IEEE Futurebus. *Proceedings of 13th Int. Symp. on Comp. Architecture*, pp. 414–423, June 1986.

[Thac88] C. P. Thacker et alli. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[Taba90] Daniel Tabak. *Multiprocesors.* Prentice–Hall, 1990.

[Tang76] C. K. Tang. Cache System Design in the Tightly Coupled Multiprocessor System. *Proc. AFIPS*, 45:749–753, 1976.

[ThDe90] Manu Thapar e Bruce Delagi. Stanford Distributed–Directory Protocol. *IEEE Computer*, 23(6):78–80, 1990.

[Yang89] Q. Yang et alli. Analysis and Comparison of Cache Coherence Protocols for a Packet-Switched Multiprocessor. *IEEE Transactions on Computers*, 38(8):1143–1153, August 1989.

[YeFu82] W. C. Yen e K. S. Fu. Coherence Problem in a Multicache System. *Proc. Int. Conf. on Parallel Processing*, pp. 332–339, 1982.