Coerência de Memórias Cache e Modelos de Consistência de Memória

Renato Silva das Neves
RA 057639
Universidade Estadual de Campinas
Instituto de Computação
Campinas, Brasil
renato.neves@students.ic.unicamp.br

ABSTRACT

*A*rquiteturas de computadores com memória compartilhada e múltiplos processadores, cada um com seu esquema de memórias cache, têm que considerar problemas de coerência de dados entre memória e caches. Existem duas abordagens principais em hardware que surgiram para manter a coerência, uma baseada em monitoração (*snooping*) e outra baseada em diretórios. Vários protocolos implementam tais abordagens e alguns deles serão discutidos neste trabalho. A questão sobre modelos de consistência de memórias em sistemas com multiprocessadores, tais como consistências sequencial e relaxadas, também será abordada.

Keywords

Coerência de cache, consistência de memória, *snooping*, diretórios, *write-once*, berkeley, illinois, firefly, consistência sequencial, consistência relaxada.

1. INTRODUÇÃO

Atualmente, é comum observar os processadores atuais com um ou dois níveis de memória cache. Caches são memórias rápidas que possuem cópias dos dados e instruções da memória principal usados mais recentemente, fornecendo um acesso com menor latência para o processador [1]. Assim, o uso de memórias cache surgiu como um método eficiente para reduzir o tempo de acesso médio à memória em computadores monoprocessados, pelo fato de explorar a localidade de referências à memória no tempo (localidade temporal) e no espaço (localidade espacial), fazendo com que a maioria das requisições de memória sejam supridas pela própria cache, evitando assim acessos à memória principal, que é um gargalo nos sistemas computacionais modernos.

Com a dificuldade em aumentar a velocidade dos microprocessadores atuais, a tecnologia de arquiteturas multiprocessadas com memória compartilhada está se tornando cada vez mais uma alternativa viável para o desenvolvimento de sistemas com melhor desempenho. Por exemplo, chips multicore não são mais restritos a apenas servidores ou sistemas de alto-desempenho, já estão se expandindo para computadores desktop, laptops, videogames e em outra série de dispositivos [8]. Daqui a alguns anos será difícil encontrar sistemas com um único processador disponíveis no mercado.

Em arquiteturas multiprocessadas o uso de caches também permite uma melhora em desempenho, evitando na maioria dos casos a latência da memória principal. Usando uma memória principal compartilhada, o compartilhamento de código e dados se torna mais simples entre os processos sendo executados em um ambiente paralelo (multiprocessado). Esse compartilhamento pode resultar em várias cópias de um mesmo bloco em uma ou mais caches ao mesmo tempo. Se os processadores atualizarem suas cópias livremente, os dados poderão ser vistos de forma incoerente, com um processador tendo um valor diferente de um mesmo dado presente em um bloco em relação a outro processador. Esse é o chamado problema de coerência de memórias cache e algum mecanismo deve ser implementado, em software ou hardware, para evitar tal problema. Protocolos em hardware adicionam um hardware especial que detecta acessos à cache e implementa um protocolo de coerência de cache que é transparente ao programador e compilador [6]. Em uma abordagem em software, o compilador deve gerar código consistente, se beneficiando do mecanismo de cache e de algumas instruções especiais que podem ser usadas para manter a coerência.

A coerência de memórias cache poderia ser garantida simplesmente se o valor retornado por uma instrução de carga fosse sempre o último valor gravado por uma instrução de armazenamento para uma mesma posição de memória. Porém, como vários processadores estão executando instruções em paralelo de forma assíncrona, garantir esta propriedade não é uma tarefa trivial. Assim, um modelo de consistência de memória deve garantir aos programadores uma visão da ordem de execução no tempo de instruções de armazenamento e carga na memória, em diferentes processadores, assim como permitir operações de sincronização, a fim de que a consistência possa ser mantida [7].

As seções a seguir irão abordar em maiores detalhes aspectos sobre a coerência de memórias cache, mostrando alguns protocolos implementados em hardware que se baseiam em monitoração e em diretórios. Uma série de protocolos de monitoração serão apresentados, refletindo diferentes implementações encontradas nos sistemas atuais. Uma breve discussão sobre modelos de consistência de memórias também será apresentada, seguida pela conclusão do trabalho.

2. COERÊNCIA DE MEMÓRIAS CACHE

A coerência de memórias cache pode ser realizada por software ou por algum harware especial. As soluções por software geralmente se baseiam em uma análise dos códigos dos programas pelo compilador, que pode verificar conjuntos de dados compartilhados e, por exemplo, não permitir que sejam armazenados na cache. Isso pode levar a uma utilização ineficiente da cache, visto que alguns dados compartilhados podem ser de uso exclusivo de um processador em algum intervalo de tempo. Outras técnicas permitem uma análise mais detalhada dos códigos, procurando determinar períodos em que blocos compartilhados podem ser mantidos na cache e inserindo novas instruções para manter a coerência nestes períodos. As abordagens em software têm a vantagem de evitarem o uso de hardware adicional para manter a coerência de cache, mas por serem efetuadas em tempo de compilação, tendem a tomar decisões que não aproveitam a cache da melhor maneira possível [10].

As soluções em hardware devem ser capazes de detectar acessos à memória incoerentes e garantir a coerência dos blocos da cache (invalidando ou atualizando o bloco), em tempo de execução [7]. Basicamenta tais protocolos se dividem em duas classes: protocolos de monitoração e protocolos baseados em diretório. Os protocolos de monitoração (*snoopy protocols*) são implementados nos controladores de cada cache e ficam monitorando o barramento para saber se eles possuem alguma cópia do bloco solicitado. Assim, as informações de compartilhamento do bloco são mantidas em cada cache. Em contrapartida, o esquema baseado em diretório mantém as informações de cópias e compartilhamento dos blocos da cache de todos os processadores centralizado em um local, chamado diretório. Esses esquemas serão abordados com mais detalhes nas próximas seções.

2.1 Protocolos de Monitoração

Existem dois protocolos de monitoração básicos, chamados protocolo de invalidação de gravação e protocolo de atualização de gravação. No protocolo de invalidação de gravação, a cada escrita realizada em um bloco de uma cache de um processador, uma mensagem é enviada pelo barramento com o objetivo de invalidar todas as cópias presentes nos demais processadores. Assim, esse processador fica com acesso exclusivo à linha da cache para efetuar operações de escrita, dando a idéia de leitores-escritores, ou seja, podem haver vários leitores para uma linha da cache, mas apenas um escritor. No caso do protocolo de atualização de gravação, ao ser realizada uma escrita em uma linha da cache por um processador, ela é atualizada em todas as cópias presentes nas caches dos outros processadores.

Em geral, o mais eficiente desses protocolos depende do padrão de leituras e escritas à memória [10]. A abordagem de invalidação é a mais usada nos sistemas multiprocessados atuais, pois sugere menor tráfego no barramento. Assim, uma série de protocolos de invalidação serão descritos a seguir.

2.1.1 *Write-Once*

Historicamente é o primeiro protocolo de invalidação que foi proposto na literatura [3] (figura 1). Nesse esquema, os blocos em uma cache podem estar em um de quatro estados: inválido, válido (não modificado, possivelmente compartilhado), reservado (não é necessário um *write-back* e é a única

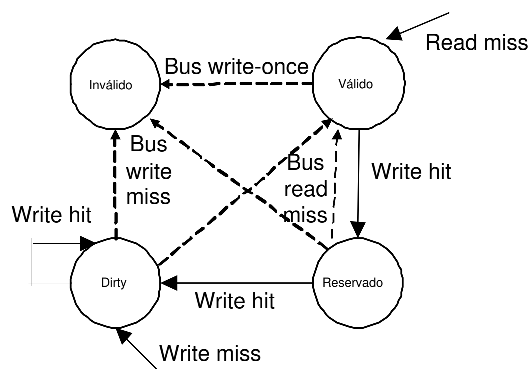

Figure 1: Protocolo Write-Once.

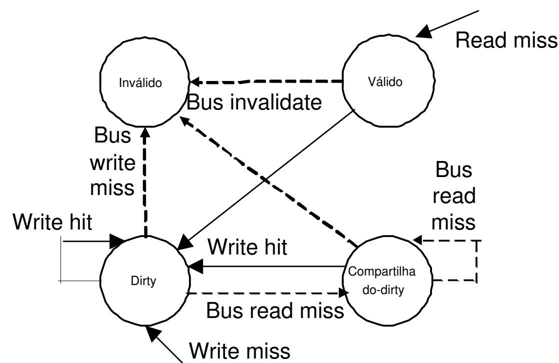

Figure 2: Protocolo Berkeley.

cópia em qualquer cache) e *dirty* (é necessário um *write-back* e é a única cópia em qualquer cache).

Em uma falha de leitura (*read miss*), o bloco que está no estado *dirty* fornece o bloco solicitado, realizando um *write back* para a memória. Caso nenhum bloco está no estado *dirty*, então o bloco vem da memória. Todas as caches com uma cópia do bloco definem seu estado como válido.

Em um acerto de escrita (*write hit*), caso o estado do bloco seja *dirty*, a escrita é realizada sem alteração de estado. Se o estado for reservado, então o estado do bloco é mudado para *dirty*. Se o estado for válido, o bloco é gravado na memória por *write-through* e seu estado mudado para reservado. O estado válido quer dizer que é um dado compartilhado e deve ser colocado no barramento a informação para as outras cópias serem invalidadas.

Em uma falha de escrita (*write miss*), o bloco é carregado da memória ou do bloco de outra cache cujo estado seja *dirty*. Uma vez que o bloco foi carregado, a escrita é realizada e o estado é modificado para *dirty*. Todas as outras caches invalidam suas cópias.

2.1.2 *Berkeley*

Desenvolvido pela Universidade da Califórnia para ser aplicado em uma máquina RISC multiprocessada, esse esquema (figura 2) usa transferências diretas entre caches [5]. Seus possíveis estados são: inválido, válido (não modificado, possivelmente compartilhado), compartilhado-*dirty* (modificado e possivelmente compartilhado) e *dirty* (modificado e não há nenhuma outra cópia em caches). Um bloco no estado compartilhado-*dirty* ou *dirty* só pode estar em uma cache,

Figure 3: Protocolo Illinois.

Figure 4: Protocolo Firefly.

mostrando que a cache é a proprietária do bloco. A diferença é que se em uma cache o estado é compartilhado-*dirty*, nas outras caches o bloco pode ser válido. Blocos no estado compartilhado-*dirty* e *dirty*, quando escolhidos para substituição na cache, são gravados na memória principal por *write-back*.

Em uma falha de leitura, a cache que tem o bloco nos estados compartilhado-*dirty* ou *dirty* transfere diretamente o bloco para a cache que fez o pedido, mudando seu estado para compartilhado-*dirty*. Se nenhuma cache possui o bloco ou ele está em qualquer outro estado, então o bloco vem da memória principal. O bloco da cache que fez o pedido tem seu estado alterado para válido.

Em um acerto de escrita, se o estado do bloco é *dirty*, então o estado não é alterado. Se o estado é válido ou compartilhado-*dirty*, o estado é alterado para *dirty* e um *broadcast* é feito pelo barramento para que todas as outras caches possam invalidar suas cópias.

Em uma falha de escrita, o bloco é colocado no estado *dirty* e todas as cópias em outras caches são invalidadas.

2.1.3 *Illinois*

Este método [9] (figura 3) se baseia no fato de que falhas de acesso a blocos podem ser tratadas procurando por blocos em outras caches ou na memória principal e também assume que a cache que requisitou o bloco pode determinar a origem do bloco. A melhora de performance proposta por esse método é que ele percebe que invalidações para acertos de escrita não são necessárias em blocos privados não modificados, pois é possível determinar a partir dos estados se o bloco é ou não compartilhado. Assim como os anteriores, também são usados quatro estados: inválido, válido-exclusivo (não modificado, é a única cópia em todas as caches), compartilhado (não modificado e possivelmente tem outras cópias) e *dirty* (modificado e é a única cópia em todas as caches). Também é usada a abordagem de *write-back*, mas somente quando o bloco está no estado *dirty*.

Em uma falha de leitura, a cache com maior prioridade que tenha uma cópia vai colocar o bloco no barramento e gravar o bloco na memória principal se o estado for *dirty*. Todas as caches que possuam uma cópia do bloco mudam seu estado para compartilhado. Se o bloco vir da memória principal, então nenhuma outra cache tem o bloco e o estado é definido como válido-exclusivo.

Em um acerto de escrita, um bloco *dirty* é escrito sem transições e um bloco válido-exclusivo é mudado para *dirty*.
Se o bloco é compartilhado, então é necessário invalidar todas as cópias existentes, e o bloco gravado também é definido como *dirty*.

Em uma falha de escrita, todas as caches com cópias são invalidadas e o bloco é carregado como *dirty*.

2.1.4 *Firefly*

É o esquema (figura 4) usado no *workstation* Firefly [11]. Neste caso são usados três estados: válido-exclusivo (não modificado, é a única cópia nas caches), compartilhado (não modificado, possivelmente com outras cópias) e *dirty* (modificado, é a única cópia nas caches). Também é usado *write-back* para blocos *dirty* a serem substituídos em caso de conflito. A principal diferença dessa técnica é que ela não gera invalidações (não apresenta o estado inválido), permitindo vários escritores. O dado de cada escrita para um bloco compartilhado é transmitido para cada cache que possua uma cópia.

Em uma falha de leitura, a cache que possui o bloco responde fornecendo o bloco com o dado correto. Neste caso o estado do bloco de todas as caches que possuam uma cópia são definidos como compartilhado e caso o bloco seja *dirty* ele é gravado na memória principal. Se nenhuma cache possui o bloco, este vem da memória principal e é definido para ficar no estado válido-exclusivo.

Em um acerto de escrita, nenhuma alteração de estado é realizada caso o estado seja *dirty*. Se o estado for válido-exclusivo, então é modificado para *dirty*. Quando o estado for compartilhado é que aparece a principal modificação desse protocolo: como pode haver cópias em outras caches e não há invalidações, então o dado gravado precisa ser atualizado em todas as cópias existentes. Uma linha de barramento especial é usada para determinar se existem cópias ou não para uma determinada linha da cache: se essa linha estiver com sinal alto, o compartilhamento existe e então o estado é mantido em compartilhado. Caso contrário, não há cópias e o estado pode ser definido como válido-exclusivo.

Em uma falha de escrita, se nenhuma cache possui o bloco, este é carregado da memória no estado *dirty* e depois escrito. Se alguma cache o possui, então o bloco é carregado no estado compartilhado e a palavra é escrita na memória, sendo que os blocos que possuem uma cópia são atualizados.

2.1.5 *MESI*

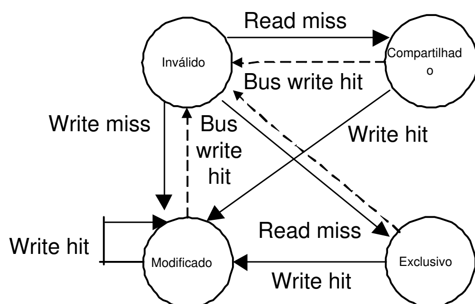

Figure 5: Protocolo MESI.

No protocolo MESI [10] (figura 5), utilizado no Pentium II, um bloco pode ter um de quatro estados: inválido, exclusivo (a linha de cache é igual a da memória principal e é a única cópia entre as caches), compartilhado (a linha da cache é igual a da memória principal e pode estar compartilhado) e modificado (bloco modificado, diferente da memória principal e é a única cópia entre as caches).
Em uma falha de leitura, se alguma cache tem o bloco no estado modificado, então ela fornece o bloco para a cache que o requisitou e ambos definem seu estado para compartilhado. Caso alguma cache tenha o bloco no estado compartilhado ou exclusivo, então a cache que fez o pedido do bloco faz a leitura do bloco da memória principal e todas caches com cópias mudam seu status para compartilhado. Se nenhuma cache possui o bloco requisitado, então o processador carrega o bloco da memória principal e define seu estado como exclusivo.
Em um acerto de escrita, caso o estado do bloco seja modificado, então nenhuma transição é necessária. Se o estado for exclusivo, então o estado é alterado para modificado. Caso o estado do bloco seja compartilhado, é necessário invalidar outras cópias existentes. O bloco passa de compartilhado para modificado.
Em uma falha na escrita, o bloco é carregado da memória principal, caso não existam cópias, é marcado como modificado e logo após é realizada a escrita. Se alguma cache possui uma cópia do bloco, ela escreve no barramento o bloco e marca seu estado para inválido. O processador da cache que fez a requisição lê esta linha, realiza sua escrita e marca como modificada. Todas as outras cópias existentes são invalidadas.

2.2 Protocolos de Diretório

Os protocolos de monitoração são limitados a sistemas multiprocessados com tipicamente menos que 32 processadores, porque o barramento se torna o gargalo para um grande número de processadores. Protocolos de coerência baseados em diretório eliminam a necessidade de um barramento compartilhado, pois mantêm informações de compartilhamento dos dados armazenadas nas caches em diretórios. Isso permite que os sistemas sejam mais escaláveis, com um número maior de processadores, em que informações sobre coerência podem ser enviadas entre processadores usando conexão ponto-a-ponto, sem a necessidade de *broadcast* para todos os processadores em um barramento compartilhado, que é o que acontece com os protocolos de monitoração [6].
As implementações de diretório associam uma entrada de diretório a cada bloco de memória, mas não impede que a propósito de otimizações seja relacionada uma entrada a cada bloco da cache. Além disso, cada entrada possui informação adicional sobre quais caches (processadores) possuem cópia do bloco, para uma posterior busca e/ou invalidação. O diretório pode ser centralizado, juntamente com uma memória compartilhada centralizada, ou também pode ser distribuído, assim como a memória. Neste caso, o diretório não se torna um gargalo, permitindo que acessos a diferentes entradas de diretórios possam ser realizados de forma distribuída [4].
Os esquemas baseados em diretório se apresentam basicamente em três categorias possíveis: diretórios com mapeamento completo, diretórios limitados e diretórios encadeados. Em diretórios com mapeamento completo, cada entrada de diretório tem um bit por processador mais um bit de *dirty*. Em diretórios limitados, existem um número fixo de bits para indicar cópias em processadores, restringindo o número de cópias simultâneas de qualquer bloco. Em diretórios encadeados, um mapeamento completo é emulado distribuindo o diretório entre as caches [1].
Os esquemas baseados em diretório tipicamente funcionam da seguinte maneira: os diretórios mantêm informação sobre quais processadores possuem em suas caches determinados blocos. Um processador deve ter acesso exclusivo ao bloco do diretório antes de ser permitido a ele realizar uma escrita no bloco. Assim, o diretório envia uma mensagem para todos os processadores que possuem cópias do bloco para invalidá-los e espera as confirmações das requisições de invalidação, para logo após ter acesso exclusivo ao bloco. Quando um processador tenta ler um bloco que é de exclusividade de outro processador, uma falha de leitura da cache ocorre e o diretório irá enviar ao processador proprietário uma mensagem para ele realizar um *write back* na memória. O bloco pode então ser compartilhado para leitura pelo processador original e pelo o que requisitou o bloco [6].
Em [4] um exemplo simples de protocolo de diretório é apresentado, que implementa as duas operações citadas anteriormente: falha de leitura da cache e gravação de um bloco da cache. Para implementar isso o protocolo mantém três estados para cada bloco da cache:

- Compartilhado - o bloco na memória está atualizado e um ou mais processadores possuem em sua cache tal bloco;
- Não inserido na cache - nenhum processador tem uma cópia do bloco da cache;
- Exclusivo - o bloco na memória está desatualizado e somente um processador tem a cópia do bloco na cache (proprietário).

Neste protocolo o diretório é armazenado de forma distribuída. Assim surgem três conceitos: nó local, de onde uma requisição foi realizada, nó inicial, onde o diretório está localizado, e nó remoto, que possui uma cópia do bloco da cache, seja ela exclusiva ou compartilhada. As possíveis mensagens entre nós podem ser vistas na tabela 1 [4].

Quando um bloco está no estado não inserido na cache e ocorrer uma falha de leitura, o estado do bloco se torna compartilhado e a cache local recebe os dados da memória.

Table 1: Mensagens enviadas entre nós para manter coerência em um protocolo baseado em diretórios.

Tipo de Mensagem	Origem	Destino	Conteúdo da Mensagem	Função dessa Mensagem
Falha de Leitura	Cache Local	Diretório Inicial	P,A	O processador P tem uma falha de leitura no endereço A; solicitar dados e fazer de P um compartilhador de leitura.
Falha de Gravação	Cache Local	Diretório Inicial	P,A	O processador P tem uma erro de gravação no endereço A; solicitar dados e fazer de P o proprietário exclusivo.
Invalidar	Diretório Inicial	Cache Remota	A	Invalidar uma cópia compartilhada de dados no endereço A.
Buscar	Diretório Inicial	Cache Remota	A	Buscar o bloco no endereço A e enviá-lo a seu diretório inicial; mudar o estado de A na cache remota para compartilhado.
Buscar/invalidar	Diretório Inicial	Cache Remota	A	Buscar o bloco no endereço A e enviá-lo a seu diretório inicial; invalidar o bloco na cache.
Resposta de valor de dados	Diretório Inicial	Cache Local	D	Retornar um valor de dados da memória inicial.
Write back de dados	Cache Remota	Diretório Inicial	A,D	Executar o *write back* de um valor de dados no endereço A.

Caso ocorra uma falha de escrita, o processador solicitante se torna proprietário, definindo o estado do bloco como exclusivo. O conjunto de compartilhadores do bloco reflete esse estado, mantendo indicação de cópia de bloco somente para esse processador.
Quando o bloco está no estado compartilhado e ocorrer uma falha de leitura, o processador solicitante recebe os dados da memória e é adicionado ao conjunto de compartilhadores do bloco. Se houver uma falha de escrita, todos os processadores recebem mensagens para invalidarem seus blocos de cache e o estado do bloco se torna exclusivo, indicando no conjunto de compartilhadores que o único processador a possuir uma cópia foi o que realizou a solicitação.
Quando o bloco está no estado exclusivo e ocorrer uma falha de leitura, o processador proprietário recebe uma mensagem de busca de dados, mudando o estado do bloco para compartilhado e fornecendo ao solicitante o bloco desejado. É adicionado ao conjunto de compartilhadores o processador solicitante. Ao substituir um bloco, o processador proprietário deverá realizar *write back* de dados, atualizando a cópia em memória e removendo o processador do conjunto de compartilhadores de tal bloco. Em falhas de escrita, uma mensagem é enviada ao antigo proprietário invalidando seu bloco na cache. Logo depois, o conjunto de compartilhadores mantém somente o processador solicitante, mantendo o estado do bloco como exclusivo para ele.

3. MODELOS DE CONSISTÊNCIA DE MEMÓRIA

O modelo de consistência de memória especifica a ordem na qual operações em memória serão vistas pelo programador. Uma leitura em uma posição de memória deveria retornar o valor da última escrita na mesma posição de memória. Em sistemas multiprocessados, onde há vários processadores lendo e escrevendo em posições de memória, essa noção de "último" não é bem definida.
Alguns métodos de consistência de memória foram propostos na literatura, dentre eles a consistência sequencial e modelos de consistência relaxados, que podem ser classificados de acordo com as ordenações que relaxam em consistência do processador, consistência fraca ou consistência de liberação [4].
A consistência sequencial fornece a noção de que todas as operações de memória parecem ser executadas uma a cada instante e que todas as operações de um único processador são executadas na ordem descrita pelo programa que está sendo executado no processador. Segundo [2], duas condições são suficientes para manter a consistência sequencial:

1. Antes de uma instrução de carga ser permitida executar em relação a qualquer outro processador, todas as instruções anteriores de carga e armazenamento devem ser executadas, e

2. Antes de uma instrução de armazenamento ser permitida executar em relação a qualquer outro processador, todas as instruções anteriores de carga e armazenamento devem ser executadas.

Os modelos de consistência relaxados levam em consideração justamente essa característica de sistemas multiprocessados: leituras e escritas fora de ordem. A fim de manter a consistência sequencial, instruções de sincronização são usadas para impor a ordenação. Há três casos de relaxamento de ordenação: escrita para leitura, escrita para escrita e todas as ordens.

No caso da escrita para leitura, é permitida uma leitura ser reordenada em relação a escritas anteriores do mesmo processador. Isso pode violar a consistência sequencial dos programas da figura 6. Assim, as duas condições para consistência sequencial são relaxadas para [2]:

1. Antes de uma instrução de carga ser permitida executar em relação a qualquer outro processador, todas as instruções anteriores de carga devem ser executadas, e

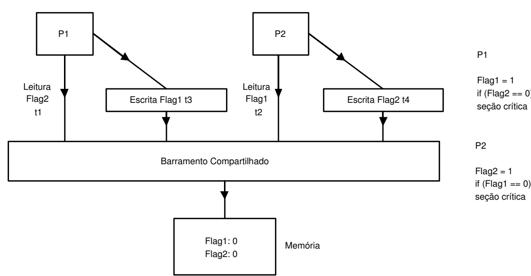

Figure 6: Relaxamento de escrita para leitura.

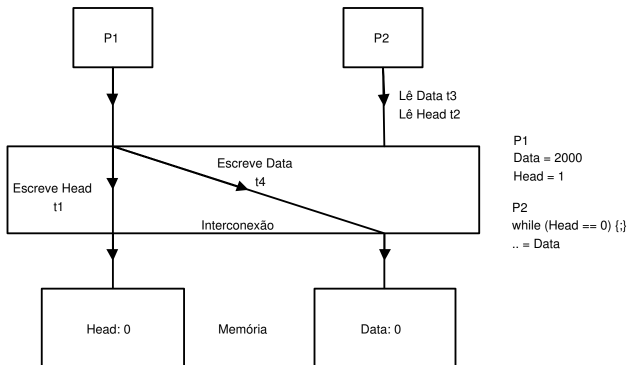

Figure 7: Relaxamento de escrita para leitura e de escrita para escrita.

2. Antes de uma instrução de armazenamento ser permitida executar em relação a qualquer outro processador, todas as instruções anteriores de carga e armazenamento devem ser executadas.

No caso de escrita para leitura e de escrita para escrita, é permitido que escritas para diferentes posições em um mesmo processador sejam colocadas em *pipeline* ou sobrepostas, podendo acessar a memória ou cache fora da ordem de execução do programa. Isso pode violar a consistência sequencial dos programas da figura 7, onde P1 e P2 são processadores que possuem programas diferentes que compartilham dados. A sequência cronológica das operações é indicada por t1, t2, t3 e t4.

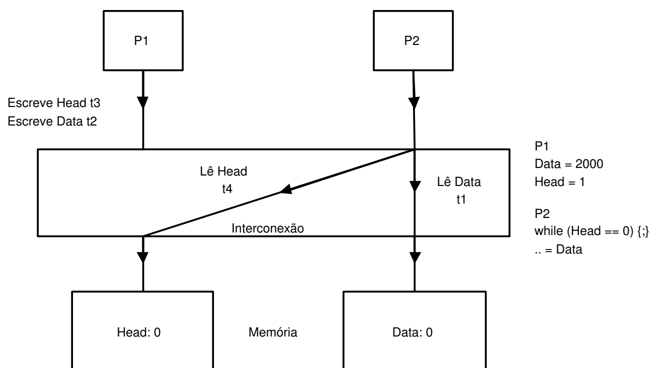

Figure 8: Relaxamento para todas as ordens.

No caso de todas as ordens de instruções serem relaxadas, é permitido que leituras e escritas sejam reordenadas em relação às leituras e escritas subsequentes. Isso pode violar a consistência sequencial dos programas da figura 8. Um dos modelos que é derivado deste caso é o chamado consistência fraca, que se utiliza de pontos de sincronização para manter a consistência das leituras e escritas. Neste caso três condições devem ser satisfeitas, segundo [2]:

1. Antes de uma instrução de carga ou armazenamento ser permitida executar em relação a qualquer outro processador, todos pontos de sincronização anteriores devem ser executados, e

2. Antes de um ponto de sincronização ser permitido executar em relação a qualquer outro processador, todas instruções de carga e armazenamento devem ser executadas, e

3. Pontos de sincronização são sequencialmente consistentes em relação a outro ponto de sincronização.

Assim, enquanto que o mecanismo de coerência de cache garante que as escritas em todas as caches para um bloco específico terão a mesma ordem lógica, a consistência de memória define ao programador como a ordem das escritas para diferentes blocos será percebida por cada processador.

4. CONCLUSÃO

O presente trabalho apresentou uma revisão bibliográfica sobre dois problemas existentes nos sistemas multiprocessados modernos: coerência de memórias cache e consistência de memória. Dentre o problema de coerência de cache foram discutidos mecanismos baseados em software, que a partir de análise de código pelo compilador gera código coerente, e mecanismos baseados em hardware, que são os mais comuns em arquiteturas atuais. Basicamente mecanismos em hardware se dividem em protocolos de monitoração e baseados em diretório. Para sistemas multiprocessados com barramento compartilhado e com um número limitado de processadores, as técnicas baseadas em monitoração são as preferidas, variando desde protocolos simples até protocolos que implementam máquinas de estados mais complexas a fim de melhorar o desempenho das caches. Basicamente tais protocolos baseados em estados mantém um ou dois estados que indicam que o bloco é exclusivo para uma determinada cache e quaisquer gravações invalidam todas as cópias existentes nas caches, marcando tal bloco como exclusivo ou *dirty*. Outro estado, em geral, indica se o bloco é possivelmente compartilhado ou não. Para sistemas com um número maior de processadores, técnicas baseadas em diretório são as preferidas. Um exemplo simples foi apresentado, implementando um protocolo de coerência de caches baseado em diretórios distribuídos e uma máquina com três estados possíveis para cada bloco de cache.
Por último foi realizada uma breve discussão à respeito de modelos de consistência de memória, que definem a ordem na qual as instruções serão vistas pelo programador, ou seja, quando um valor gravado será retornado por uma leitura. A coerência de memórias cache complementa esse conceito, definindo que valores podem ser retornados por uma leitura.

Mecanismos de sincronização são fundamentais para manter a consistência de memória em arquiteturas modernas, pois permitem que a consistência sequencial seja garantida somente em certos pontos e não em todas as instruções, degradando a performance do sistema.

5. REFERENCES

[1] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-based cache coherence in large-scale multiprocessors. *Computer*, 23(6):49–58, 1990.

[2] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 15–26, New York, NY, USA, 1990. ACM Press.

[3] J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture*, pages 124–131, Los Alamitos, CA, USA, 1983. IEEE Computer Society Press.

[4] J. L. Hennessy and D. A. Patterson. *Arquitetura de Computadores uma Abordagem Quantitativa*. Editora Campus, 2003.

[5] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. In *ISCA '85: Proceedings of the 12th annual international symposium on Computer architecture*, pages 276–283, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[6] R. Lawrence. A survey of cache coherence mechanisms in shared memory multiprocessors. 1998.

[7] D. J. Lilja. Cache coherence in large-scale shared-memory multiprocessors: issues and comparisons. *ACM Comput. Surv.*, 25(3):303–338, 1993.

[8] M. M. K. Martin. Formal verification and its impact on the snooping versus directory protocol debate. *Proceedings of the 2005 International Conference on Computer Design*, pages 543–549, 2005.

[9] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA '84: Proceedings of the 11th annual international symposium on Computer architecture*, pages 348–354, New York, NY, USA, 1984. ACM Press.

[10] W. Stallings. *Arquitetura e Organização de Computadores*. Prentice Hall, 2003.

[11] C. Thacker. Private communication. 1984.