Simpósio Brasileiro de Arquitetura de Computadores
e Processamento de Alto Desempenho

Sistemas de Memória para Multiprocessadores

Roberto A Hexsel*
Departamento de Informática UFPR

Recife, Agosto de 1996

1 Introdução

O objetivo deste tutorial é examinar as alternativas para a implementação de sistemas de memória fisicamente distribuída e logicamente compartilhada para multiprocessadores. Para tanto são estudados os métodos e técnicas empregados no projeto e construção deste tipo de sistema de memória.

As memórias cache para uniprocessadores são especialmente relevantes neste contexto já que estas permitem o acoplamento de processadores rápidos a memórias lentas. O projeto de memórias cache em uniprocessadores é abordado na Seção 2, onde os conceitos e terminologia básicos da área são revisados.

O uso de memórias cache em multiprocessadores propicia os benefícios de menor tempo médio de acesso a memória mas introduz problemas relacionados a consistência de dados, causados pela existência de múltiplas cópias de dados compartilhados. Várias soluções para estes problemas existem que envolvem, em geral, protocolos de coerência que devem ser implementados em hardware por ser essa a forma de se obter maiores velocidades de operação. Contudo, existem implementações híbridas onde as partes mais críticas do protocolo são implementadas em hardware enquanto que partes que são pouco executadas são implementadas em software. Protocolos implementados completamente em hardware são estudados na Seção 3.

Os protocolos de coerência resolvem um problema mas introduzem algumas ineficiências inerentes a seu funcionamento. Por exemplo, quando uma variável compartilhada é atualizada, cópias porventura existentes em caches de outros processadores devem ser invalidadas antes que a atualização da variável possa se concretizar. De outra forma, existiriam no sistema versões diferentes da mesma variável, o que é geralmente, considerado um erro. O modelo de consistência de memória determina os instantes em que a memória se encontra num estado coerente. A definição de consistência de memória e as diferenças entre os vários modelos propostos são discutidas na Seção 4.

2 Caches em Uniprocessadores

Com os avanços na tecnologia de integração de circuitos, hoje é possível, em uma única pastilha, a implementação de processadores com complexidade e sofisticação comparáveis a de "mainframes" de poucos anos atrás. Além do aumento na complexidade, a velocidade também tem crescido a taxas de 55% ao ano. A velocidade da memória dinâmica (RAM) não tem acompanhado a dos processadores. O tempo de acesso de RAMs decresce lentamente, em torno de 33% em dez anos [29]. A velocidade dos processadores cresce a taxas muito mais altas do que a redução no tempo de acesso à memória.

*Depto. de Informática UFPR, Caixa Postal 19081, Centro Politécnico, 81531-970 Curitiba, PR. Voz: 041 267 5244, fax: 041 267 6874, roberto@inf.ufpr.br, http://www.inf.ufpr.br/~roberto.

A técnica empregada para a redução da diferença das velocidades de operação entre processadores e memória é a inserção de um armazenador de alta velocidade entre o processador e a memória principal, como mostra a Figura 1. Este armazenador é chamado de *memória cache*. Como o nome indica, uma memória cache é invisível ao programador *i.e.,* fica escondida entre processador e memória. Quando o processador referencia uma palavra, se ela está contida na cache, a cache fornece o conteúdo da palavra ao processador; senão, a palavra deve ser copiada da memória para a cache e então entregue ao processador. Na próxima vez em que esta palavra for referenciada, ela já estará na cache.

Memórias cache são implementadas em tecnologias com velocidade de acesso semelhante à do processador. Em se tratando de circuitos de memória, maior velocidade significa menor capacidade e, maior custo por bit. O tamanho da cache é uma fração do tamanho da memória principal. Contudo, devido aos padrões de comportamento da imensa maioria dos programas, mesmo com uma cache relativamente pequena é possível uma redução considerável na diferença entre o ciclo do processador e o tempo de acesso à memória percebido pelo processador [53, 29].

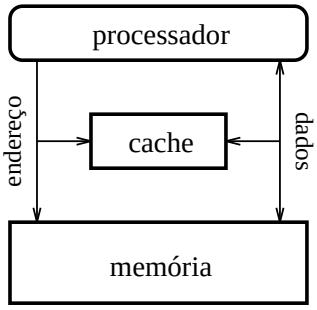

Figura 1: Processador com memória cache.

Organização de memórias cache A Figura 2 mostra a organização de uma memória cache. Cada linha horizontal na figura representa um *bloco* da cache. Cada bloco consiste de três campos. O bit `vál` (válido) indica se o conteúdo do bloco é ou não válido. O campo `etiqueta` permite a identificação do conteúdo do campo de `dados`. O campo de `dados` tem espaço para conter quatro palavras, que são armazenadas em posições adjacentes da memória. A um bloco na cache corresponde uma *linha* na memória.

O endereço de uma palavra é formado por quatro campos. O campo `índice` é usado para escolher um dos blocos da cache. O campo `etiqueta` identifica univocamente o conteúdo do campo de dados. A concatenação da `etiqueta` com o `índice` corresponde aos bits mais significativos do endereço de uma palavra e identificam cada uma das linhas da memória. O campo `pal` (palavra) determina qual das palavras do bloco está sendo referenciada. O campo `byte` indica o byte dentro de uma palavra.

Considere um processador de 32 bits associado a uma cache com 1024 blocos de 64 bytes. São necessários 6 bits para identificar cada byte num bloco ou linha e 10 bits para indexar os 1024 blocos. O campo `byte` corresponde aos bits 0 e 1 e seleciona 1 de 4 bytes numa palavra; o campo `pal` corresponde aos bits 2–5, selecionando 1 de 16 palavras num bloco ou linha. O campo `índice` corresponde aos bits 6–15 e seleciona 1 de 1024 blocos da cache. Dos 32 bits de um endereço, sobram os bits 16–31 para identificar a linha que está no bloco indexado. Estes bits correspondem ao campo `etiqueta`.

O *controlador da cache* consiste de uma máquina de estados que executa as ações necessárias para manter a cache repleta com dados ou instruções sendo referenciados pelo processador. Quando o processador coloca nas suas linhas de endereço o endereço de uma palavra, o controlador da cache compara a etiqueta do bloco apontado pelo `índice` com o campo `etiqueta` do endereço. O bit `vál` indica se o conteúdo do bloco é ou não válido. Se <`vál`=1> e as duas etiquetas forem iguais, ocorreu um *acerto* na cache e a palavra selecionada por `pal` é entregue ao processador. Se <`vál`=0> ou as duas etiquetas forem diferentes, ocorreu uma *falta* na cache e a linha de memória que contém a palavra requisitada deve ser copiada da memória para a cache e então a palavra requisitada é entregue ao processador. Quando o sistema é inicializado, todos os blocos da cache são marcados como inválidos (`vál` $\leftarrow 0$).

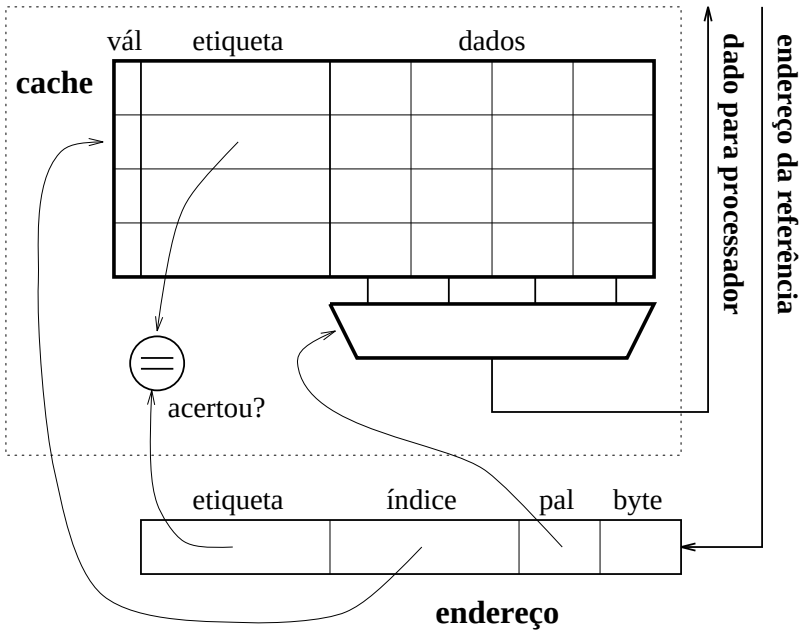

Figura 2: Organização de uma cache.

Princípios da Localidade Uma cache reduz o tempo médio de acesso á memória porque, quando ocorre um acerto, o objeto referenciado é entregue rapidamente ao processador. Quando ocorre uma falta, o objeto deve ser buscado na memória antes de ser entregue ao processador. A eficácia da inclusão de memória cache entre processador e memória advém do comportamento da imensa maioria dos programas. Dois tipos básicos de comportamento são descritos pelos *Princípios da Localidade Temporal* e *Espacial*.

Um processador executa instruções que são armazenadas sequencialmente na memória *i.e.,* a menos de desvios, o processador executa as instruções nos endereços N, N+1, N+2, ... Quando o processador acessa os elementos de um vetor ou matriz, com frequência os acessos referenciam elementos adjacentes da estrutura. Estes dois exemplos ilustram o *Princípio da Localidade Espacial*: os acessos no futuro próximo serão em endereços próximos ao do acesso corrente.

Antes de executar o código de uma função, o processador aloca espaço na pilha para as variáveis locais à função. Enquanto o processador estiver executando a função, as variáveis locais na pilha são acessadas com frequência. Este comportamento é descrito pelo *Princípio da Localidade Temporal*: quando um objeto é acessado, ele será acessado novamente num futuro próximo.

A memória cache pode ser dividida em duas, uma *cache de dados* e uma *cache de instruções*. Esta divisão permite a otimização das caches de forma a reduzir o tempo médio de acesso à memória porque a localidade de acesso a código é diferente da localidade de acesso a dados. A diferença principal entre instruções e dados é que, por princípio, instruções nunca são escritas enquanto que dados podem ser tanto lidos quanto escritos. Além disso, a localidade espacial de dados tende a ser muito pior que a de instruções. Por exemplo, na linguagem C, as variáveis locais referenciadas por uma funçõ são alocados dinamicamente na pilha e variáveis globais são alocadas estaticamente no monte ("heap").

Taxas de Faltas e de Acertos Quando o sistema é inicializado, o conteúdo da cache é todo inválido. Quando o processador faz um acesso à memória, se a palavra não se encontra na cache, ocorre uma falta na cache e a linha apropriada deve ser copiada da memória para a cache. Quando o processador fizer outro acesso a esta mesma palavra, ocorrá um acerto na cache e a palavra é entregue imedia-

tamente ao processador. Por definição, todas as referências que não são acertos são faltas.

A *taxa de acertos* é definida como

$$taxa\ de\ acertos = \frac{número\ de\ acertos}{número\ de\ referências}$$

enquanto que a *taxa de faltas* é definida como

$$taxa\ de\ faltas = 1 - taxa\ de\ acertos\ .$$

Para a maioria dos padrões de localidade exibidos pelos programas, quanto maior for a cache, menor a taxa de faltas, porque a cache vai conter um conjunto maior de objetos sendo referenciados pelo processador. Para caches com tamanho na faixa 64-512Kbytes, taxas de acerto da ordem de 95-99% são comuns [29].

A Figura 2 mostra blocos da cache com quatro palavras. Tamanhos típicos de blocos variam de 1 a 32 palavras. Blocos com várias palavras são usados por causa da localidade espacial. Quando uma instrução é referenciada e ocorre uma falta, as instruções adjacentes a ela no bloco são trazidas para a cache numa busca antecipada implícita. Blocos grandes demais pioram o desempenho das caches porque, para um tamanho fixo da cache, blocos grandes implicam em menor número de blocos distintos. O tamanho ideal deve ser definido com base em simulações com várias combinações de tamanhos e programas aplicativos. Ainda, quanto maior o bloco, maior o tempo consumido na cópia da linha em memória para a cache. Outro problema com blocos grandes é a *poluição da cache*: um bloco muito grande pode conter umas poucas palavras sendo referenciadas enquanto que o restante do bloco não é usado; estas palavras não-usadas ocupam lugares na cache que poderiam ser aproveitados por dados em uso.

Tempo médio de acesso à memória O tempo de acesso à memória RAM é da ordem de 100–500 ciclos de relógio do processador (em 1996). As caches são projetadas para que tenham um tempo de acesso muito próximo daquele do processador, de 1 a 2 ciclos. Em caso de acerto na cache, a referência custa o tempo de acesso à cache. Em caso de falta na cache, a referência custa o tempo de acesso à cache para determinar falta ou acerto mais o acesso à memória para buscar o bloco. Quanto maior o número de acertos, melhor o desempenho do sistema porque o tempo médio de acesso a dados e instruções fica mais próximo do tempo de acesso à cache. Para um tempo de acesso à cache $tCache$ ($\simeq tProc$) e um tempo de acesso à memória $tMem$ ($\gg tProc$), o tempo médio de acesso $tMed$ é dado por

$$tMed = acertos * (tCache + faltas * tMem)$$

onde *acertos* e *faltas* são as taxas de acertos e faltas, respectivamente. A Figura 3 mostra um diagrama de tempos com um acerto e uma falta. A falta incorre no custo do acesso à cache ($tCache$) e à memória (A) mais a cópia de quatro palavras da linha para o bloco ($4 \cdot tC$).

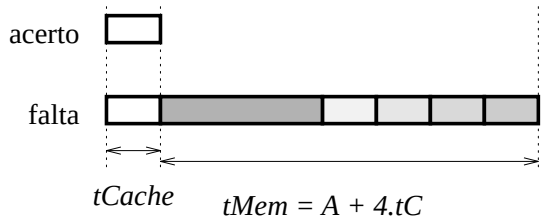

Figura 3: Tempo de acesso à cache e à memória.

Causas de faltas — CCC Quando o processador inicia a execução de um novo programa, a cache não contém nenhuma instrução ou dado daquele programa. Durante um certo intervalo, todas as linhas distintas com instruções e dados referenciados causarão faltas na cache e deverão ser buscados na memória. Estas faltas são chamadas de *compulsórias* porque objetos que são referenciados pela primeira vez necessariamente causam uma falta na cache. Após a fase de inicialização do programa, este já terá referenciado um grande número de posições diferentes de memória. Depois de algum tempo, toda a cache contém objetos já referenciados pelo processador. Referência a uma palavra que ainda não esteja na cache provoca a remoção de uma palavra da cache para abrir espaço para a palavra recém-referenciada. Quando a palavra que for removida é referenciada novamente, a falta que ocorre é chamada de falta por *capacidade i.e.,* a cache não tem capacidade

para conter todo o conjunto de trabalho do processador. Quando duas palavras P e Q tem endereços tais que ambos são mapeados no mesmo bloco da cache, se estas palavras são acessadas frequentemente, P expulsa Q que expulsa P e assim por diante. A falta após a expulsão é chamada de falta por *conflito* por causa do conflito de mapeamento dos endereços das palavras nos blocos da cache.

2.1 Alternativas de Projeto

A estratégia de projeto de caches para uniprocessadores consiste em, dado um conjunto de aplicações, determinar os parâmetros que minimizem o tempo médio de acesso a dados e instruções e, portanto, aumentem a utilização do processador. A escolha apropriada destes parâmetros é o que determina a qualidade e a velocidade de um projeto.

Associatividade Na cache mostrada na Figura 2, cada bloco só pode estar na posição indicada pelo seu `índice`. Este método de mapeamento de linhas em blocos é chamado de *mapeamento direto* [33]. Uma cache *associativa* é aquela em que uma determinada linha pode ser alocada em qualquer dos blocos da cache. A cada referência, todas as etiquetas são comparadas em paralelo para determinar se a palavra desejada está em algum dos blocos. Numa cache com *associatividade N-ária*, uma linha pode estar em um de N blocos. A escolha de um *conjunto* com N blocos é determinada pelo `índice`; dentre os blocos do conjunto indexado, ocorre a comparação em paralelo das N etiquetas para se determinar o acerto ou a falta. A Figura 4 mostra uma cache associativa com um conjunto com 4 blocos, uma cache com dois conjuntos de 2 blocos (*i.e.,* associatividade binária) e, uma cache com 4 conjuntos de um bloco (*i.e.,* mapeamento direto ou associatividade unária).

Para um dado tamanho de cache, quanto maior a associatividade, maior a taxa de acertos. Posto de outra forma, a taxa de faltas de uma cache com mapeamento direto e tamanho N é aproximadamente a mesma que uma cache com associatividade binária de tamanho N/2 [29]. A melhoria na taxa de acertos é consequência da redução nas faltas por conflito. A desvantagem da associatividade mais alta decorre da comparação das etiquetas. Como a cache fica no caminho crítico entre processador e memória, as comparações devem ser feitas em paralelo, o que implica em circuitos maiores que no caso do mapeamento direto. Além disso, o resultado das comparações, se foi um acerto, ainda deve ser selecionado por um multiplexador, o que aumenta o tempo de acesso à cache.

Em caches com associatividade maior que 1, quando ocorre uma falta, o conteúdo de um dos blocos no conjunto onde a linha é mapeada é substituído pela linha recém-lida da memória. A localidade temporal sugere que o melhor candidato a remoção é o bloco que foi acessado no passado mais distante (PMD). Contudo, manter um registro de qual dos blocos num conjunto foi acessado há mais tempo é muito oneroso em termos de implementação. Uma técnica com custo muito reduzido mas que apresenta resultados que aproximam PMD razoavelmente bem consiste em escolher um bloco a esmo e substituí-lo. Um contador com n bits gera uma sequência pseudo-aleatória. Quando um bloco deve ser removido, a vítima é apontada pelo próximo número da sequência.

Hierarquias de caches Processadores de projeto recente incorporam uma cache na própria pastilha de silício. Devido a limitações tecnológicas, esta cache costuma ser relativamente pequena (8–64K) mas opera com tempo de acesso próximo a um ciclo de relógio do processador. Esta cache é chamada de *cache primária* e, em geral, é dividida em cache de instruções e cache de dados. A cache primária é muito pequena para conter o conjunto de trabalho da maioria dos programas. Uma outra cache de tamanho maior, e velocidade menor, é colocada entre a cache primária e a memória. Esta *cache secundária* é construída para operar com tempo de acesso da ordem de 10-20 ciclos e tamanho $\geq$ 128K [62, 53, 13, 41].

As hierarquias de caches são usadas porque, com elas, se atinge um tempo médio de acesso próximo ao da cache primária a um custo por bit próximo ao da memória RAM. O tempo médio de acesso da hierarquia depende da taxa de acertos em cada nível e do custo da carga do bloco desde o nível

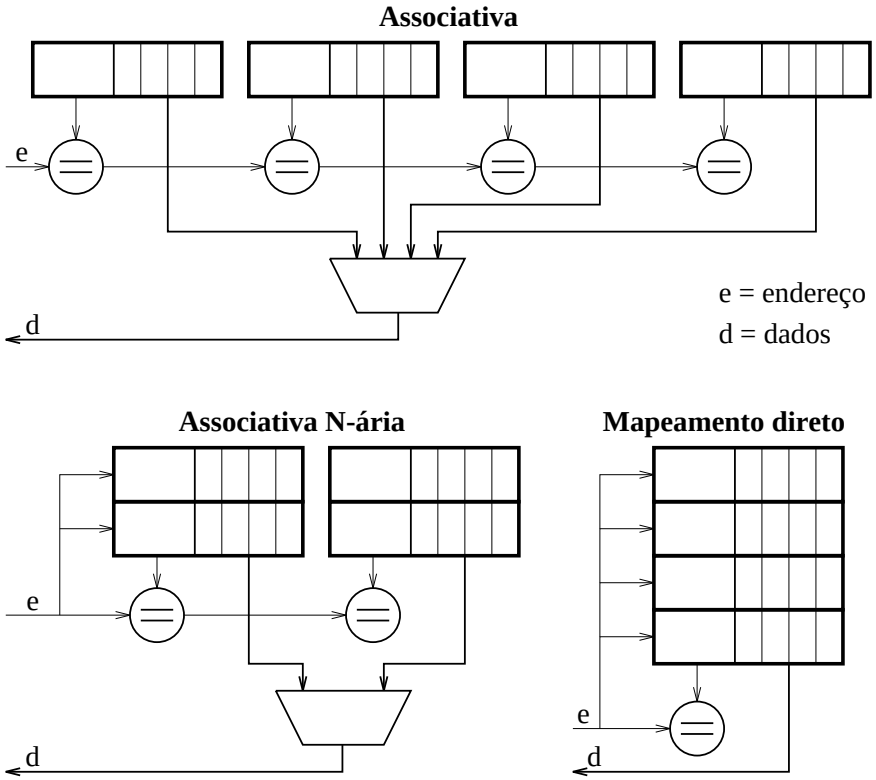

Figura 4: Caches associativas.

imediatamente inferior. A Tabela 1 mostra a ordem de grandeza do custo, em ciclos de relógio do processador, do acesso a uma palavra em cada um dos níveis e o custo da cópia de um bloco com tamanho B do nível imediatamente inferior. O tempo de acesso ao nível n é A_n.

cache	n	acesso	carga (C_n)
primária	1	$O(1)$	$B_1 * A_2$
secundária	2	$O(10)$	$B_2 * A_3$
memória	3	$O(10^2)$	—

Tabela 1: Custo de acesso e carga de blocos.

A Tabela 2 mostra o custo de um acesso a uma palavra nos vários níveis da hierarquia. Um acerto na cache de nível n custa A_n ciclos; uma falta F_n no nível n custa A_n ciclos para testar a presença do bloco mais C_n ciclos para copiar o bloco do nível $n+1$ para a cache do nível n. Note que C_n corresponde a coluna *carga* na Tabela 1.

cache	n	acerto	falta
prim.	1	A_1	$A_1 + C_1$
secund.	2	$A_2 + F_1$	$\sum_{i=1}^{2}(A_i + C_i)$
mem.	3	$A_3 + \sum_{i=1}^{2} F_i$	$\sum_{i=1}^{3}(A_i + C_i)$

Tabela 2: Custo dos acessos na hierarquia.

Uma hierarquia de memória deve manter a *propriedede de inclusão* de dados para garantir a semântica correta dos programas [9]. Uma hierarquia de caches deve garantir que os blocos contidos na cache de nível n também estão presentes no nível $n+1$. Posto de outra forma, a cache do nível n é um subconjunto da cache do nível $n+1$. Esta propriedade é chamada de *Propriedade de Inclusão*. Alguns dos problemas de se manter a inclusão advém da reposição de blocos em caches associativas (todos os blocos do conjunto devem ser examinados) e da reposição de blocos quando níveis diferentes tem blocos de tamanho distintos

– o bloco do nível $n+1$ contém t blocos do nível n e, quando o bloco b_{n+1} é reposto na cache $n+1$, todos os t blocos da cache do nível n, contidos no bloco b_{n+1}, devem ser invalidados.

Políticas de escrita – acertos Quando o processador faz um acesso para escrita e encontra a palavra na cache, uma cache com *escrita forçada* ("write-through") propaga a escrita até a memória, causando um ciclo de atualização na memória. O custo de uma escrita é então o custo de um acesso à memória ($tCache \ll tMem$). A alternativa consiste em escrever o novo valor na cache, sem propagá-lo até a memória. Quando o bloco que sofreu atualizações for substituído por outro, o valor atualizado é então propagado até a memória. Este método é chamado de *escrita preguiçosa* ("write-back") porque ele acumula várias atualizações em uma palavra na cópia da cache e só propaga o resultado até a memória quando da substituição do bloco. Ao contrário da escrita forçada, que sempre mantém a consistência dos dados na cache e em memória, na escrita preguiçosa o conteúdo de alguns blocos na cache pode ser diferente das linhas correspondentes na memória. A escrita preguiçosa se baseia em um bit de estado –`sujo`– que indica se o bloco está "sujo" *i.e.,* se ele sofreu uma atualização. Um bloco sujo deve ser gravado em memória quando for substituído; um bloco "limpo" não causa uma escrita em memória porque seu conteúdo é idêntico ao da linha correspondente.

Fila de escrita Acessos para escrita tem uma frequência relativa mais baixa que acessos para leitura. Escritas perfazem aproximadamente 20-35% das referências a dados [29]. Nas caches com escrita forçada, cada acesso para escrita causa um acesso aos níveis inferiores da hierarquia para sua atualização. Isso faz com que cada escrita incorra no custo de um acesso à memória *i.e.,* o processador fica bloqueado por $O(10^2)$ ciclos. Nas caches com escrita preguiçosa, a reposição de uma linha suja incorre no custo adicional da propagação da escrita até a memória.

A maneira de reduzir este custo consiste na introdução de uma *fila de escrita* entre a cache primária e a cache secundária [40, 29]. O processador insere a referência de escrita na fila; o controlador da cache se encarrega de propagar a referência para os níveis inferiores da hierarquia, liberando o processador para que este prossiga executando instruções e outras referências à memória. Dessa forma, a fila de escrita esconde a latência das escritas reduzindo drasticamente o custo destas referências. A Figura 5 mostra uma fila de escrita entre a cache primária (de dados) e a cache secundária. Filas de escrita são geralmente implementadas com 4 a 8 elementos, com elementos com capacidade para uma palavra de ponto flutuante de precisão dupla, ou para um bloco completo. Quando a fila está cheia, novas escritas bloqueiam o processador até que a escrita menos recente complete, liberando um elemento na fila.

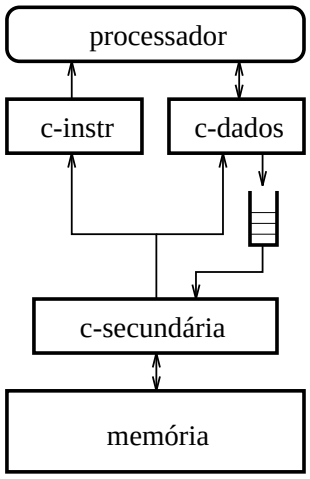

Figura 5: Fila de escrita.

Para que uma fila de escrita seja eficaz, ela deve ser implementada de tal forma que referências de leitura que ocorrem após uma referência de escrita que foi inserida na fila possam completar antes de a escrita ser propagada até a memória. Existem duas possibilidades. Uma leitura em um endereço *diferente* de todas as referências enfileiradas pode ultrapassar todas elas. Uma leitura em um endereço com uma referência enfileirada *deve* retornar o novo valor contido na referência que está na fila. Obedecidas estas restrições, o sistema de

memória mantém a semântica correta dos programas, desde que os intertravamentos internos ao processador resolvam todas as dependências de dados e controle [37, 29].

Políticas de escrita – faltas O controlador de cache pode implementar uma das seguintes políticas para o tratamento de acessos para escrita que provocam uma falta [40, 29]. A política de *busca na escrita* causa a cópia da linha em memória para a cache. Com esta política, um acesso para escrita envolve as ações de uma falta na cache. Com a *alocação de espaço na escrita*, o bloco onde o endereço faltante é mapeado é alocado na cache (talvez causando a escrita de um bloco sujo), embora a linha não seja copiada para a cache. As próximas escritas naquele bloco vão preenchê-lo com os valores atualizados.

A política de *invalidação na escrita* invalida o bloco na cache em caso de falta e insere a referência na fila de escrita. Os próximos acessos ao bloco vão causar sua carga. Em caso de acerto esta política efetua a atualização da palavra na cache. Em caches com escrita forçada, a política de *contorno da cache na escrita* consiste em, quando ocorrer uma falta, depositar a referência na fila de escrita, sem causar qualquer alteração na cache. Quando ocorrer um acerto, o bloco é atualizado. Simulações em [40] indicam que diferentes combinações destas políticas podem reduzir a taxa de faltas em até 50% para alguns programas.

Vazão através da cache Processadores super-escalares podem iniciar mais de um acesso a memória em cada ciclo de relógio [37, 29]. Um dos desafios no projeto de hierarquias de caches para estes processadores consiste em conceber sistemas onde o processador não fique bloqueado esperando pela memória sem necessidade [39, 40, 29]. Como mostra a Figura 5, a divisão da cache primária em duas duplica o número de caminhos entre processador e memória já que o processador pode buscar uma instrução ao mesmo tempo em que referencia uma variável. A fila de escrita também aumenta a vazão de dados ao reduzir o custo das escritas e portanto os bloqueios.

Numa cache com escrita forçada, uma escrita pode ser efetivada em um ciclo porque o processador não necessita esperar pela comparação da etiqueta para saber se houve acerto ou falta. Numa cache com escrita preguiçosa, uma escrita custa dois ciclos: um para a comparação da etiqueta e outro para a efetivação da escrita. Note que a atualização de um bloco sujo antes de saber se ocorreu um acerto causa a perda de seu conteúdo. A escrita preguiçosa reduz o tráfego de escrita na saída da cache (lado próximo à memória) mas aumenta o custo dos acessos à entrada da cache (lado próximo ao processador). A escrita forçada causa mais tráfego na saída da cache enquanto que reduz o custo dos acessos à entrada da cache. Por estas razões, as hierarquias consistem geralmente de caches primárias com escrita forçada e caches secundárias com escrita preguiçosa.

Quando ocorre uma falta na cache de dados, o processador fica bloqueado até que o bloco faltante seja carregado na cache. Um processador super-escalar pode continuar emitindo instruções que não acessam memória mas ficará bloqueado no próximo acesso à memória. Numa *cache não-bloqueante*, uma falta não bloqueia o processador [43]. O bloco na cache onde ocorreu a falta é registrado e a requisição daquele bloco é enviada para a cache secundária enquanto o processador segue executando instruções. As requisições pendentes ficam armazenadas em *registradores de estado de faltas* (REFs). Quando a cache secundária retorna o bloco faltante, este é carregado no local correto na cache primária e, o processador é avisado. O número de faltas pendentes depende do número de REFs disponíveis. Cada falta é memorizada num REF e, enquanto houverem REFs livres, o processador pode continuar emitindo instruções e, talvez, provocando novas faltas. Obviamente, a hierarquia de caches deve ser projetada para aguentar a demanda por dados gerada pelos processadores e pelas otimizações nas caches [56].

Endereçamento virtual ou físico O processador emite referências à memória usando endereços virtuais. Estes são traduzidos para endereços físicos pelo mapeamento virtual–físico contido na *tabela de mapeamento de memória virtual* (TMMV).

A cache primária pode ser acessada com endereços virtuais ou físicos [62, 29]. Se a cache é acessada com endereços virtuais, ela é chamada de *cache virtual* e a consulta à TMMV para a tradução do endereço ocorre em paralelo com a comparação da etiqueta. Todos os blocos de uma cache virtual devem ser invalidados quando ocorre uma troca de contexto porque vários processos podem acessar suas variáveis com os mesmos endereços virtuais. Além disso, um processo de usuário e um do sistema operacional podem acessar o mesmo endereço físico através de dois endereços virtuais diferentes. Estas coincidências de endereços virtuais são chamadas de *sinônimos*. Se a cache não resolver sinônimos corretamente, duas versões da mesma palavra podem co-existir na cache. Uma *cache física* é acessada com endereços físicos. A tradução dos endereços ocorre antes da comparação da etiqueta, o que aumenta o tempo de acesso à cache de meio a um ciclo de processador. Isto pode ser compensado por um segmento adicional no circuito de dados do processador [29].

3 Caches em Multiprocessadores

Como visto na Seção anterior, caches são efetivas na redução do tempo médio de acesso à memória. A função das caches em multiprocessadores com memória compartilhada (MMCs) também é reduzir o tempo médio de acesso à memória. Isso decorre da replicação de variáveis compartilhadas nas caches dos processadores e da migração de dados para a cache do processador que os está usando, como mostrado na Figura 6. A discussão abaixo contempla sistemas com uma hierarquia de caches por processador. As idéias apresentadas se apliquem com relativa facilidade a sistemas com mais de um processador compartilhando uma cache, como por exemplo o DASH [48].

Quando todos os processadores de um multiprocessador operam sobre um mesmo espaço de endereçamento, estes processadores tem acesso a todas as posições de memória, que é logicamente compartilhada por eles. Quando caches são usadas em MMCs, há a possibilidade de que existam várias cópias de variáveis compartilhadas, espalhadas pelas caches do sistema [35, 48, 29]. Se vários processadores podem atualizar uma dada palavra, o multiprocessador deve conter mecanismos tais que todos os processadores tenham uma visão coerente da memória, isto é, todos os processadores devem ter acesso a apenas a versão "mais recente" daquela palavra. De outro modo, caches não seriam transparentes para o programador.

Nesta Seção são investigados os mecanismos e políticas para a manutenção da coerência de memória passíveis de implementação em hardware. A definição de coerência de memória é enunciada na Seção 3.1. As Seções 3.2 e 3.3 descrevem os protocolos de coerência. Os tipos de dados compartilhados são apresentados na Seção 3.4. A Seção 3.5 lista exemplos e referências para descrições detalhadas dos sistemas citados.

3.1 Protocolos de Coerência

Um *infoduto* é uma estrutura que permite a interligação dos processadores, caches e memórias de um MMC. A Figura 6 mostra um multiprocessador com três processadores que se comunicam entre si e com memória através do infoduto. A figura mostra o estado do sistema de memória após os três processadores terem referenciado a variável X, havendo portanto três cópias de X, uma em cada cache. Quando o processador $\mathcal{A}$ atualiza a sua cópia de X para X', os outros dois processadores continuam enxergando o valor desatualizado X. Na maioria dos casos, tal comportamento seria considerado errôneo. Um multiprocessador que contenha algum mecanismo para evitar que esta situação ocorra possui *caches coerentes*.

Definição de coerência Uma definição simplista de um sistema de memória coerente é um sistema onde a leitura de uma variável retorna o resultado da escrita mais recente naquela variável. Num uniprocessador, a noção de "mais recente" é relativamente fácil de se definir com precisão. Num multiprocessador com uma hierarquia de memória, a definição de "operação mais recente" é complicada pela presença das várias caches no sistema e, possivelmente, por diferenças nos tempos de acesso as diversas porções de um espaço de endereçamento fisicamente distribuído.

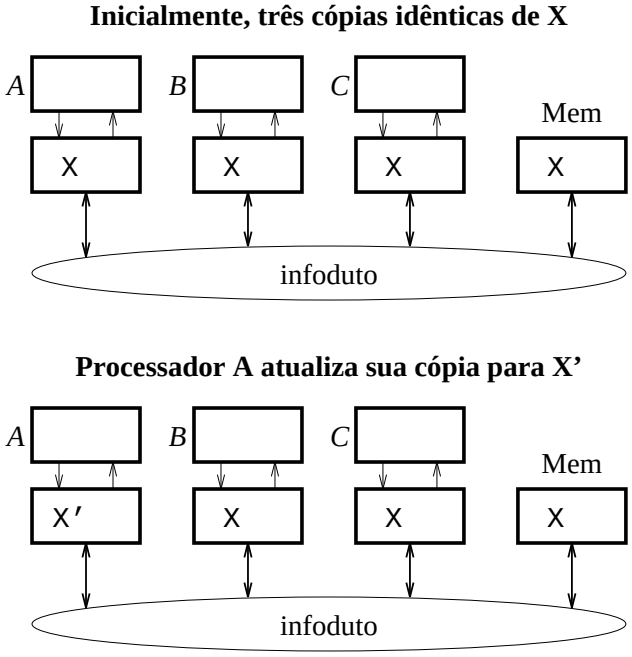

Figura 6: Multiprocessador com memória compartilhada.

Uma definição precisa de coerência de memória é enunciada em [29]:

1. uma leitura de uma variável X pelo processador $\mathcal{P}$ sempre retorna o valor escrito por $\mathcal{P}$, desde que nenhum outro processador tenha escrito em X no intervalo entre a escrita e a leitura de X por $\mathcal{P}$.
2. uma leitura de X por um processador $\mathcal{P}$ após uma escrita em X por outro processador $\mathcal{Q}$ retorna o valor escrito por $\mathcal{Q}$ se o intervalo entre a leitura e a escrita for suficientemente grande e se não ocorreu nenhuma outra escrita em X entre as duas referências.
3. escritas em um mesmo endereço são serializadas: escritas em uma variável por quaisquer dois processadores são percebidas na mesma ordem por todos os processadores.

A primeira propriedade garante que referências à memória ocorrem na mesma ordem em que aparecem no código. A segunda propriedade garante que processadores tenham uma visão coerente da memória. A terceira propriedade garante que o resultado de duas escritas consecutivas numa mesma variável é o valor gravado pela segunda escrita.

Uma vez definido o que seja coerência de memória, os mecanismos empregados para implementar MMCs podem ser descritos. Dois tipos de *protocolos de coerência* são usados em MMCs [5, 57, 50]. Protocolos baseados em *espionagem* são implementados em máquinas cujo infoduto permita a difusão de informação e um custo muito baixo (*i.e.*, um barramento ou seu equivalente topológico, um anel) enquanto que protocolos baseados em *fofoca* são empregados em máquinas onde a difusão de informação tem um custo proibitivo. Para simplificar a descrição que segue, um acesso para leitura que resulta numa falta na cache é uma *falta do tipo L*; uma escrita que resulta numa falta é uma *falta do tipo E*; acertos em acessos de leitura e escrita são *acerto tipo L* e *acerto tipo E*, respectivemente.

Protocolos com atualização Em máquinas construídas com barramentos, o custo da difusão é baixo. Isso torna atrativos os *protocolos com atualização* de cópias. Os controladores das caches do sistema espionam todo o tráfego no barramento. Quando um controlador percebe, no barramento, um ciclo de escrita em um bloco do qual sua cache

contém uma cópia, o controlador atualiza a sua cópia ao replicar em sua cache o ciclo de escrita que ocorre no barramento. Dessa forma, todas as caches sempre contém a versão mais recente de todas as variáveis compartilhadas.

Protocolos com invalidação Nesta classe de protocolos, quando um processador atualiza uma variável compartilhada, seu controlador de cache deve enviar comandos de invalidação para todas as caches que possuam cópia do bloco que contém a variável a ser escrita. As formas de localizar as cópias são discutidas abaixo.

Quando um processador lê uma variável pela primeira vez, ocorre uma falta-L e o processador fica bloqueado até que uma cópia da linha que contém a variável seja gravada em sua cache. Antes de o processador escrever em uma das palavras do bloco, este deve consultar o controlador de memória e enviar comandos de invalidação para as outras caches com cópia do bloco. Quando todas as cópias estão invalidadas, o processador pode então completar o ciclo de escrita. Quando ocorre uma falta-E, o controlador da cache consulta o controlador de memória. Se houverem cópias, estas devem ser invalidadas; senão, a linha é copiada para a cache e o processador completa o ciclo de escrita. Quando um processador referencia uma palavra de um bloco que foi invalidado, ocorre uma falta na sua cache e uma cópia nova e recente deve ser obtida pelo controlador de cache. As ações do protocolo de coerência causam um quarto tipo de falta. Uma invalidação remove um bloco da cache; quando este bloco for referenciado novamente vai ocorrer uma *falta por coerência* (veja pág. 4).

Os protocolos com invalidação são preferidos sobre os com atualização até nos sistemas com barramentos. Num protocolo com atualização, cada escrita provoca um ciclo de atualização em todas as caches que contém cópia do bloco. Mesmo que a escrita só altere uma palavra do bloco, o bloco completo deve ser atualizado nas cópias. Num protocolo com invalidação, somente a escrita inicial causa invalidações. A grande maioria das implementações de coerência de caches existentes se baseia na invalidação de cópias. Os exemplos que seguem são desta classe de protocolos.

Caches coerentes são implementadas com dois conjuntos de etiquetas. Um dos conjuntos é acessado pelo processador, como descrito acima. O segundo conjunto de etiquetas é acessado pelo controlador da cache quando ocorre alguma transação de coerência. Numa atualização, o controlador deve verificar se o bloco que está sendo atualizado existe na cache; se existe, o controlador copia o bloco atualizado para a cache. Numa invalidação, o controlador procura pelo bloco e o invalida, caso ele esteja na cache. A existência de dois conjuntos de etiquetas reduz a competição entre processador e controlador de cache pelo acesso às etiquetas pois ambos podem efetuar as comparações concorrentemente.

3.2 Protocolos com Espionagem

Quando o infoduto permite difusão a baixo custo, os controladores das caches observam o tráfego no barramento. As operações dos processadores sobre dados compartilhados são difundidas através do barramento. Outras caches que contenham cópia do dado recém-modificado atualizam suas cópias com o novo valor.

A Figura 7 mostra um multiprocessador com um módulo de memória e duas caches, interligados por um barramento. Este barramento contém linhas de dados e de endereços além de sinais de controle. A linha Lê/esc indica se o ciclo de barramento é uma leitura <Lê/esc=L> ou uma escrita <Lê/esc=e>. A linha excl indica se alguma das caches contém a versão *exclusiva* (*i.e.*, a única válida) do bloco que está sendo requisitado por alguma outra cache. O sinal habil habilita a passagem de dados da cache para o barramento, dessa forma permitindo a transferência de blocos entre caches e memória. Do ponto de vista do processador, ações executadas pelo seu controlador de cache são *ações locais* enquanto que ações iniciadas pelos controladores das caches dos demais processadores são *ações remotas*.

Protocolos com espionagem dependem de alguns sinais de controle no barramento que são usados para implementar os comandos de coerência. Na Figura 7, quando o controlador de cache $\mathcal{C}_1$ contém a única versão válida da linha L, se a cache $\mathcal{C}_2$ so-

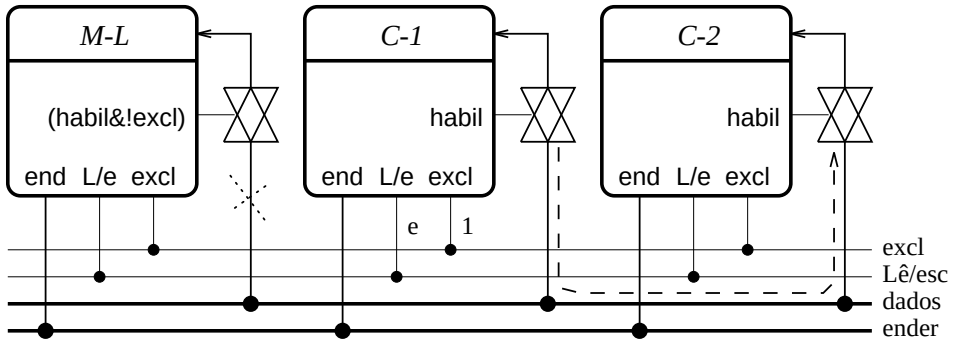

Figura 7: Multiprocessador com memória compartilhada.

licita cópia daquela linha, $\mathcal{C}_1$ deve ser capaz de inibir o controlador de memória $\mathcal{M}_L$ e fornecer a linha solicitada. Para tanto, quando $\mathcal{M}_L$ inicia um ciclo de escrita para copiar L em $\mathcal{C}_2$, o controlador de $\mathcal{C}_1$ ativa sua linha excl (excl=1), inibindo a escrita por $\mathcal{M}_L$ (habil & !excl), enquanto $\mathcal{C}_1$ coloca a sua versão de L nas linhas de dados e faz a linha Lê/esc = e, completando o ciclo de escrita no barramento.

3.2.1 Protocolo Write-Once

Para ilustrar o funcionamento de um protocolo de coerência por espionagem, o protocolo *Write-once* [26, 57] é descrito no que segue. Além das linhas de controle no barramento, a implementação do protocolo implica em que cada bloco na cache contenha, além dos dados, dois bits que indicam o estado do conteúdo do bloco. O controlador de cache efetua as ações prescritas em cada mudança de estado, difundindo comandos de coerência no barramento, efetuando a cópia de dados do barramento ou memória para o bloco, gravando dados sujos em memória, etc.

Neste protocolo, um bloco pode estar em um dos seguintes estados

- `inválido`: o conteúdo do bloco é inválido.
- `válido`: o bloco contém uma cópia com o mesmo valor que a memória.
- `reservado`: ocorreu exatamente uma escrita no bloco e o conteúdo do bloco está com o mesmo valor em memória, que é a única outra cópia que existe.
- `sujo`: ocorreu mais de uma escrita no bloco; seu conteúdo é a única versão válida.

O protocolo usa escrita forçada na primeira escrita em um bloco; após a primeira escrita, o protocolo usa escrita preguiçosa. Quando um bloco `sujo` é substituído na cache, todo o bloco deve ser copiado para a linha em memória. A Figura 8 mostra o diagrama de estados com as possíveis transições e suas causas. As linhas contínuas denotam operações iniciadas pelo processador; linhas tracejadas denotam ações remotas iniciadas pelos controladores das outras caches e difundidas no barramento.

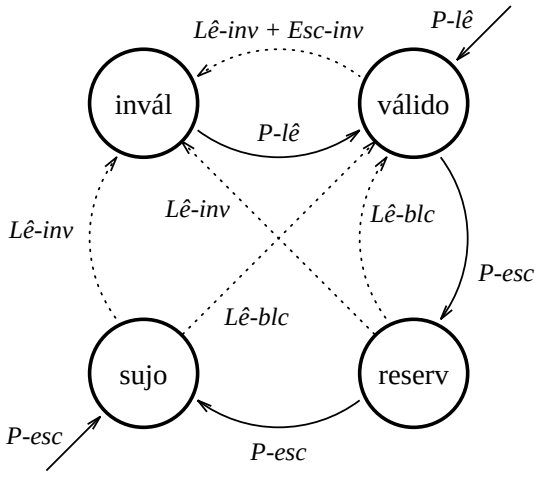

Figura 8: Diagrama de estados de um bloco em cache no protocolo Write-once.

O diagrama de estados mostra as transições que ocorrem em função dos comandos de coerência gerados pelo processador local (P-$*$) ou pelos controladores de cache remotos (Le-$*$ ou Esc-$*$).

- *P-lê*: processador referencia um bloco.
- *P-esc*: processador atualiza um bloco.
- *Lê-blc*: controlador remoto solicita cópia de bloco.
- *Lê-inv*: controlador remoto solicita ao controlador de memória cópia do bloco e a invalidação de outras cópias.
- *Esc-inv*: controlador remoto atualiza bloco e invalida todas as cópias.

As ações dos controladores das caches, em função dos comandos recebidos do processador ou controladores remotos, são especificadas abaixo:

- *Acerto-L* – processador recebe os dados imediatamente. Não há alteração no estado do bloco.
- *Falta-L* – se não há uma cópia `suja`, a memória fornece cópia coerente da linha e o bloco fica `válido`. Se há uma cópia `suja`, a cache que a possui fornece a cópia e a memória é atualizada. Os blocos nas duas caches ficam `válidos`.
- *Acerto-E* – se o bloco está `reservado` ou `sujo`, a atualização ocorre localmente e o novo estado é `sujo`. Se o bloco é `válido`, o comando *Esc-inv* é colocado no barramento, invalidando todas as cópias. A linha na memória é atualizada e o estado do bloco passa a ser `reservado`.
- *Falta-E* – a cópia vem de uma cache com um bloco `sujo` e, a memória atualiza sua linha. Se não existe um bloco `sujo`, a memória fornece a cópia. Um comando *Lê-inv* é difundido no barramento que faz a busca de uma cópia enquanto invalida todas as outras. O bloco é então atualizado e ele fica `sujo`.
- *Reposição* – se o bloco está `sujo`, a linha correspondente é atualizada. Se o bloco está em outro estado, ele é substituído pela linha recém-lida. Dependendo da referência (L ou E), uma das ações acima deve ser executada para trazer uma cópia da nova linha para a cache.

Este protocolo usa dois estados (`reservado` e `sujo`) para reduzir o tráfego causado por escritas. Veja outros exemplos em [19, 57, 35, 29].

3.3 Protocolos com Fofoca

Nos MMCs cujo infoduto seja topologicamente mais complexo que um barramento ou anel, devem ser usados protocolos de coerência baseados na invalidação das cópias. Em geral a atualização de cópias tem custo proibitivo em máquinas baseadas em topologias como tori n-dimensionais ($n \geq 2$) com mais que uns poucos processadores. Um MMC consiste de um conjunto de *nós*. Geralmente, cada nó contém um ou mais processadores com suas caches primárias, uma cache secundária (compartilhada pelos processadores do nó), um bloco de memória principal, um controlador de cache e um controlador de comunicação. Nestas máquinas, a comunicação entre os nós ocorre pela troca de mensagens através do infoduto. A difusão de comandos de atualização a cada escrita para todos os nós do sistema causaria congestionamento constante no infoduto.

Os protocolos de coerência empregados nesta classe de MMCs são baseados na troca de mensagens entre os controladores de cache e de memória distribuídos pelos nós do infoduto. Ao contrário dos protocolos baseados em espionagem, onde os comandos de coerência são difundidos no barramento, numa ação de um protocolo baseado em fofoca, os comandos de coerência são encapsulados em mensagens e remetidos somente para os nós que contém cópias da linha em questão. A localização das cópias fica armazenada em um diretório (discutido abaixo). Para garantir que as mensagens sejam recebidas pelo seu destinatário, a cada mensagem c contendo um comando de coerência corresponde uma mensagem de aceitação a, devolvida pelo destinatário de c ao seu remetente.

3.3.1 Diretórios

Em máquinas onde o custo da difusão é muito alto, os protocolos de coerência mantém a informação sobre a existência e o paradeiro de cópias de uma linha de memória num *diretório*. Antes de um processador atualizar sua cópia de uma variável compartilhada, seu controlador de cache deve consultar o diretório e enviar comandos de invalidação para todas as caches com cópias daquela variável.

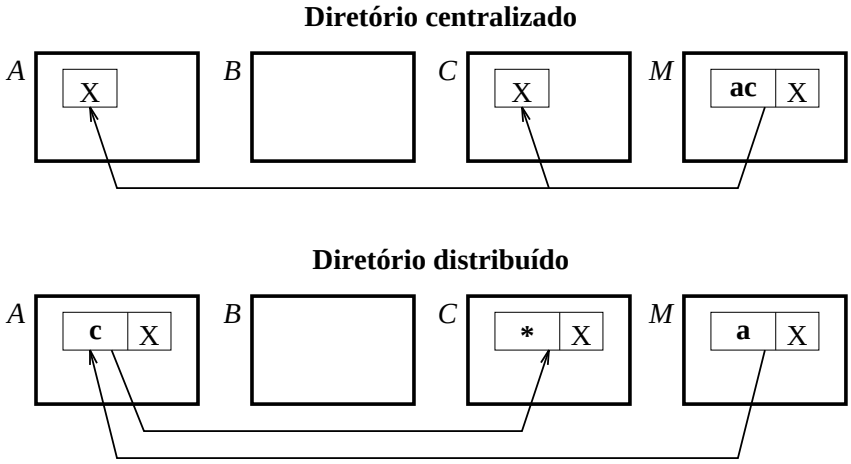

Figura 9: Diretórios centralizados e distribuídos.

No que toca à sua localização no sistema de memória, os diretórios podem ser implementados de forma centralizada em memória ou distribuídos entre memória e caches. A Figura 9 mostra um sistema com três caches e memória. As caches $\mathcal{A}$ e $\mathcal{C}$ contém uma cópia da variável X. Os *diretórios centralizados* podem conter um apontador para cada processador no sistema (diretório total) ou, podem conter um número limitado de apontadores (diretório parcial).

Um *diretório total* contém um vetor de bits associado a cada linha de memória. Cada vetor contém um bit para cada processador. Os bits correspondentes aos processadores com cópia de uma linha são mantidos em **1** enquanto que os outros ficam em **0**. Um *diretório parcial* contém uns poucos apontadores (1–8). Para um sistema com N apontadores, quando o número de cópias for N, a próxima cópia vai exceder o número de de apontadores e para evitar que isso aconteça, todas as cópias existentes são invalidadas. Uma alternativa consiste em invalidar uma das cópias (talvez a usada menos recentemente), liberando dessa forma um apontador para satisfazer a solicitação. Ainda outra alternativa consiste em manter uma lista com as cópias excedentes em memória [17]. Em geral, o número de cópias é pequeno e as duas últimas alternativas são mais eficientes que a primeira. O diretório em memória pode tornar-se um gargalo por causa do número de acessos à informação nele contida. Por causa disso, a distribuição da informação sobre compartilhamento entre vários módulos de memória ou entre caches e memória é usada ao invés da centralização em memória.

Num *diretório distribuído*, a informação sobre o número e paradeiro das cópias é distribuída entre a memória e as caches. A linha em memória contém um apontador para a cache que contém a cópia da linha solicitada mais recentemente. O diretório na cache contém um apontador para outras caches que também possuem cópia do bloco. A cópia mais antiga contém um apontador nulo. Dessa forma, todas as caches que contém cópia de uma certa linha formam uma lista encadeada. Para facilitar as operações de remoção de elementos no meio da lista, o diretório em cada cache contém dois apontadores, um para a próxima cache e outro para a cache anterior na lista. Remoções desta lista ocorrem quando o bloco compartilhado deve ser substituído por outra linha numa das caches. A Scalable Coherent Interface é um exemplo de protocolo baseado em listas encadeadas [59, 36]. Existem propostas de protocolos baseados em árvores ao invés de listas [38, 52]. Para outras alternativas de implementação de diretórios veja [7, 16, 45, 57].

3.3.2 Scalable Coherent Interface

Como exemplo de protocolo baseado em fofoca, a Scalable Coherent Interface (SCI) é descrita a seguir, numa adaptação de [31]. O padrão

IEEE 1596-1992 define três subsistemas da Scalable Coherent Interface: as interfaces de nível físico, o protocolo de comunicação e o protocolo distribuído de coerência de caches.

Nível físico As interfaces físicas consistem de ligações unidirecionais ponto-a-ponto de alta velocidade. A versão mais rápida da interface prescreve uma conexão paralela de 16 bits que pode transferir dados a uma taxa de 1 Gbyte/s. O padrão define um infoduto genérico que pode acomodar até 64K nós. Um nó SCI pode conter um módulo de memória, um processador com suas caches, processadores de E/S ou uma combinação destes. A topologia mais simples consiste de um anel unidirecional com 2 a 20 nós. Em sistemas de grande porte, vários anéis são interligados por comutadores SCI.

Protocolo de comunicação O protocolo de comunicação é baseado na comutação de pacotes e define os tipos e tamanhos dos pacotes, além das ações necessárias para a transferência de dados entre os nós. Um *símbolo* corresponde a uma unidade de transferência de informação do nível físico (*e.g.* 16 bits a cada pulso de relógio de transmissão). Um *pacote*, que consiste de uma sequência contínua de símbolos de 16 bits, contém os endereços do remetente e do destinatário, informação de controle e estado, dados opcionais e uma palavra de paridade.

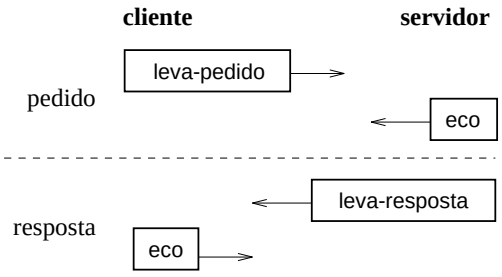

Figura 10: Transação com pedido e resposta.

O protocolo suporta dois tipos de ação: *pedidos* e *respostas*. A Figura 10 mostra as mensagens trocadas entre pedinte e respondente para executar um comando do protocolo de coerência. Uma transação completa, *e.g.* busca de um bloco em outro nó, inicia com o pedinte enviando um pacote do tipo leva-pedido para o respondente. A aceitação do pacote pelo respondente é sinalizada com um pacote eco-pedido. Quando o respondente executa a ação solicitada, ele envia um pacote leva-resposta contendo informação de estado e, possivelmente, dados. O pedinte, ao receber este pacote, completa a transação enviando um pacote eco-resposta ao respondente. Um pacote que transporta um comando de coerência tem 8 símbolos. Um pacote com 64 octetos de dados tem 40 símbolos e, um pacote de eco tem 4 símbolos. O protocolo garante progresso contínuo e contém mecanismos para evitar "deadlock" e "livelock".

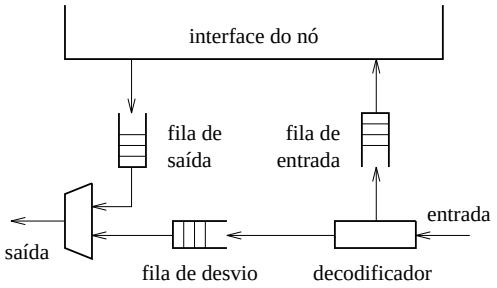

Figura 11: Interface física SCI.

O método de controle de acesso ao meio físico usado na SCI é a inserção de registrador [30]. A Figura 11 mostra um diagrama de blocos da interface física. Um nó retém os pacotes endereçados para si e transfere os demais para a saída. Uma transação inicia quando o pedinte insere um pacote leva-pedido, endereçado ao respondente, na fila de saída. A transmissão deste pacote pode iniciar se a fila de desvio estiver vazia; senão, a transmissão fica bloqueada até que a fila de desvio se esvazie. Na interface do respondente, o decodificador examina os pacotes recebidos e, se destinados a si, os transfere para a fila de entrada. Quando um pacote é aceito, o decodificador gera um pacote de eco endereçado ao pedinte e o insere na fila de saída, no lugar do pacote transferido para a fila de entrada. Se a fila de entrada tiver espaço para o pacote recebido, o eco contém a aceitação do pacote; senão, o eco informa que o pacote foi ignorado. Neste caso, o pedinte deverá retransmitir o pacote leva-pedido. Note que a inserção de um eco

(4 símbolos) no lugar do pacote transferido para a fila de entrada (8–40 símbolos) libera espaço na fila de desvio, possivelmente permitindo o início da transmissão de um pacote retido na fila de saída.

Protocolo de coerência caches O protocolo é baseado na invalidação de cópias e num diretório distribuído com listas duplamente encadeadas. Cada bloco contém dois apontadores de 16 bits, um para o próximo nó e outro para o nó anterior na lista de cópias. Além dos apontadores, o bloco contém bits de estado que indicam se o bloco é *inválido*, *sujo*, etc. Cada linha em memória contém um apontador e bits de estado que indicam a existência e o tipo da lista de cópias. O endereço de uma linha em memória consiste do identificador do nó com 16 bits e um endereço dentro do espaço de endereçamento do nó, com 48 bits. Os próximos parágrafos descrevem as ações necessárias para a criação de uma lista de cópias e, a destruição da lista quando uma das cópias é atualizada.

Considere os processadores $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$, que compartilham, para leitura, uma linha L, que reside no nó $\mathcal{M}$. A sequência de eventos para a criação da lista de cópias é mostrada na Figura 12: setas contínuas denotam apontadores da lista encadeada e setas pontilhadas denotam mensagens transmitidas.

1. Inicialmente, o estado da linha em memória é *única*, indicando que aquela é a única versão do bloco L. Os blocos onde a linha L seria mapeada estão no estado *inválido*.
2. O processador $\mathcal{A}$ solicita ao controlador de memória $\mathcal{M}$ uma cópia da linha L (1). O estado de L muda de *única* para *fora*, indicando que há cópia(s) de L e, uma cópia da linha é remetida para $\mathcal{A}$ (2). O bloco na cache $\mathcal{A}$ passa para o estado *cab*eça da lista.
3. Quando o processador $\mathcal{B}$ pede uma cópia de L (3), $\mathcal{M}$ envia um apontador para $\mathcal{A}$ (4). $\mathcal{B}$ inicia uma transação cache-a-cache (5) pedindo uma cópia de L para $\mathcal{A}$. Quando recebe o pedido, $\mathcal{A}$ muda seu ponteiro para $\mathcal{B}$, envia a cópia solicitada (6) e muda o estado de seu bloco para *fim* da lista. $\mathcal{B}$ passa a ser o novo *cab*eça da lista.

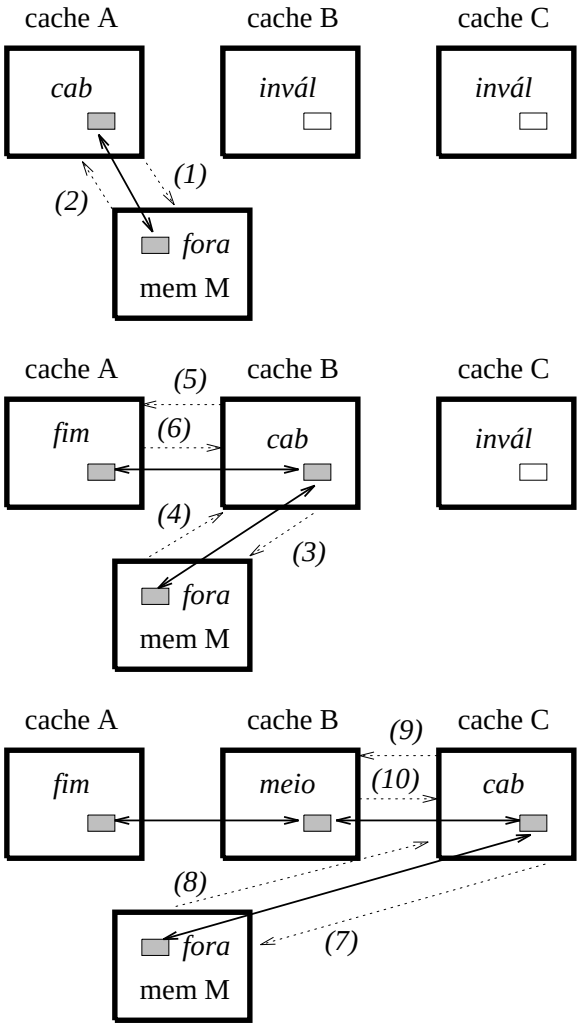

Figura 12: Criação de uma lista de cópias.

4. O nó $\mathcal{C}$ também solicita uma cópia de L (7) e em resposta, $\mathcal{M}$ envia um apontador para $\mathcal{B}$ (8). $\mathcal{C}$ então solicita uma cópia a $\mathcal{B}$ (9), que envia a cópia (10) e muda o estado de seu bloco para *meio* da lista. $\mathcal{C}$ é o novo *cab*eça da lista.

Os controladores de memória de nós SCI redirecionam imediatamente os pedidos de cópias para a cache na cabeça da lista, a fim de evitar congestionamento nos controladores de memória.

Se a cópia de L na cache do processador $\mathcal{B}$ deve ser substituída por alguma outra linha, $\mathcal{B}$ envia uma mensagem para $\mathcal{A}$ com um apontador para $\mathcal{C}$ e uma mensagem para $\mathcal{C}$ com um apontador para $\mathcal{A}$. Dessa forma, $\mathcal{B}$ se desliga da lista de cópias. Note

que, quando da substituição de um bloco, o desligamento de uma cache da lista de cópias pode ser feito após a carga da nova linha, de forma a reduzir o custo da falta. A implementação de uma "fila de desligamentos", similar a uma fila de escrita, para evitar bloqueios no controlador de cache por causa destas operações é sugerida em [42].

O protocolo de coerência de caches da SCI usa de invalidações para manter a coerência entre as várias cópias de uma linha de memória. Antes de atualizar um bloco que é compartilhado com outra(s) cache(s), o processador na cabeça da lista deve expurgar os demais elementos da lista para se tornar dono exclusivo da linha. A Figura 13 mostra a sequência de eventos para que o processador $\mathcal{A}$ possa atualizar sua cópia da linha L:

1. o controlador da cache $\mathcal{A}$ envia um comando de invalidação para $\mathcal{B}$ (1), que invalida sua cópia de L e retorna um apontador para $\mathcal{C}$ (2).
2. O controlador da cache $\mathcal{A}$ então envia um comando de invalidação para $\mathcal{C}$ (3), que invalida sua cópia de L e retorna um indicador de fim-de-lista (4).
3. O estado do bloco em $\mathcal{A}$ passa de *cab*eça da lista para *excl*usivo e o controlador de memória em $\mathcal{M}$ é informado do novo estado da linha (5), que muda para *suja*. O processador pode então prosseguir com a atualização do bloco.

Quando um nó que não é o cabeça da lista deseja atualizar alguma variável, ele primeiro deve tornar-se o cabeça e então expurgar as outras cópias. Se o nó estiver no meio da lista, este deve desligar-se dela, tornar-se o novo cabeça, expurgar as demais cópias e então atualizar a variável. Dependendo do número de cópias, esta operação pode ser muito custosa. O padrão define algumas otimizações que reduzem o custo de algumas operações frequentes [36, 42].

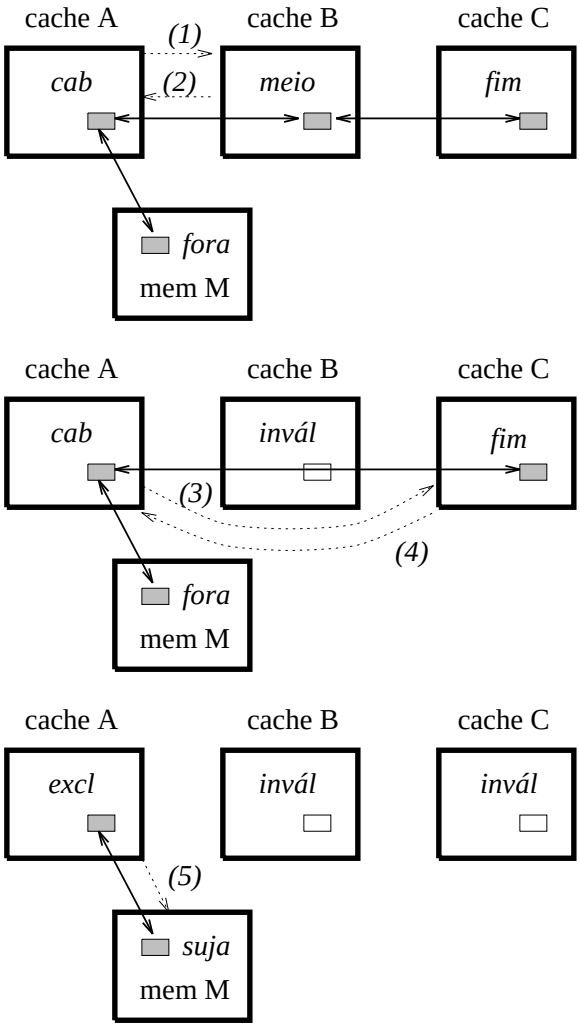

Figura 13: Destruição de uma lista de cópias.

3.4 Tipos de Dados Compartilhados

Como visto acima, algumas operações dos protocolos de coerência podem ser muito custosas. Para que seja possível introduzir otimizações nos protocolos, é necessária a identificação e classificação dos vários padrões de compartilhamento de variáveis. Uma vez identificados os padrões, mede-se a frequência relativa de cada um deles para então ser possível fazer uma avaliação de custo/benefício das otimizações.

Os tipos de dados compartilhados dividem-se nas classes abaixo [63, 55, 12, 48].

- *código e dados privados* – por definição, estes objetos são de uso exclusivo de cada processador. Faltas e acertos são tratados como nos uniprocessadores exceto quando um objeto privado desaloja um objeto compartilhado da cache. Nesse caso, o protocolo de coerência especifica o comportamento do controlador da cache.

- *objetos migratórios* – variáveis compartilhadas que são usadas por somente um processador durante um estado da computação (iteração de um loop). A cada troca de estado, estas variáveis migram de processador em processador (ou de cache em cache). A cada migração, umas poucas cópias (1–2) devem ser invalidadas.
- *dados quase-fixos* – variáveis com uma frequência de leituras muito maior que de atualizações. Quando uma delas é atualizada, várias/muitas cópias devem ser invalidadas.
- *dados atualizados frequentemente* – variáveis com frequência de escritas maior ou igual que a de leituras. A atualização destas variáveis causa poucas invalidações porque, em geral, as escritas são tão frequentes que poucos processadores conseguem obter cópias entre as atualizações.
- *objetos de sincronização* – este grupo consiste das básculas ("locks") e barreiras. Os padrões de compartilhamento dependem do nível de competição pelo objeto. Em caso de pouca competição, uma ou duas cópias devem ser invalidadas a cada atualização. Em caso de muita competição, uma cópia por processo deve ser invalidada a cada atualização. Sobre sincronização veja [49, 1].

Um caso frequente consiste no compartilhamento de uma variável por dois processos (*e.g.* pares produtor–consumidor). O protocolo de coerência da SCI otimiza este tipo de interação com a definição de uma lista especial de forma que a *cab*eça e o *fim* de fila possam atualizar a variável sem envolver o controlador de memória nas transações [36].

Falso compartilhamento Os tamanhos de blocos e linhas de memória em MMCs são escolhidos de forma a minimizar o tráfego no infoduto e, ao mesmo tempo, tirar máximo proveito da localidade espacial. Tamanhos típicos são de 32 a 128 bytes, maiores, portanto, que os usados em uniprocessadores (4–64 bytes). Um problema decorrente do uso de blocos grandes é o *falso compartilhamento* de variáveis [57, 60]. Isso ocorre quando dois processadores referenciam duas variáveis independentes uma da outra mas que foram alocadas na mesma linha de memória pelo compilador. Como as ações de coerência tem a granularidade de um bloco ou linha, mesmo que palavras de estruturas diferentes sejam alocadas no mesmo bloco, elas são tratadas como se compartilhadas fossem. Obviamente, quanto maior o tamanho da linha maior o nível de interferência desnecessária de um processador na computação do outro por causa de ações do protocolo de coerência. Uma solução simples, porém não muito eficiente, consiste em preencher os objetos até que eles ocupem uma ou mais linhas completas. Dessa forma evita-se o falso compartilhamento embora os objetos fiquem maiores que o necessário. O preenchimento pode ser feito tanto manual como automaticamente (pelo compilador). Note que o correto alinhamento de um objeto dentro de uma linha também é necessário.

3.5 Exemplos de MMCs

Existe uma quantidade enorme de publicações sobre MMCs. O que segue é uma pequena lista de exemplos destes sistemas, agrupados por complexidade do infoduto. Para mais exemplos veja as referências em [6, 35, 48].

Barramentos A maioria dos MMCs comerciais são baseados em barramentos: Silicon Graphics [22], Sun-SPARC [15]. Firefly [58].

Hierarquia de barramentos A Data Diffusion Machine, além de usar uma hierarquia de barramentos também implementa uma hierarquia de caches onde o que seria a memória principal também se comporta como uma cache. Este mecanismo é chamado de *memória de atração* ou "Cache Only Memory Architecture" (COMA) [28, 51].

Hierarquia de anéis O KSR-1 consiste de uma hierarquia de anéis com protocolo de espionagem e memória de atração [14]. Express Ring [10, 11] e Hector [61, 21, 34] são hierarquias de anéis com protocolos de coerência altamente otimizados.

Toro N-dimensional A Scalable Coherent Interface [36, 31, 32], já discutida. O Multiplus usa um n-cubo invertido para interligar

agregados com 8 processadores. Seu protocolo de coerência limita o compartilhamento com atualizações dentro de um agregado [8]. O multiprocessador DASH usa um protocolo com invalidação, baseado num diretório completo, distribuído entre caches e memória [45, 47, 48]. O infoduto do DASH consiste de duas malhas bidimensionais para transportar os comandos do protocolo de coerência (uma para pedidos e uma para respostas). DASH ainda suporta modelos fracos de consistência de memória, discutidos na Seção 4.

Troca de contextos O custo das operações de coerência pode ser escondido se, a cada referência a dados compartilhados, ocorre uma troca de contexto interna ao processador. Quando um contexto volta a executar, sua referência remota provavelmente já completou. Esta forma de esconder o custo da coerência é implementada no Alewife [17, 18, 4].

4 Modelos de Consistência

A seção anterior apresentou mecanismos para a manutenção da coerência de memória em MMCs com caches associadas aos processadores. Em sistemas com diretórios, o preço a ser pago é um aumento substancial na latência de escritas a dados compartilhados por causa da invalidação de cópias. Nesta seção são estudadas técnicas para tolerância a latências elevadas associadas a acessos a variáveis compartilhadas em memórias fisicamente distribuídas.

Nos MMCs construídos com barramento que mantém coerência de memória via protocolos com espionagem, atualizações de variáveis compartilhadas são percebidas no mesmo instante por todos os processadores, com diferenças de 1–2 ciclos de relógio. Nos MMCs construídos com infodutos mais complexos e protocolos baseados em diretórios distribuídos, uma referência de escrita pode demorar centenas de ciclos para completar, por causa das invalidações. Devido a condições adversas de tráfego no infoduto, processadores diferentes podem perceber atualizações de duas variáveis em ordens diferentes. Como visto anteriormente, uma fila de escrita possibilita mudanças na ordem de execução de instruções com relação ao programa fonte. A combinação de um infoduto complexo e dispositivos para esconder a latência fazem com que a semântica de algumas instruções seja diferente da esperada pelo programador, cuja intuição causa a expectativa de que as instruções completem na ordem especificada no código fonte.

Esta seção é organizada como segue. O ordenamento de operações de memória é discutido na Seção 4.1. Alguns dos modelos de consistência de memória são descritos na Seção 4.2. Finalmente, a relação de custo/desempenho dos vários modelos e os custos adicionais de implementação são discutidos na Seção 4.3.

4.1 Ordenamento de Operações

Considere o fragmento de código mostrado na Figura 14, onde $\mathcal{P}_1$ e $\mathcal{P}_2$ são processos que executam concorrentemente em dois processadores distintos. Na discussão que segue, processos são identificados com os processadores onde eles executam. Inicialmente, às variáveis A e B são atribuídos zeros, e são trazidas para as caches de $\mathcal{P}_1$ e $\mathcal{P}_2$, respectivamente. Após a segunda atribuição, $\mathcal{P}_1$ referencia B, causando uma falta-L porque B não está em sua cache. O mesmo ocorre com $\mathcal{P}_2$ e A. Existe a possibilidade de que a invalidação de A por $\mathcal{P}_1$ chegue ao controlador de memória após a busca de A por $\mathcal{P}_2$. Neste caso, o resultado deste programa é <X=2, Y=1> quando o programador esperava que o resultado fosse <X=Y=2>.

$\mathcal{P}_1$	$\mathcal{P}_2$
`A = 0;`	`B = 0;`
⋮	⋮
`A = 1;`	`B = 1;`
`X = A + B;`	`Y = A + B;`

Figura 14: Ordenamento de operações.

Um MMC possui um sistema de memória que é *sequencialmente consistente* se a memória serializa todas as referências efetuadas por cada um dos processadores de acordo com o código fonte e as referências dos processadores são tratadas pela

memória como uma mistura das sequências individuais [44]. Este modelo concorda com a intuição de que as instruções em cada segmento de código devem ser executadas na ordem em que elas aparecem no programa. A implementação de consistência sequencial é muito simples: o processador deve ficar bloqueado até que cada referência a dados compartilhados complete (*i.e.,* todas as invalidações sejam efetivadas nas caches remotas). Este modelo é muito restritivo, proibindo o uso de filas de escrita. Em máquinas sequencialmente consistentes, todo o custo dos acessos para escrita é exposto pelo sistema de memória.

Como mostra a Figura 14, se a memória não for sequencialmente consistente, o MMC deve conter mecanismos que garantam a consistência da memória pelo menos em circunstâncias bem determinadas. A vantagem de uma memória *fracamente consistente* advém do uso de técnicas que permitem aos processadores ter mais de um acesso à memória por completar. Considere uma fila de escrita com 4 blocos que causarão uma série de invalidações. Num sistema que *não seja* sequencialmente consistente, o processador pode seguir executando instruções enquanto o controlador de cache se encarrega de completar as escritas pendentes. Deve ficar bem definido para o programador qual é o modelo de consistência implementado no MMC bem como quais são as restrições no ordenamento das operações de memória.

Sincronização de acessos Processadores de projeto recente contém instruções que facilitam a implementação de primitivas de sincronização interprocessos [19, 15, 29]. Instruções que fazem dois acessos inseparáveis à memória (uma leitura seguida de uma escrita no mesmo endereço) são usadas na implementação de *básculas*, ou seja, de exclusão mútua no acesso à regiões críticas. A primitiva `fecha(B)` permite o acesso à região crítica protegida pela báscula `B`, enquanto que `abre(B)` deve ser executada ao sair da região crítica, liberando assim o acesso para outro processo. Um programa que contenha todos os acessos às suas regiões críticas protegidos por básculas é chamado de *sincronizado* [3]. Um processador que executa `fecha(B)` *adquire* a báscula `B`; quando este processador executa `abre(B)`, ele *libera* aquela báscula.

Uma instrução que bloqueie o processador até que a fila de espera se esvazie permite que a sequência estática de execução de instruções seja reestabelecida. Estas instruções são chamadas de *cercas* ("fences") porque as instruções anteriores à cerca devem completar antes de que a cerca complete e instruções posteriores à cerca não podem iniciar antes de que a cerca complete. Uma *cerca-L* impede que as referências de leitura anteriores à cerca completem após ela mesma e que as leituras posteriores iniciem antes dela completar. Uma *cerca-E* tem a semântica equivalente para referências de escrita. Uma *cerca-M* separa temporalmente todas as referências à memória de um processador que estejam pendentes (sendo equivalente a uma cerca-LE). As instruções que implementam básculas, barreiras e cercas são chamadas de *referências de sincronização*. Com estas instruções, o programador pode explicitar os pontos onde a memória deve estar num estado consistente. Note que consistência de memória é um conceito que se aplica a referências a variáveis diferentes enquanto que coerência de memória se aplica a referências para uma mesma variável.

4.2 Modelos de Consistência

Um modelo de consistência especifica a ordem em que os eventos causados por um processo devem ser observados por todos os outros processos. Um processo "observa" as ações de outro ao acessar posições de memória compartilhadas por ambos. Em MMCs com memória fisicamente distribuída, o tempo de acesso a diferentes porções do espaço de endereçamento varia por causa da dispersão dos módulos de memória pelos nós do MMC. Assim, torna-se necessária a definição precisa do que seja "um acesso à memória".

A consistência sequencial é um modelo dito "forte" porque restringe fortemente o ordenamento na execução das instruções. Modelos ditos "fracos" relaxam algumas das restrições e, portanto, permitem a implementação de mecanismos como filas de escrita ou segmentação do caminho de acesso à memória. Definições de quatro modelos

de consistência são enunciadas a seguir. Três primitivas são suficientes para definir um *acesso à memória* [20]:

1. uma *leitura* pelo processador $\mathcal{P}$ é considerada *concluída* com relação ao processador $\mathcal{Q}$ no instante em que a emissão de uma *escrita* no mesmo endereço por $\mathcal{Q}$ não pode afetar o valor retornado pela *leitura*.
2. uma *escrita* por $\mathcal{P}$ é considerada *concluída* com relação a $\mathcal{Q}$ no instante em que uma *leitura* no mesmo endereço por $\mathcal{Q}$ retorna o valor gravado pela *escrita* de $\mathcal{P}$.
3. uma *leitura* é considerada *concluída globalmente* se ela está concluída com respeito a todos os processadores e se a *escrita* que gravou o valor retornado pela *leitura* foi concluída com relação a todos os processadores.

Consistência sequencial (CS) "um sistema multiprocessador é *sequencialmente consistente* se o resultado de qualquer execução é o mesmo que se as operações de todos os processadores forem executadas em alguma ordem sequencial e, as operações de cada processador individual aparecerem nessa sequência na ordem especificada pelo seu programa" [44].

Consistência de processador (CP) As condições que definem CP são:

1. antes de que uma *leitura* possa concluir com respeito a qualquer outro processador, todas as *leituras* anteriores devem estar concluídas.
2. antes de que uma *escrita* possa concluir com respeito a qualquer outro processador, todas as *escritas* e *leituras* anteriores devem estar concluídas.

O modelo CP é mais fraco que CS porque elimina algumas restrições sobre o ordenamento de escritas por processadores diferentes, possibilitando a inclusão de filas de escrita no sistema de memória [27]. Referências de leitura que aparecem no código após uma escrita podem completar antes da escrita. As escritas por um mesmo processador estão sempre na ordem em que aparecem no código fonte: a consistência das escritas é mantida em cada processador. As escritas por processadores diferentes podem estar fora da ordem do código fonte *i.e.,* a ordem das leituras em cada processador pode ser arbitrária desde que estas não envolvam variáveis compartilhadas com outros processadores, que podem escrever nelas.

Consistência fraca (CF) São três as condições que definem CF:

1. todos as referências de *sincronização* devem estar concluídas antes de que uma *escrita* ou *leitura* possa ser concluída com respeito a qualquer outro processador.
2. todas as *escritas* e *leituras* devem estar concluídas antes de que uma referência de *sincronização* possa ser concluída com respeito a qualquer outro processador.
3. referências de *sincronização* são sequencialmente consistentes entre si.

Se todos os acessos de um programa as suas regiões críticas protegidas por básculas, as variáveis de uma região não são de fato compartilhadas enquanto a báscula está fechada. O processador que fechou a báscula pode atualizar as variáveis protegidas desde que, antes de abri-la, todas as escritas pendentes sejam concluídas. Nesse caso, as referências que ocorrem dentro da região crítica podem ocorrer fora de ordem, o que permite a implementações agressivas do caminho de acesso à memória [54, 19, 2, 20]. Estas condições implicam que as filas de escrita sejam esvaziadas antes que uma primitiva `fecha(B)` ou `abre(B)` seja concluída – estas primitivas funcionam como cercas que garantem a conclusão das referências às variáveis compartilhadas.

Consistência na liberação (CL) São três as condições:

1. antes de que uma *leitura* ou *escrita* possa ser concluída com respeito a qualquer outro processador, todas *aquisições* anteriores devem estar completadas.
2. antes de que uma *liberação* possa ser concluída com respeito a qualquer outro processador, todas as *escritas* e *leituras* devem estar concluídas.

3. os acessos de *sincronização* são ordenados segundo consistência de processador.

A primitiva `fecha(B)` é normalmente implementada com uma instrução que faz dois acessos indivisíveis à memória. O estado da báscula fica conhecido no primeiro acesso (*i.e.,* leitura de `B`) enquanto que seu fechamento ocorre no segundo acesso (*i.e.,* escrita em `B`), que corresponde à *aquisição* da báscula. A primitiva `abre(B)` é implementada por uma escrita em `B` e este acesso corresponde a *liberação* da báscula. O modelo CL determina que as sincronizações ocorram segundo a consistência de processador; não há restrições sobre a ordem das demais referências exceto de que as cercas representadas por `fecha(B)` e `abre(B)` devem ser respeitadas [25, 48].

Para que uma implementação de CL satisfaça estas condições, ela deve bloquear o processador até que a aquisição em `fecha(B)` esteja concluída e impedir a conclusão da liberação em `abre(B)` até que todas as referências anteriores completem. Os modelos CF e CL baseiam-se em instruções de sincronização que são identificadas pelos controladores de cache e memória de forma que estes possam bloquear o processador se necessário.

4.3 Custo e Desempenho

A implementação de um MMC com memória CS é simples: o processador fica bloqueado até que cada referência seja concluída. Esta simplicidade tem impacto substancial em termos de eficiência já que o custo de todos os acessos à memória fica exposto. A implementação de um MMC com consistência de processador também é relativamente simples: basta uma fila de escrita que permita que referências de leitura ultrapassem referências de escrita esperando na fila. O controlador de cache deve ser capaz de administrar até duas referências simultâneas à memória e, o sistema de memória deve manter, para cada processador, a ordem relativa das leituras e escritas. O ganho de CP com relação a CS advém da amortização do custo das faltas-E com a sobreposição de leitura(s) com a escrita. Ganhos de desempenho da ordem de 5–25% são possíveis relaxando CS para CP [24].

A implementação dos modelos mais fracos depende de mecanismos que desacoplem as referências emitidas pelo processador das respostas a elas do sistema de memória. As caches devem ser não-bloqueantes para que mais de uma referência possa ficar pendente sem que o processador seja bloqueado desnecessariamente. Os controladores das caches devem manter um registro de todas as escritas pendentes. Nos modelos CF e CL, uma escrita pode prosseguir assim que o bloco estiver disponível para o processador, que faz a atualização e deposita a referência na fila de escrita. O controlador de cache, agora com o com o bloco sujo, deve enviar os comandos de invalidação para todas as cópias e só liberar o elemento da fila de escrita quando todas as confirmações retornarem [45, 46, 48]. Os ganhos em desempenho compensam o custo adicional da implementação. Resultados de simulações descritos em [24] mostram reduções de 10–35% no tempo de execução de programas quando o modelo de consistência é relaxado de CS para CL. Contudo, o programador ou compilador deve explicitar os pontos de sincronização e as seções críticas [23].

A Tabela 3 mostra as restrições no ordenamento das referências impostas pelos modelos. O ordenamento entre duas referências, mostrado como $<A \rightarrow B>$, significa que B não pode ser concluída antes de que A o seja. 'L' corresponde a uma referência de leitura, 'E' de escrita e 'S' a uma referência de sincronização, com 'S_A' sendo uma aquisição e 'S_L', uma liberação. Nos modelos CF e CL, inexistem as restrições $<\{E,L\} \rightarrow \{E,L\}>$.

mod.	restrições
CS	$L \rightarrow L$, $L \rightarrow E$, $E \rightarrow L$, $E \rightarrow E$
CP	$L \rightarrow L$, $L \rightarrow E$, $E \rightarrow E$
CF	$S \rightarrow L$, $S \rightarrow E$, $E \rightarrow S$, $L \rightarrow S$, $S \rightarrow S$
CL	$S_A \rightarrow L$, $S_A \rightarrow E$, $E \rightarrow S_L$, $L \rightarrow S_A$, $S_A \rightarrow S_A$, $S_A \rightarrow S_L$, $S_L \rightarrow S_A$, $S_L \rightarrow S_L$

Tabela 3: Restrições de ordenamento.

A Figura 15 mostra as restrições impostas pelos quatro modelos de consistência de memória defi-

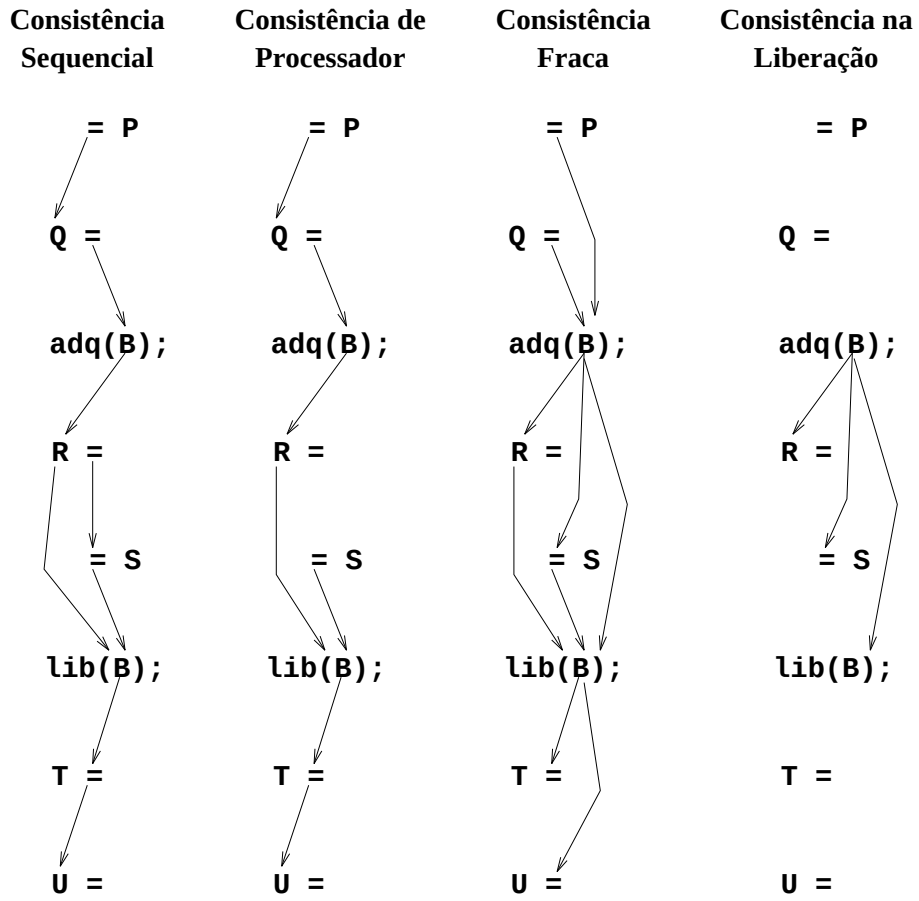

Figura 15: Comparação das restrições no ordenamento pelos modelos de consistência de memória.

nidos acima. Esta figura é baseada na página 717 de [29]. Restrições por transitividade não são indicadas na figura.

5 Conclusão

Este tutorial apresenta algumas das técnicas de projeto de multiprocessadores com memória fisicamente distribuída e logicamente compartilhada (MMCs). Uma rápida introdução ao projeto de caches para uniprocessadores serve de base para a discussão sobre coerência de caches em MMCs. Programadores devem possuir bons conhecimentos sobre hierarquias de memória para tirar proveito da localidade de acessos, assim produzindo código eficiente e veloz.

Sistemas com caches coerentes permitem o compartilhamento de variáveis, oferecendo ao programador um modelo de programação similar ao de uniprocessadores. Contudo, o custo de referências a dados contidos nas caches de outros processadores tende a ser alto, especialmente a escrita em variáveis compartilhadas. Alguns MMCs implementam modelos de consistência de memória que relaxam restrições no ordenamento das referências. Isto permite a execução concorrente de referências, assim escondendo parte do custo das atualizações. Estes modelos com consistência relaxada aumentam a eficiência do MMC mas sua programação exige mais esforço.

Agradecimentos Os avaliadores do SBAC-PAD ofereceram sugestões sobre o conteúdo e tratamento do material. Bruno Müller Jr e Renato J S Carmo fizeram inúmeras sugestões sobre enfoque e apresentação do texto. O autor recebe bolsa RD do CNPQ.

Referências

[1] N M Aboulenein, J R Goodman, S Gjessing, and P J Woest. Hardware support for synchronisation in the Scalable Coherent Interface (SCI). In *Proc of the 8th Intl Parallel Processing Symposium*, pages 141–150, Cancún, 1994. IEEE Comp Soc Press.

[2] Sarita V Adve and Mark D Hill. Weak ordering: A new definition and some implications. Tech Report 902, Computer Sciences Dept, Univ of Wisconsin–Madison, December 1989.

[3] Sarita V Adve and Mark D Hill. A unified formalization of four shared-memory models. *IEEE Trans. on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[4] A Agarwal, R Bianchini, D Chaiken, K L Johnson, D Kranz, J Kubiatowicz, B-H Lim, K Mackenzie, and D Yeung. The MIT Alewife machine: Architecture and performance. In *Proc. 22nd Intl. Symp. on Computer Arch.*, pages 2–13. ACM Comp Arch News 23(2), June 1995.

[5] A Agarwal, R Simoni, J Hennessy, and M Horowitz. An evaluation of directory schemes for cache coherence. In *Proc. 15th Intl. Symp. on Computer Arch.*, pages 280–289, May 1988.

[6] George S Almasi and Allan Gottlieb. *Highly Parallel Computing*. Benjamin-Cummings, 1994. ISBN 0-8053-0443-6.

[7] J Archibald and J-L Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Trans on Computer Systems*, 4(4):273–298, November 1986.

[8] J S Aude, A M Meslin, A A Cruz, C M P Santos, G Bronstein, I N Cota, L F M Cordeiro, M O Barros, M João Jr, S B Pinto, and S C Oliveira. Implementation of the MULTIPLUS/MULPLIX parallel processing environment. In *VII Simp Brasileiro de Arquit de Computadores – Proc de Alto Desempenho*, pages 621–635, Canela, RS, julho 1995.

[9] Jean-Loup Baer and Wen-Hann Wang. On the inclusion properties for multi-level cache hierarchies. In *Proc. 15th Intl. Symp. on Computer Arch.*, pages 73–80, May 1988.

[10] Luiz A Barroso and Michel Dubois. Cache coherence on a slotted ring. In *Proc 1991 Intl Conf. Parallel Processing*, volume 1, pages 230–237, August 1991.

[11] Luiz A Barroso and Michel Dubois. The performance of cache-coherent ring-based multiprocessors. In *Proc. 20th Intl. Symp. on Computer Arch.*, pages 268–277. ACM Comp Arch News 21(2), May 1993.

[12] Mats Brorsson and Per Stenström. Visualising sharing behaviour in relation to shared memory management. Tech Report Dt-150, Dept of Computer Engineering, Lund Univ, December 1992. In Proc. of 1992 Intl Conf. on Parallel and Distributed Systems, Hsinchu, Taiwan, December 1992, pages 528–536.

[13] H O Bugge, E H Kristiansen, and B O Bakka. Trace driven simulations for a two-level cache design in open bus systems. In *Proc. 17th Intl. Symp. on Computer Arch.*, pages 250–259. ACM Comp Arch News 18(2), May 1990.

[14] H Burkhardt, S Frank, B Knobe, and J Rothnie. Overview of the KSR1 computer system. Tech Report KSR-TR-9202001, Kendall Square Research, Boston, 1992.

[15] Ben Catanzaro. *Multiprocessor System Architectures*. Prentice-Hall, 1994. ISBN 0-13-089137-1.

[16] D Chaiken, C Fields, K Kwihara, and A Agarwal. Directory-based cache coherence in large-scale multiprocessors. *IEEE Computer*, 23(6):49–59, June 1990.

[17] D Chaiken, J Kubiatowicz, and A Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *4th Intl. Conf. on Arch'l. Support for Progr. Lang. and Oper. Sys.*, pages 224–234. ACM Comp Arch News 19(2), April 1991.

[18] David Chaiken and Anant Agarwal. Software-extended coherent shared memory: Performance and cost. In *Proc. 21st Intl. Symp. on Computer Arch.*, pages 314–324. ACM Comp Arch News 22(2), April 1994.

[19] M Dubois, C Scheurich, and F Briggs. Synchronisation, coherence and event ordering in multiprocessors. *IEEE Computer*, 21(2):9–21, February 1988.

[20] Michel Dubois and Christoph Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Trans. on Software Engineering*, 16(6):660–673, June 1990.

[21] K Farkas, Z Vranesic, and M Stumm. Cache consistency in hierarchical ring-based multiprocessors. Tech Report EECG TR-92-09-01, Univ

of Toronto, 1992. Also in Proc. of Supercomputing '92.

[22] Mike Galles and Eric Williams. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. White paper, Silicon Graphics Computer Systems, 1995.

[23] K Gharachorloo, S V Adve, A Gupta, J L Hennessy, and M D Hill. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing*, 2(15):399–407, 1992.

[24] K Gharachorloo, A Gupta, and J Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *4th Intl. Conf. on Arch'l. Support for Progr. Lang. and Oper. Sys.*, pages 245–257. ACM Comp Arch News 19(2), April 1991.

[25] K Gharachorloo, D Lenoski, J Laudon, P Gibbons, and J L Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17th Intl. Symp. on Computer Arch.*, pages 15–26. ACM Comp Arch News 18(2), May 1990.

[26] James R Goodman. Using cache memory to reduce processor-memory traffic. In *Proc. 10th Intl. Symp. on Computer Arch.*, pages 124–131, June 1983.

[27] James R Goodman. Cache consistency and sequential consistency. Tech Report 1006, Computer Sciences Dept, Univ of Wisconsin–Madison, February 1991. Also published as SCI Committee Report 61, March 1989.

[28] E Hagersten, P Andersson, A Landin, and S Haridi. A performance study of the DDM – a cache-only architecture. Tech Report R91:17, Swedish Inst of Computer Science, November 1991.

[29] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2nd edition, 1996. ISBN 1-55860-329-8.

[30] Roberto A Hexsel. *Redes de Dados: Tecnologia e Programação*. a ser publicado, 1996.

[31] Roberto A Hexsel and Nigel P Topham. The performance of SCI multiprocessor rings. *Journal of the Brazilian Computer Society*, 1(2):24–37, July 1995.

[32] Roberto A Hexsel and Nigel P Topham. The performance of cache coherency in SCI-based multiprocessors. In *VIII Simp Bras de Arquit de Computadores – Proc de Alto Desempenho*, pages 47–56, Recife, agosto 1996.

[33] Mark D Hill. A case for direct-mapped caches. *IEEE Computer*, 21(12):25–40, December 1988.

[34] Mark Holliday and Michael Stumm. Performance evaluation of hierarchical ring-based shared memory multiprocessors. *IEEE Trans. on Computers*, C-43(1):52–67, January 1994.

[35] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, 1993. ISBN 0-07-031622-8.

[36] IEEE. *IEEE Std 1596-1992 – Standard for Scalable Coherent Interface*. IEEE, 1992.

[37] Mike Johnson. *Superscalar Microprocessor Design*. Prentice-Hall, 1991. ISBN 0-13-875634-1.

[38] Ross E Johnson. *Extending the Scalable Coherent Interface for Large-Scale Shared-Memory Multiprocessors*. PhD dissertation, Computer Sciences Dept, Univ of Wisconsin–Madison, February 1993. Also Tech Report 1136.

[39] Norman P Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. 17th Intl. Symp. on Computer Arch.*, pages 364–373. ACM Comp Arch News 18(2), May 1990.

[40] Norman P Jouppi. Cache write policies and performance. In *Proc. 20th Intl. Symp. on Computer Arch.*, pages 191–201. ACM Comp Arch News 21(2), May 1993.

[41] Norman P Jouppi and Steven J E Wilson. Tradeoffs in two-level on-chip caching. In *Proc. 21st Intl. Symp. on Computer Arch.*, pages 34–45. ACM Comp Arch News 22(2), April 1994.

[42] A Kägi, N Aboulenein, D C Burger, and J Goodman. An analysis of the interactions of overhead-reducing techniques for shared-memory multiprocessors. In *Proc of the Intl Conf on Supercomputing (ICS95)*, pages 11–20, Barcelona, July 1995. ACM Press.

[43] D Kroft. Lock-up free instruction fetch/prefetch cache organization. In *Proc. 8th Intl. Symp. on Computer Arch.*, pages 81–85, June 1981.

[44] Leslie Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, September 1979.

[45] D Lenoski, J Laudon, K Gharachorloo, A Gupta, and J L Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proc. 17th Intl. Symp. on Computer Arch.*, pages 148–159. ACM Comp Arch News 18(2), May 1990.

[46] D Lenoski, J Laudon, K Gharachorloo, W-D Weber, A Gupta, J L Hennessy M Horowitz, and M S Lam. The Stanford Dash multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[47] D Lenoski, J Laudon, T Joe, D Nakahira, L Stevens, A Gupta, and J Hennessy. The DASH prototype: Implementation and performance. In *Proc. 19th Intl. Symp. on Computer Arch.*, pages 92–103. ACM Comp Arch News 20(2), May 1992.

[48] Daniel E Lenoski and Wolf-Dietrich Weber. *Scalable Shared-Memory Multiprocessing.* Morgan Kaufmann, 1995. ISBN 1-55860-315-8.

[49] John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans on Computer Systems*, 9(1):21–65, February 1991.

[50] Shubhendu S Mukherjee and Mark D Hill. An evaluation of directory protocols for medium-scale shared-memory multiprocessors. In *Proc of the Intl Conf on Supercomputing (ICS94)*, pages 64–74, Manchester, July 1994. ACM Press.

[51] H L Muller, P W A Stallard, and D H D Warren. An evaluation study of a link-based data diffusion machine. In *Proc of the Intl Workshop on Support for Large Scale Shared Memory Architectures*, pages 115–128, Cancún, April 1994. With 8th IPPS.

[52] Hakan Nilsson and Per Stenström. The scalable tree protocol – a cache coherence approach for large-scale multiprocessors. Tech Report Dt-149, Dept of Computer Engineering, Lund Univ, December 1992. In Proc. of 4th IEEE Symp. on Parallel and Distributed Processing, December 1992, pages 498–506.

[53] Steven A Przybylski. *Cache and Memory Hierarchy Design: a Performance-Directed Approach.* Morgan Kaufmann, 1990. ISBN 1-55860-136-8.

[54] Christoph Scheurich and Michel Dubois. Correct memory operation of cache-based multiprocessors. In *Proc. 14th Intl. Symp. on Computer Arch.*, pages 234–243. ACM Comp Arch News 15(2), June 1987.

[55] J P Singh, W-D Weber, and A Gupta. SPLASH: Stanford ParalleL Applications for SHared-memory. Technical Report CSL-TR-91-469, Computer Science Dept, Stanford Univ, April 1991. Also in ACM SIGARCH Comp Arch News 20(1).

[56] Gurindar Sohi and Manoj Franklin. High-bandwidth data memory systems for superscalar processors. In *4th Intl. Conf. on Arch'l. Support for Progr. Lang. and Oper. Sys.*, pages 53–62. ACM Comp Arch News 19(2), April 1991.

[57] Per Stenström. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 23(6):12–24, June 1990.

[58] C P Thacker, L C Stewart, and E H Satterthwaite Jr. Firefly: A multiprocessor workstation. *IEEE Trans. on Computers*, C-37(8):909–920, August 1988.

[59] Manu Thapar and Bruce Delagi. Cache coherence for large scale shared memory multiprocessors. *ACM SIGARCH Comp. Arch. News*, 19(1):114–191, March 1991. Reprinted from Proc of SPAA 1990.

[60] J Torrelas, M S Lam, and J L Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Trans. on Computers*, C-43(6):651–663, June 1994.

[61] Z Vranesic, M Stumm, D Lewis, and R White. Hector: a hierarchically structured shared memory multiprocessor. *IEEE Computer*, 24(1):72–78, January 1991.

[62] W-H Wang, J-L Baer, and H M Levy. Organization and performance of a two-level virtual-real cache hierarchy. In *Proc. 16th Intl. Symp. on Computer Arch.*, pages 140–148. ACM Comp Arch News 17(3), May 1989.

[63] Wolf-Dietrich Weber and Anoop Gupta. Analysis of cache invalidation patterns in multiprocessors. In *3rd Intl. Conf. on Arch'l. Support for Progr. Lang. and Oper. Sys.*, pages 243–256. ACM Comp Arch News 17(2), April 1989.