Coerência de Cache: Fundamentos, Protocolos e Técnicas Avançadas

Tiago Cariolano de Souza Xavier [1]
Anderson Faustino da Silva [1]

Resumo: Consistência e coerência são dois requisitos fundamentais em sistemas de memória compartilhada. Porém, fornecê-los por meio de um sistema eficiente, confiável e transparente é um grande desafio em arquiteturas multiprocessadores. A principal dificuldade envolvida é solucionar o problema de coerência de cache. Devido a importânicia deste tema nos sistemas multiprocessadores, este tutorial revisita os principais conceitos relacionados a coerência de cache, descreve os principais protocolos de coerência e analisa as duas abordagens para manutenção de coerência: *snooping* e diretórios. Além disso, apresenta técnicas avançadas de coerência de cache atuais.

1 Introdução

Sistemas multiprocessadores de memória compartilhada têm se tornado cada vez mais polulares nos dias de hoje. Isso se deve a capacidade que este tipo de arquitetura possui de interconectar vários processadores de custo relativamente baixo e ainda assim, fornecer alto poder computacional [15].

Em tais arquiteturas, uma das principais questões de projeto é definir como será a organização da hieraquia de memória, haja vista que, em contraste às arquiteturas uniprocessadores, em que todo o sistema de memória é de uso exclusivo de um único processador, em sistemas multiprocessadores cada um dos núcleos de processamento pode compartilhar unidades da hierarquia de memória, mas precisamente unidades de *cache* blocos de *cache* entre si. Este compartilhamento da memória faz surgir um novo desafio arquitetural: o problema da coerência de *cache* [18].

Cada processador (ou núcleo de uma arquitetura *multicore* – daqui para frente tratados apenas como núcleo) de um sistema de memória compartilhada possui sua própria *cache* privada, porém o espaço de endereçamento é único e cada linha de *cache* pode estar compartilhada entre os diversos núcleos. Se um núcleo realiza uma operação de escrita em um bloco de *cache* compartilhado, caso nenhuma precaução seja tomada, outros núcleos e a memória

[1] Departamento de Informática
Universidade Estadual de Maringá
Avenida Colombo, 5790 - Bloco C56 - Maringá/PR - CEP 87020-900
{tiago.cariolano@gmail.com, anderson@din.uem.br}

principal, que também possuam este bloco, passarão a ter um dado inválido. Essa é a situação que caracteriza o conhecido problema de coerência de *cache* [18].

Para solucionar este problema protocolos de coerência de *cache* são incorporados na arquitetura a fim de garantir que a parte da memória compartilhada por todos os núcleos esteja coerente. Duas abordagens para definição de protocolos são frequentemente utilizadas para manter a coerência de *cache*: protocolos baseados em *snooping* e protocolos baseados em diretórios [12].

Arquiteturas que fazem coerência com protocolos baseados em *snooping* possuem um meio de comunicação *broadcast* entre os núcleos, geralmente o próprio barramento do sistema , que interconecta todos os núcleos. Este barramento permite que todos os controladores de *caches* privadas observem todas as transações emitidas por cada núcleo e realizem a ação apropriada para cada requisição.

Protocolos baseados em diretórios, por outro lado, possuem uma estrutura denominada diretório que armazena informações sobre as cópias de *cache* de cada bloco de memória compartilhada. Quando um controlador de *cache* privada recebe uma requisição de seu núcleo, ele consulta o diretório para descobrir se o bloco é compartilhado e por quais núcleos. Quando ocorre uma escrita de um bloco, primeiro ocorre uma consulta ao diretório para descobrir se o bloco é compartilhado. Caso seja, seu estado é alterado de maneira que o núcleo tenha acesso exclusivo a este bloco para fazer modificações e algum mecanismo deve existir para que os outros núcleos possam obter a cópia correta do bloco compartilhado. Protocolos baseados em diretórios podem ser implementados de maneira centralizada ou distribuída, neste último caso cada núcleo possui um diretório que mantém as informações dos blocos compartilhados por ele.

Protocolos baseados em *snooping* geralmente sofrem problemas de latência de acesso a memória devido ao uso compartilhado de um barramento. Como este recurso é disputado por todos os núcleos ele se torna um ponto de gargalo do sistema caso ocorram muitas requisições de coerência [7, 8]. Além disso, o tráfego de coerência requer uma maior largura de banda, que com o uso de um barramento centralizado limita a escalabilidade do sistema.

Utilizar uma estrutura como um diretório centralizado também pode ser um ponto de gargalo no sistema [2] [5]. Primeiro, como essa estrutura deve armazenar as informações de todos os blocos de *cache* compartilhados, seu tamanho aumenta rapidamente em função da quantidade de blocos de *cache* e da quantidade de núcleos. Segundo, se a aplicação possuir muitos dados compartilhados diversas indireções ocorrerão para o diretório, o que causará um aumento da latência de acesso à hierarquia de memória.

Como abordagens baseadas em *snooping* e abordagens baseadas em diretórios conseguem manter a coerência ao custo de uma perda de desempenho, novas técnicas foram propostas a fim de reduzir o impacto que a manutenção da coerência impõe ao desempenho

do sistema.

Este tutorial apresenta o tema coerência de *cache* fornecendo uma descrição detalhada da abordagem baseada em *snooping*, como também da baseada em diretórios. Além disso, a fim de fornecer uma fundamentação deste tema, são apresentados os principais protocolos de coerência. Por fim, são apresentadas técnicas avançadas de coerência atuais. As principais contruições deste tutorial são descritas a seguir:

- Apresenta uma fundamentação de conceitos relacionados à coerência de *cache*.
- Descreve três protocolos de coerência de *cache*, a saber: `MSI`, `MESI` e `MOESI`.
- Descreve as duas principais abordagens para implementações de protocolos de coerência: *snooping* e diretórios.
- Apresenta técnicas avançadas para solucionar o problema de coerência de *cache* visando aumentar o desempenho global dos sistemas atuais.

O restante deste tutorial está organizado como segue. A Seção 2 descreve os conceitos fundamentais sobre coerência de *cache*. A Seção 3 apresenta os protocolos baseados em *snooping* e diretórios. A Seção 4 apresenta algumas técnicas presentes na literatura cujo objetivo é reduzir o impacto que a manutenção da coerência impõe ao desempenho do sistema. E por fim, A Seção **??** finaliza este tutorial com as considerações finais.

2 Fundamentos

Um sistema de memória compartilhada (SMP – *symmetric shared memory multiprocessor*) é constituído por dois ou mais nós de processamento que possuem capacidade computacional comparável. Esses nós, também chamados processadores ou núcleos, são conectados entre si por um meio de comunicação, usualmente o barramento do sistema, que também conecta eles a memória principal e a subsistemas de entrada e saída [19].

Em SMPs o uso de memória compartilhada acarreta que todos os núcleos tenham acesso a qualquer endereço de memória. Consequentemente, dois ou mais núcleos podem acessar alguma linha de suas respectivas *caches* privadas que corresponde ao mesmo bloco de memória.

Um sistema SMP típico possui uma hierarquia de memória constituída de dois ou mais níveis de *cache* e a memória principal [4] [19, 16]. O primeiro nível (*cache* L1) geralmente é privado, porém, devido ao sistema de memória compartilhada, cada uma de suas linhas de *cache* podem ser compartilhadas entre as *caches* L1 de qualquer outro núcleo. Os demais

níveis frequentemente são públicos, apesar de existirem implementações em que cada núcleo possui sua própria *cache* L2 privada com um terceiro nível compartilhado.

Um núcleo pode realizar operações de leitura e/ou escrita em sua *cache* L1, porém algumas vezes o dado são estará neste nível de *cache* sendo necessário neste caso ser buscado da memória (ou do nível acima) para a *cache* privada do processador. Durante uma escrita a uma determinada linha de *cache*, o correspondente bloco de memória deve ser atualizado. Este processo pode ser realizado por meio de duas políticas: *write-through* e *write-back*. Na primeira, sempre que ocorre uma modificação da linha de *cache* o bloco correspondente é atualizado na memória. Na segunda a atualização é feita apenas na substituição da linha de *cache*.

É fácil verificar que na política *write-back*, quando um mesmo dado está compartilhado entre dois núcleos, se um deles faz pelo menos uma escrita em sua *cache* privada, a linha de *cache* do outro núcleo (com o dado compartilhado) conterá um bloco de memória desatualizado. O mesmo problema também ocorre na política *write-through*, embora de maneira mais sútil: se um núcleo tem uma falta de um dado que é compartilhado por diversos núcleos, obviamente o dado será buscado na memória principal, entretanto durante o período que o processador aguarda a recuperação do dado na memória, este dado pode ser modificado por algum dos outros núcleos e assim a informação recuperada também estará desatualizada. Essas duas situações delineiam o que é o problema de coerência de *cache* [20].

A Figura 1 apresenta uma ilustração deste problema de maneira mais detalhada. Nesta figura há dois núcleos (N_0 e N_1) que possuem intenção de compartilhar o mesmo endereço de memória X, que possui inicialmente o valor 0. Cada subfigura representa o estado do sistema após um ciclo de *clock*. Após o primeiro *clock* N_0 realiza uma leitura de X, o que ocasiona a carga de X para sua *cache* privada, em seguida N_1 realiza a mesma operação. Porém, no terceiro *clock* N_0 realiza uma escrita em X em sua *cache* privada, no entanto a *cache* de N_1 não foi atualizada e o endereço X na *cache* privada de N_1 ainda possui o valor 0. Em seguida, quando N_0 realiza a leitura de X o valor obtido é 0 e não 1 como esperado.

Protocolos de coerência de *cache* são utilizados para impedir que esse problema ocorra. Um protocolo de coerência mantem uma máquina de estados finitos para cada linha de *cache* e fornece requisições de coerência que representam eventos que alteram esses estados.

O protocolo pode ser implementado em um controlador de coerência distribuído através das diversas *cache*s e da memória. A principal tarefa desse controlador é examinar as requisições de coerência que chegam e, baseado nelas, alterar o estado da linha de *cache* e/ou emitir outras requisições para os controladores das *cache*s dos outros processadores ou da memória.

A máquina de estado para cada linha de *cache* é representada por meio de alguns *bits* adicionais na linha (*snooping*) ou em algum local centralizado que armazena a informação de estado para as linhas de todas as *cache*s (diretório). Um estado é atualizado com base

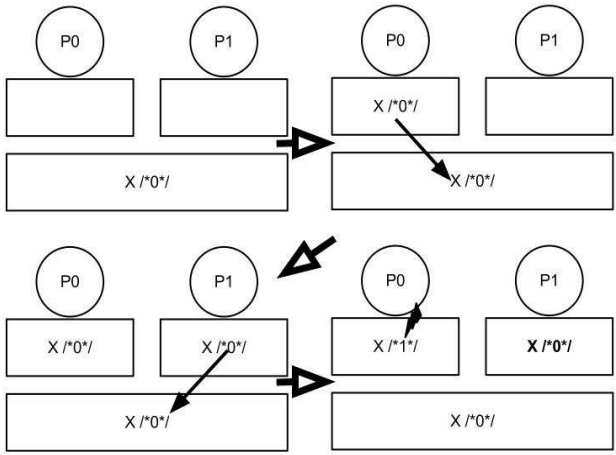

Figura 1. Problema de coerência de *cache*

no estado e na requisição de coerência atual. As requisições de coerência podem chegar ao controlador por duas fontes, de acordo com a classificação a seguir:

Requisições internas. São feitas pelo processador para o seu controlador de coerência. Podem ser *read Hit, write hit, read miss* ou *write miss*. Nas duas primeiras o processador solicita a leitura e escrita de um dado na *cache* e ele está na *cache*, nas duas últimas o processador faz a mesma solicitação, porém o dado não se encontra na *cache* (ou está, mas com estado inválido).

Requisições externas. Essas requisições chegam ao controlador vindas dos controladores de outros núcleos ou do controlador da *cache* compartilhada. Existem dois tipos básicos de requisições: *req_read* e *req_write*. Elas indicam que algum outro processador deseja ler ou escrever a linha de *cache*.

Existem diversos protocolos para controlar as máquinas de estados finitos, porém neste texto serão apresentados três. O primeiro é o protocolo `MSI`, cujo nome é um acrônimo dos possíveis estados de uma linha de cache, a saber: *modify*, *shared* e *invalid*. Os outros dois são extensões do protocolo `MSI` com melhorias que permitem um aumento de desempenho. Um deles é o protocolo `MESI` que possui os estados do `MSI` e um quarto estado chamado *exclusive*. O outro é o o protocolo `MOESI` que possui os quatro estados do `MESI` mais o estado *owned*.

Esses três protocolos são desenvolvidos para políticas de atualização de *cache write-back*. Caso a política seja *write-through* outros protocolos devem ser utilizados. A política de atualização *write-back* é a mais utilizada nas arquiteturas atuais, pois frequentemente apresenta melhor desempenho com relação à *write-through*, já que não atualiza a memória a cada operação de escrita. Por esse motivo, este artigo considerará a partir deste ponto que a política de atualização da *cache* é sempre *write-back*, exceto os casos que for expresso o contrário. Para uma avaliação de alguns protocolos que utilizam a estratégia *write-update* uma boa referência é o trabalho de Hashemi [8].

O problema da coerência de *cache* pode ser solucionado inteiramente via software, via hardware ou por meio de uma abordagem híbrida. Este tutorial aborda apenas sistemas que mantém a coerência por hardware e sistemas híbridos. Para um estudo sobre sistemas de coerência unicamente por software uma boa referência é o trabalho de Tartalja e Milutinoviác [21].

2.1 Protocolo MSI

O `MSI` é um modelo básico de protocolos de coerência de *cache* [18]. Apesar dele não ser utilizado na prática, devido a problemas de desempenho, ele serve como referência para os dois outros protocolos que serão apresentados na Seção 2.2 e Seção 2.3. Este protocolo possui os seguintes estados:

Modified. A linha de *cache* foi modificada, possui valor diferente daquele na memória e nenhum outro processador possui uma cópia dela.

Shared. O núcleo pode ler essa linha de *cache*, mas não escrever, pois uma cópia pode existir em outras *caches*. Além disso, o valor desta linha é igual ao que está na memória principal.

Invalid. A linha de *cache* contém um dado inválido.

A Figura 2 apresenta a máquina de estados finitos do protocolo `MSI`. Nesta figura, as transições com linhas cheias correspondem às requisições de coerência internas e as com linhas tracejadas correspondem às requisições de coerência externas ou ações realizadas pelo controlador de coerência (*flush*).

A requisições internas são:

Read hit: a linha de *cache* só pode estar ou no estado compartilhado ou no modificado (se estivesse no estado inválido haveria um *read miss*) nesta transição. Como o dado já está presente na *cache*, o processador apenas realiza a leitura e mantem o estado da linha de *cache* inalterado.

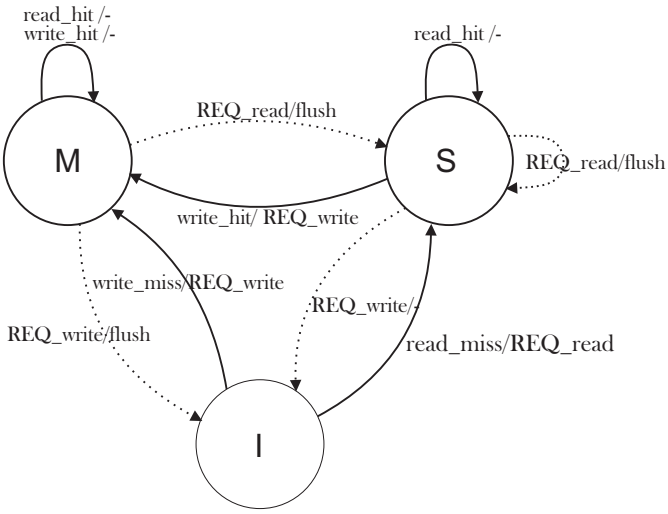

Figura 2. Máquina de estados finitos para o Protocolo `MSI`

Write hit: durante esta transição a linha de *cache* também estará nos estados compartilhado ou modificado (caso contrário seria um *write miss*). Se a linha de *cache* estiver no estado compartilhado o controlador envia uma requisição *req_write* para as demais *caches* e altera seu estado para *modified*; essa requisição tem por objetivo fazer com que os outros núcleos que possuem uma cópia da linha alterem seus estados de *shared* para *invalid*; em seguida o núcleo realiza a escrita. Se a linha de *cache* estiver no estado *modified*, o processador já possui a exclusividade desta linha e como nenhum outro núcleo tem uma cópia dela, ele pode modificá-la.

Read miss: esta transição só ocorre com a linha de *cache* no estado inválido, ou quando a linha não está na *cache* (em outro estado ocorreria um *read hit*). Ao ocorrer um *read miss* o controlador de coerência envia uma requisição *req_read* para as *caches* dos demais núcleos; essa requisição tem por objetivo fazer com que os núcleos que possuam uma cópia válida dessa linha (em estado *modified* ou *shared*) mudem seus estados para *shared* e forneçam, por algum meio de comunicação, a linha de *cache* para o núcleo requisitante. Após realizar a requisição *req_read* o controlador de memória realiza uma leitura do dado por meio de uma comunicação direta com algum outro núcleo ou com a memória e altera o estado da linha para *shared*.

Write miss: esta transição só pode ocorrer com a linha de *cache* no estado *invalid* (caso contrário seria um *write hit*). Ao ocorrer essa transição o controlador de coerência

emite uma requisição *req_write* para os demais controladores; essa requisição tem o objetivo de fazer os outros núcleos mudarem seus estados para inválido. Em seguida, o núcleo realiza a escrita e altera seu estado para modificado.

As requisições externas são:

Req_read: Ao receber essa requisição o controlador de coerência é informado que algum outro núcleo deseja realizar uma leitura na linha de *cache* correspondente. Se o estado da linha é *shared* o controlador fornece o dado para o núcleo requisitante por meio da operação *flush* e mantem o estado da linha inalterado. Se o estado é modificado é possível que o valor da linha seja diferente daquele na memória, consequentemente o controlador deve atualizar a memória para que o processador requisitante realize uma leitura na memória (*flush*); Além disso, ele altera o estado da linha para *shared*.

Req_write: Esta requisição indica que outro núcleo deseja escrever em uma linha de *cache* que está na *cache* do controlador. Se o estado for compartilhado, a cópia da linha já está atualizada com a memória e basta o seu estado ser alterado para *invalid*. Se o estado for *modified*, a cópia da linha está potencialmente suja, consequentemente a memória deve ser atualizada (*flush*) antes de o estado ser alterado para inválido.

Flush: uma operação de atualização de memória e/ou uma transferência entre *caches*.

Nas descrições anteriores foram omitidos alguns detalhes, por exemplo, como o controlador emite uma *req_read* ou *req_write* para os demais núcleos e como é realizada a resposta de um núcleo para um processador requisitante. A omissão dessas informações, por enquanto, não é prejudicial, pois o objetivo até o presente momento é apresentar a essência do funcionamento do protocolo. Nas Seções 3.2 e 3.3, onde serão descritos protocolos baseados em *snooping* e diretórios, essas questões serão retomadas e explicadas, pois elas são altamente dependentes do tipo de protocolo utilizado.

O protocolo `MSI` garante a coerência do sistema de memória, porém ele pode apresentar dois problemas de desempenho [18], a saber:

Problema 1 . Ocorre quando um núcleo realiza uma leitura (*read miss*) seguida de uma escrita (*write hit*). Suponha a situação: um núcleo N realiza um *read miss*, envia uma requisição *req_read* e em seguida altera seu estado de *invalid* para *shared*; em seguida N realiza um *write hit* para o mesmo endereço de memória, neste caso mais uma requisição será realizada (*req_write*) para que as cópias em outros núcleos sejam invalidadas. Esta sequência de eventos, bastante comum em muitas aplicações (ler uma variável para depois escrevê-la) pode ocasionar uma degradação de desempenho, pois sempre ocorrerá duas requisições.

Problema 2 . Ocorre quando uma *cache* tem uma linha no estado modificado e recebe uma requisição *req_write* de outro núcleo. Nesta situação o núcleo envia o dado tanto para o processador requisitante, quanto para a memória para que ela possa atender novas requisições com o dado atualizado. Como o dado já foi enviado por uma comunicação *cache* a *cache*, esse envio para a memória é desnecessário e causa uma perda de desempenho, devido a alta latência que a memória possui.

Estes problemas podem ser solucionados com melhorias no protocolo `MSI`, as quais são descritas nas próximas duas seções.

2.2 Protocolo MESI

O protocolo `MESI` resolve o Problema 1 do protocolo `MSI` com o acréscimo de um estado *exclusive* na máquina de estados finitos. O estado *exclusive* indica que a linha de *cache* é válida, igual àquela na memória e não está em nenhuma outra *cache*.

A Figura 3 apresenta o diagrama de estados para o protocolo `MESI`. As requisições internas deste protocolo são semelhantes às existentes no `MSI`.

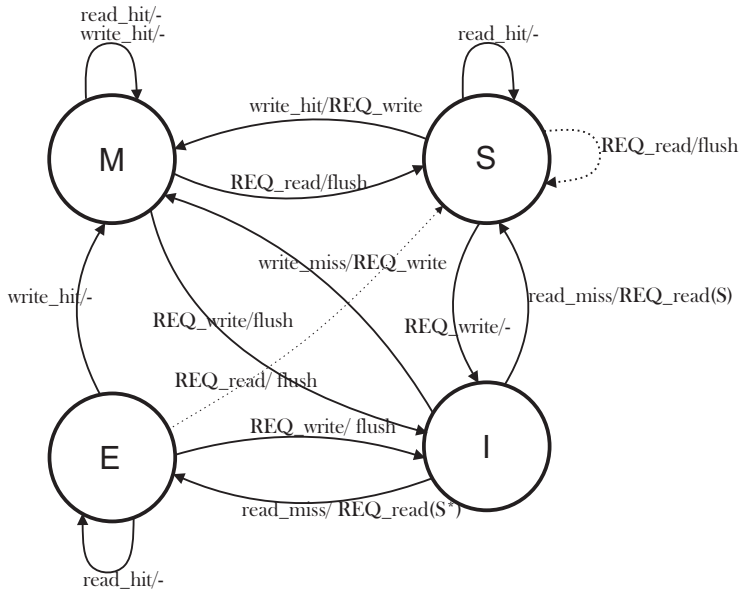

Figura 3. Máquina de estados finitos para o protocolo `MESI`

As mudanças com relação ao protocolo `MSI` são:

Read hit: Se o bloco está no estado *exclusive* o controlador de coerência realiza uma leitura e o estado permanece o mesmo.

Write hit: Se a linha de *cache* está no estado *exclusive*, então apenas o núcleo requisitante possui este dado que é o mesmo que está na memória, consequentemente este núcleo necessita apenas alterar o seu estado para modificado e realizar a escrita.

Read miss: O controlador de coerência envia uma requisição para os demais núcleos. Essa requisição é do tipo *req_read(X)*, onde $X = S, S*$ é uma condição que indica se outras *cache*s possuem uma cópia da linha ou não. Se outro núcleo possui uma cópia válida ($X = S$), então o estado da linha é alterado para compartilhado, como no protocolo `MSI`. No entanto, se nenhum outro núcleo possui a linha de *cache* ($X = S*$), então o estado da linha é alterado de *invalid* para *exclusive*. O núcleo requisitante obtém a informação de que outros núcleos possuem uma cópia da linha de *cache* por meio de uma comunicação direta ou através da memória. A forma como essa resposta é enviada é independente do protocolo ser baseado em snooping ou diretório.

Write miss: como no protocolo `MSI`.

De forma semelhante às requisições internas, as requisições externas possuem poucas mudanças com relação ao protocolo `MSI`, a saber:

Req_read: Se estado for*exclusive* o controlador fornece o dado para o núcleo requisitante (*flush*) e altera o estado para *shared*, pois outro núcleo tem a intenção de ler o dado.

Req_write: Se o estado for *exclusive* o controlador fornece o dado para o núcleo requisitante (*flush*) e altera o estado para *invalid*, pois outro núcleo tem a intenção de escrever o dado.

Flush: o mesmo que o protocolo `MSI`.

2.3 Protocolo MOESI

O protocolo `MOESI` resolve os dois problemas descritos na Seção 2.1. O Problema 1 é resolvido como no protocolo `MESI`, com o acréscimo do estado exclusivo. E o Problema 2 é solucionado com o acréscimo de um novo estado, chamado *owned*.

O estado *exclusive* tem o mesmo significado e serve para o mesmo propósito do existente no protocolo `MESI`. O estado *owned* indica que a linha de *cache* é uma cópia válida,

porém pode apenas ser lida e o bloco correspondente a ela na memória pode ter um valor diferente. Além disso, o estado *shared* apresenta uma pequena modificação: a linha de *cache* nesse estado pode ter um valor diferente do que aquele que está na memória. A Figura 4 apresenta a máquina de estados finitos do protocolo `MOESI`.

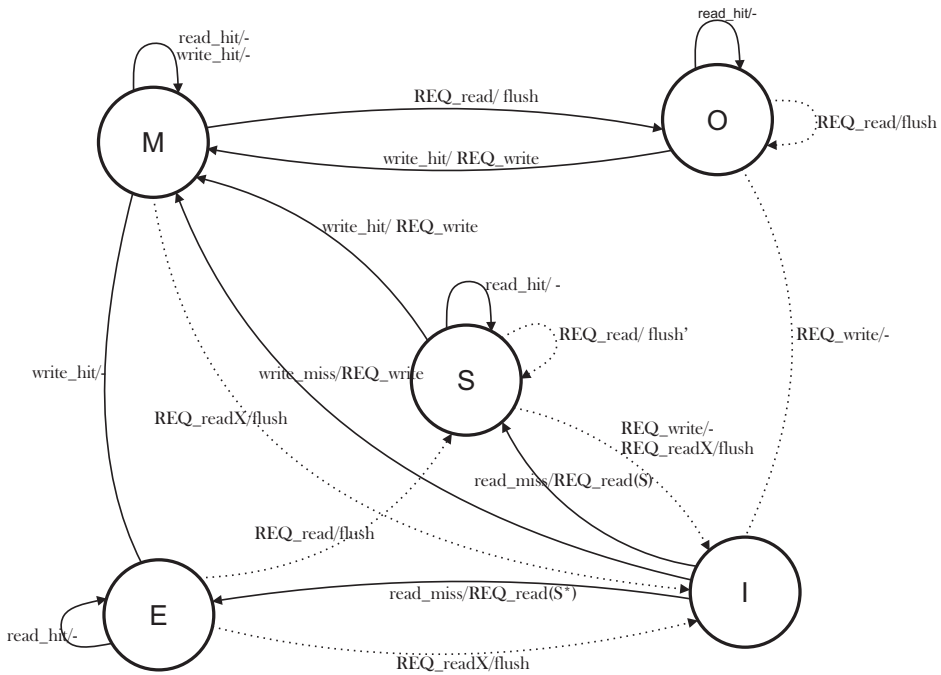

Figura 4. Máquina de estados finitos para o protocolo `MOESI`

As mudanças ocorridas em cada requisição com relação ao protocolo `MESI` é descrita como:

Read hit: Se o estado da linha de *cache* for *owned* o núcleo realiza a operação de leitura e a linha permanece no mesmo estado.

Write hit: Se o estado for *owned* controlador permite o núcleo realizar a operação de escrita, muda o estado da linha para modificado e envia uma requisição para as demais *caches*; essa requisição tem por objetivo invalidar as possíveis cópias nos outros núcleos.

Read miss: Não é possível ocorrer um *read miss* no estado *exclusive*, assim essa requisição é da mesma forma que no protocolo `MESI`.

Write miss: como no protocolo `MESI`.

As requisições externas *req_read*, *req_write* e *flush* são equivalentes às do protocolo `MESI`. Porém uma nova requisição está presente, a saber: *req_readX*.

Uma transição em *req_readX* só pode ocorrer com a linha de *cache* nos estados *modified*, *shared* ou *exclusive*. Essa requisição substitui a *req_write* nas situações em que um núcleo remoto deseja ler o dado com intenção de modificá-lo. Quando isso ocorre o núcleo local não precisa enviar o dado para a memória e a operação de *flush* é apenas uma comunicação entre *caches*. Consequentemente, se o estado da linha de *cache* local for *modified*, *shared* ou *exclusive* a operação *flush* corresponde ao envio do dado para o nó remoto, sem atualizar a memória. Isso evita que ocorra o Problema 2.

3 Protocolos

Os protocolos descritos nas Seções 2.1, 2.2 e 2.3 podem ser implementados por meio de duas abordagens: *snooping* ou diretórios [20]. Na primeira cada *cache* de um núcleo possui a informação de compartilhamento (estado) de todas as suas linhas, as mensagens entre as *caches* são transmitidas por um meio de comunicação *broadcast* centralizado (usualmente o próprio barramento do sistema) e todos os controladores de coerência das *caches* monitoram (*snooping*) o meio para realizarem as ações de coerência. Na segunda abordagem uma região de memória compartilhada chamada diretório armazena as informações de compartilhamento e a identificação do núcleo que possui determinada linha de *cache* para todas informações, as mensagens são enviadas por *unicast* e recebidas da mesma forma ou por *brodcast* pelo diretório.

Antes de examinar as duas abordagens com mais detalhes é preciso deixar claro que até agora nada foi mencionado sobre modelo de consistência de memória. Na verdade, como exposto na Seção 2, foi considerado que o resultado de qualquer execução dos núcleos fosse o mesmo como se elas fossem executadas em alguma ordem sequencial e as operações de cada processador respeitam a ordem do programa que eles executam. Essa é uma definição intuitiva para o chamado *modelo de consistência sequencial*. Na realidade, todo este tutorial considera o uso deste modelo de memória, porém outros modelos de memória podem existir.

Assim, a próxima seção apresenta uma definição de modelo de consistência de memória e em seguida, as Seções 3.2 e 3.3 apresentam as duas abordagens de protocolos de coerência.

3.1 Modelo de Consistência de Memória

Um modelo de consistência de memória, ou simplesmente modelo de consistência, especifica o comportamento do sistema de memória em termos de leituras e escritas, sem

considerar *caches* ou protocolo de coerência. Mais especificamente, ele determina *quando* um processador verá um valor que foi atualizado por outro núcleo.

Dessa forma, o problema de coerência de *cache* pode ser resolvido implementando um modelo de consistência, não necessariamente por meio de protocolos de coerência. Entretanto, na prática, a maior parte dos sistemas de memória compartilhada implementam o modelo de consistência por meio de coerência de *cache*.

O modelo mais restrito de consistência utilizado em SMPs é o sequencial. Neste modelo, a ordem das operações na memória respeita a ordem do programa em cada núcleo, de modo que, do ponto de vista do núcleo, cada operação se inicia apenas ao término de alguma anterior. Se um núcleo tem duas operações de memória op_1 e op_2 que executam nesta ordem, independente das operações dos outros núcleos, as requisições de coerência para op_1 sempre serão executadas *antes* das operações de op_2. Este modelo é o mais intuitivo, pois funciona da mesma forma como a maioria dos programadores enxergam a memória.

Modelos de consistência que não respeitam esta ordem são denominados modelos *relaxados*. Nesses modelos escritas e leituras podem ser completadas fora de ordem na memória, porém operações de sincronização são utilizadas para forçar ordenação. Existem diversos modelos de acordo com a sequência de operações que eles relaxam, a saber:

Relaxando $store \rightarrow load$ Esses modelos permitem que uma operação de leitura seja reordenada com relação a uma operação de escrita anterior do mesmo processador. Como resultado desta reordenação, em algum ponto da execução, uma leitura pode retornar o valor de uma escrita. Um exemplo desta relaxação é o modelo de constência de processador.

Relaxando $store \rightarrow store$ Esses modelos eliminam regras de ordenação entre escritas. Dessa forma, escritas do mesmo processador para diferentes locais podem ser executadas em paralelo ou sobrepostas e assim alcançar a memória ou outras cópias em *cache* fora da ordem do programa. O modelo de consistência com parcial ordenação é um exemplo desta relaxação.

Relaxando $load \rightarrow store$ **e** $load \rightarrow load$ Nestes modelos operações de memória seguidas de alguma operação de leitura podem ser sobrepostas ou reordenadas com relação a operação de escrita. Isso permite que otimizações em hardware possam ser implementadas para esconder a latência de operações de escrita. Existem diversos exemplos para este tipo de relaxação, como ordenação fraca e consistência por liberação.

Neste tutorial, sempre será considerado o modelo de consistência sequencial, a menos que seja expresso o contrário. Isso pelo fato deste modelo ser o mais simples e não ser o objetivo deste tutorial aprofundar nos modelos de consistência de memória. Para uma análise detalhada dos modelos de consistêcia é possível consultar so trabalhos [1] e [14].

3.2 Protocolos Baseados em Snooping

Uma maneira de implementar os protocolos descritos na Seção 2 é utilizar um meio centralizado, em particular um barramento chamado barramento *snoopy* [17]. Esse meio conecta todas as *caches* fornecendo uma maneira natural de comunicação broadcast para que os controladores de *cache* possam trocar requisições de coerência entre si.

Quando um núcleo emite uma requisição para sua *cache* (requisição interna), o controlador de coerência examina o estado atual da *cache*, o atualiza e toma a ação adequada, a qual pode significar gerar transações (requisições de coerência) no barramento. A coerência é mantida, pois os controladores de coerência de cada núcleo monitoram o barramento para tomar a ação necessária de acordo com a requisição recebida.

Uma mensagem recebeida por um controlador por meio do barramento (requisição externa) deve ser verificada para decidir se alguma ação deve ser tomada. Isso é feito de maneira semelhante a verificação de *tag* para identificar a presença de um dado na *cache*. A ação realizada pode envolver invalidar ou atualizar o conteúdo ou estado daquela linha de *cache* ou fornecer o último valor do bloco de *cache* para o barramento.

O uso de um barramento fornece duas propriedades a esta abordagem. A primeira é a garantia que todas as transações emitidas serão visíveis a todos os controladores. E segundo é a ordenação dos eventos, pois isto é fornecido considerando a ordem em que as requisições aparecem no barramento.

A política de atualização de *cache* também afeta como a coerência é garantida em um protocolo *snooping*. Nas próximas duas seções serão examinados as políticas *write-through* e *write-back*.

3.2.1 Write-through Em um protocolo *write-through* todas as escritas aparecem no barramento. Essa característica facilita a manutenção da coerência com a utilização de um protocolo *snooping*. Desde que apenas um barramento está sendo utilizado em um dado momento, todas as escritas para um local são serializadas pela ordem na qual elas aparecem no barramento. Como cada controlador de *cache* executa a invalidação durante a transição do barramento, invalidações são executadas por todos os controladores de *cache* naquela ordem. Desta forma o protocolo impõe uma ordenação parcial, da qual pode ser construído uma hipotética ordem total que permite manter a coerência. A ordem pode ser formalizada como segue:

- Uma operação de memória M_1 é subsequente a uma operação de memória M_2 se as operações são emitidas pelos mesmo processadores na mesma ordem.
- Uma operação de leitura é subsequente a uma operação de escrita se a leitura gera uma transação de barramento que segue aquela para a operação de escrita.

- Uma operação de escrita é subsequente a uma operação de leitura ou escrita M se M gera uma transição de barramento e a transição de barramento para a escrita segue aquela para M.

- Uma operação de escrita é subsequente a uma operação de leitura se a leitura não gera uma transição de barramento (se refere ao bloco já estar na *cache*) e não está separada da escrita por uma outra transação de barramento.

Essa ordem é estabelecida com a política *write-through* pois cada instrução de escrita é imediatamente traduzida em uma transação de escrita no barramento, o qual impede que outra transação seja executa antes que a anterior termine.

3.2.2 Write-back Como demostrado na Seção **??** em *caches write-back* existe um estado modificado que determina a posse do bloco, o qual determina uma exclusividade que a *cache* possui para modificar uma linha sem a necessidade de notificar as mudanças para os outros núcleos.

Para realizar esta ação o barramento deve fornecer um tipo particular de transação chamada *read exclusive*. Esta transação é enviada de um controlador de coerência quando ele recebe uma requisição de escrita de outro núcleo para um bloco que: não está na *cache*, ou está na *cache*, mas não no estado modificado.

Quando o controlador observa uma transação *read exclusive* no barramento, ele deve determinar se a linha de *cache* requisitada está presente e, neste caso, deve invalidar sua cópia. A transação *read exclusive* garante que aquela *cache* que está escrevendo em um bloco tem a única cópia válida.

Outro tipo de transação requerida em *caches write-back* é a transação de escrita gerada pelo controlador de *cache* na atualização da memória. Esta transação também é gerada pela operação *flush* depois de uma requisição, de outro núcleo, para a linha que está na *cache* com o estado modificado.

Embora nem todas as escritas gerem transações no barramento, entre duas transações para um mesmo bloco, apenas o núcleo que obteve uma cópia exclusiva pode executar operações de escrita neste bloco. Quando outro núcleo emite uma requisição de leitura para o bloco existe ao menos uma transação no barramento (gerada pela operação *flush*) para aquele bloco que separa o término da operação de escrita do primeiro núcleo para a requisição de operação de leitura do segundo. Esta transação garante que todas as operações de leitura verão os efeitos da operação de escrita anterior.

3.3 Protocolos Baseados em Diretórios

Uma abordagem escalável para realizar coerência de *cache* é aquela baseada em diretórios. Diferentemente da abordagem anterior em que o estado de uma linha de *cache* é determinado implicitamente qualificando as requisições no barramento compartilhado, a idéia agora é manter o estado explicitamente em um local chamado diretório [17].

O diretório pode ser organizado como uma memória *cache* que guarda um registro de cada linha de *cache*. Esse registro contem o estado da linha, bem o registro de todos os núcleos que possuem essa linha em sua *cache* atualmente e é chamado entrada de diretório para aquela linha de *cache*.

Quando um núcleo gera um *miss* para um bloco de *cache*, ele primeiro comunica com o diretório para determinar a informação do bloco, então determina onde as cópias válidas estão e quais ações deve tomar. Todos os acessos para o diretório e qualquer outro tipo de comunicação entre os núcleos são realizados pelos controladores de coerência, os quais agem para manter a consistência. Portanto, mecanismos de coerência baseados em diretórios são independentes de estruturas de interconexão.

Dado um protocolo, o sistema de coerência deve fornecer mecanismos para manter o protocolo, em particular ele deve executar os seguintes passos quando um *miss* ocorrer:

- Encontrar suficiente informação sobre o estado da linha de *cache* em outras *caches* para determinar qual ação tomar;
- Localizar essa cópias, se necessário e invalidá-las; e
- Comunicar com as outras *caches* e obter o dado delas ou invalidá-las, ou ainda atualizá-las.

Em protocolos baseados em *snooping* todas essas ações são executadas pelo mecanismo de monitoramento e *broadcast*. Em protocolos baseados em diretórios, as informações sobre os estados dos blocos em outras *caches* são encontradas pesquisando o diretório, enquanto que o local das cópias e quaisquer comunicações entre os núcleos são realizados por meio de comunicação internúcleo, sem reordenação de comunicação *broadcast*.

Em protocolos baseados em diretórios os núcleos são classificados da seguinte forma:

Residência Núcleo onde o local de memória e a entrada de diretório de um endereço reside.

Local ou requisitante Núcleo que emite uma requisição para o bloco.

Sujo Núcleo que tem uma cópia do bloco em sua *cache* no estado *modified* (sujo). Um nó pode ser sujo e home ao mesmo tempo.

Proprietário Núcleo que atualmente mantem uma cópia válida de um bloco e deve fornecer o dado quando necessário.

Exclusivo Núcleo que tem uma cópia do bloco em sua *cache* em um estado exclusivo.

4 Técnicas Avançadas

Nas seções anteriores foram discutidos os fundamentos que formam a base de sistemas de coerência de cache. Os protocolos das Seções 2.1, 2.2 e 2.3 e as abordagens de protocolos baseados em *snooping* e diretórios das Seções 3.2 e 3.3 constituem um alicerce para as soluções de coerência. Porém se aplicados estritamente sozinhos não são capazes de cumprir requisitos de desempenho, como tráfego, armazenamento, energia, tempo de execução e confiabilidade.

Dessa forma, esta seção apresenta algumas técnicas avançadas aplicadas a coerência de *cache* para atender requisitos de desempenho. É interessante notar que muitas das soluções são combinações de algumas soluções apresentadas anteriormente.

As técnicas que seguem incluem um mecanismo de desativação do protocolo de coerência para blocos privados, um mecanismo que faz coerência com *write-invalidate*, porém para um certo padrão de compartilhamento faz coerência com *write-update*, um mecanismo que utiliza um modelo de consistência relaxado, um mecanismo baseado em um protocolo híbrido, um mecanismo que mantém a coerência por meio de migração de computação e um sistema de coerência que junta diversas técnicas de coerência presentes na literatura.

4.1 Desativação do Protocolo de Coerência para Blocos Privados

Sistemas que realizam coerência por diretórios frequentemente armazenam as informações de rastreamento de todos os blocos em *caches* privadas no diretório. Entretanto, realizar um mapeamento completo de todos os blocos pode fazer do diretório uma estrutura grande e lenta, que não é capaz de cumprir requisitos de latência e espaço.

Uma alternativa adotada por muitas arquiteturas para minimizar esses problemas é projetar um diretório que mantem apenas uma parcela de todos os blocos de *cache*. Dessa forma, uma entrada do diretório mantem as informações de rastreamento de um bloco de *cache*, porém este bloco está associado a vários outros. Este tipo de mapeamento faz com que todos os blocos associados a uma entrada do diretório sejam invalidados quando ela é despejada dele para dar lugar a outro bloco. Muitos desses blocos associados, de fato, não estão compartilhados, porém devido à natureza do mecanismo são tratados como se estivessem.

Cuesta *et al* [6] conduziram o estudo de um sistema em que a coerência é mantida por caches de diretórios como descrito acima. Entretanto, o protocolo de coerência é desativado

para blocos que são privados, inclusive àqueles associados a alguma entrada no diretório. A abordagem proposta pode ser utilizada com qualquer um dos protocolos descritos nas Seções 2.1, 2.2 e 2.3, porém no estudo experimental foi utilizado o protocolo `MOESI`. O principal objetivo do trabalho foi aumentar a eficiência do protocolo de diretório por meio da desativação da coerência para blocos de memória privados.

O trabalho de Cuesta *et al* [6] demonstraram que grande parte dos blocos de memória de aplicações paralelas são de dados privados, que não são compartilhados entre processos e *threads* da aplicação. Desta form, não há razão para esses blocos serem gerenciados pelo protocolo de coerência e menos ainda, ocuparem uma entrada no diretório, que, como visto acima, é um recurso limitado.

A desativação do protocolo de coerência é possível por meio de um mecanismo que classifica os blocos como privados ou compartilhados e desativa o protocolo para aqueles classificados como privados fazendo com que eles não sejam armazenados no diretório. Este mecanismo trabalha em uma granularidade de página de memória. Portanto, tanto a classificação dos blocos como a desativação do protocolo é feita a nível de sistema operacional com o auxílio do hardware, embora que em pequena escala.

Para identificar as páginas que são realmente compartilhadas o mecanismo realiza uma série de ações. Inicialmente todas as páginas de memória são classificadas como privadas e para essas páginas o mecanismo de coerência é desativado. Quando ocorre uma falta de um bloco que pertence a uma página privada, o compartilhamento deste bloco com outros núcleos não é verificado e nenhuma informação dele é mantida no diretório. Por outro lado, todos os blocos que pertencem a uma página com o *status* de compartilhada têm suas informações de rastreamento armazenadas no diretório. Apesar de muitos desses blocos serem privados, todos blocos de uma página classificada como compartilhada são considerados compartilhados também.

São consideradas compartilhadas todas as páginas que possuem blocos acessados por mais de um núcleo. O sistema operacional é responsável pela detecção deste compartilhamento. Para isso, cada entrada da TLB (*translation lookaside buffer*) é acrescentada de alguns campos que armazenam a identidade do primeiro processador a solicitar esta página. Quando uma requisição para esta página acontece o SO verifica se o processador que realizou a requisição é o mesmo que esta amarzenado na entrada desta página na TLB. Caso não seja, esta página é considerada compartilhada, pois dois processadores acessam blocos da mesma página.

Cuesta *et al* demonstraram que este mecanismo de desativação do protocolo de coerência reduz a taxa de *miss* em cerca de 35%, acarretando uma melhoria no desempenho de 15%. Como menos acessos são feitos ao diretório (5% menos) e a memória (20%) menos tráfico é realizado entre os níveis de memória (15%) ocasionando uma redução na dissipação de

energia dinâmica.

4.2 Protocolo Adaptativo Baseado em Padrão de Compartilhamento

Na Seção 2 todos os protocolos foram apresentados sob a definição de que o protocolo de coerência era *write-invalidate*. No entanto, existe outra categoria de protocolo chamada *write-update*. A diferença fundamental entre os dois protocolos é que no *write-update*, quando um bloco compartilhado vai ser escrito por algum núcleo, ao invés das suas cópias serem invalidadas, elas são atualizadas a cada operação de escrita. Este tipo de protocolo geralmente não é utilizado na prática, pois além de sua implementação ser mais complexa a quantidade de tráfego é consideravelmente maior do que no *write-invalidate*.

Não obstante, aplicações com altas taxas de operações de leitura se beneficiam de protocolos *write-update*, pois o dado a ser lido pelo núcleo sempre está atualizado. Neste caso o *write-invalidate* possui uma desvantagem: como os blocos compartilhados são invalidados quando algum núcleo deseja escrever em algum deles, sempre que algum outro núcleo necessita ler a linha de *cache*, ele precisa fazer uma requisição de coerência para os outros núcleos.

Para tentar contornar os problemas envolvendo os dois tipos de protocolo o trabalho de Kayi e El-Ghazawi [10] propôs um protocolo de coerência adaptativo. Kayi e El-Ghazawi propõem é um mecanismo de previsão em um protocolo *write-invalidate*, que realiza atualizações de escrita para linhas que apresentam um padrão de compartilhamento produtor-consumidor.

O padrão produtor-consumidor ocorre quando um núcleo N_a realiza uma operação de escrita para um bloco w e em seguida mais de um núcleo $N_x \neq P_a$ realiza uma operação de leitura para w. Neste caso a é o produtor e x é o consumidor.

A previsão que o mecanismo faz é se o padrão produtor-consumidor ocorre para alguma linha de *cache* w. Caso isso aconteça, então ao invés de invalidar as cópias de *cache* do conjunto de núcleos que estão atuando como consumidores, elas são atualizadas. Desta forma, o objetivo é prever com o máximo de precisão o conjunto de núcleos consumidores. Assim, as operações de leitura não irão envolver requisições de coerência adicionais.

Para realizar esta previsão, em cada núcleo é implementado um previsor de padrão de compartilhamento. Este previsor pode ser visto como uma *cache* que armazena para cada bloco compartilhado as informações necessárias para realizar a previsão. Para determinar se um bloco é compartilhado o previsor utiliza um *bit* para checar durante um *miss* na *cache* se o núcleo que está realizando a leitura corresponde ao último escritor. Assim, as linhas com diferentes leitores são classificadas como blocos de coerência.

As principais vantagens da utilização deste previsor é que ele pode ser utilizado tanto

em um modelo de consistência sequencial, quanto em um modelo relaxado. De maneira semelhante, qualquer um dos protocolos descritos nas Seções 2.1, 2.2 e 2.3 podem ser utilizados. Além disso, o único acréscimo de hardware necessário é a implementação do previsor para cada núcleo.

Os experimentos realizados mostraram que utilizando esta técnica de previsão foi possível reduzir a quantidade de *misses* devido a requisições de coerência em cerca de 20%. O tempo de execução das aplicações também foi reduzido, na média elas executaram 5% mais rápido com o protocolo adaptativo.

4.3 Coerência de Cache com Modelo de Consistência Relaxado

Um modelo de consistência sequencial requer que os acessos à memória por cada núcleo sejam ordenados e acessos entre diferentes núcleos sejam arbitrariamente intercalados [9]. Este modelo fornece um paradigma de programação fácil e simples, porém como todos os núcleos devem aguardar até que o núcleo que está executando uma ação de leitura ou escrita termine sua operação, ele geralmente possui problemas de desempenho, principalmente em sistemas com grandes quantidades de núcleos.

Em contrapartida ao modelo sequencial existem os modelos de consistência relaxados. Nesses modelos não é necessário que as operações realizadas pelos processadores tenham uma ordem. Na verdade, elas ocorrem fora de ordem e mecanismos de sincronização são utilizados para forçar a ordenação. A principal vantagem desses modelos é que os processadores não precisam aguardar até que algum outro núcleo termine sua operação, pois a ordem é forçada em pontos de sincronização.

Ashby *et al* [3] propuseram um esquema de coerência híbrido hardware-software que utiliza o modelo de consistência relaxado *acquire/require* sob um protocolo de coerência *write-invalidate*. Neste mecanismo a parte de sofware é responsável por disparar as ações de coerência e um dispositivo de hardware é utilizado para selecionar as operações de invalidações a fim de ganhar desempenho.

Esse mecanismo fornece três operações para que os programadores possam gerenciar os dados que são compartilhados: *acquire*, *require* e *barrier*. Essas operações são pontos de sincronização. Cada núcleo pode ler e escrever linhas de *cache* compartilhadas em sua *cache* privada e a coerência é forçada quando algum ponto de sincronização é alcançado.

Sempre que um programa vai fazer uso de uma variável compartilhada ele utiliza uma sequência de operações *acquire* $\rightarrow$ *release* ou *barrier*. Quando um núcleo N_1 alcança um *acquire* ou *barrier*, todas as linhas de *cache* dele são invalidadas, assim se este núcleo ler ou escrever em uma variável x compartilhada após o *acquire* ou *barrier* um *miss* na cache irá ocorrer. Se o núcleo N_2 também já possuí x em sua *cache*, como ele também utilizou

um *acquire* ou *barrier* ele poderá modificar o valor da variável compartilhada e quando N_2 realizar a operação de *release* um *write-back* ocorrerá atualizando o nível imediatamente superior de memória com o novo valor de x. Como a leitura de x por N_1 causou uma *miss* na *cache*, N_1 precisará buscar o novo valor no nível de memória superior, que devido ao *write-back* de N_2 estará atualizado.

Neste esquema sempre que ocorre um *acquire* ou *barrier* todas as linhas de *cache* daquele núcleo são invalidadas. Com certeza isso torna o mecanismo muito conservativo. Desta forma, Ashby *et al* utilizaram um dispositivo de hardware chamado filtro *bloom* [], que ao invés de invalidar todos os blocos, tenta selecionar de maneira mais agressiva apenas aqueles blocos que são compartilhados.

Os resultados com este trabalho demonstraram que em comparação com um esquema totalmente baseado em software e outro totalmente baseado em hardware, o mecanismo híbrido teve tempo de execução cerca de 93% menor do que o primeiro e teve tempo de execução próximo ao esquema totalmente em hardware por cerca de 5% para a maior parte dos *benchmarks* avaliados.

4.4 Coerência de Cache com Protocolos Híbridos

Nas Seções 3.2 e 3.3 foram salientadas diversas questões de projeto que afetam de maneira posita e negativa o desempenho de sistemas com protocolos baseados em *snooping* e diretórios. Protocolos *snooping* tem bom desempenho em sistemas com poucos núcleos, pois a disputa pelo barramento se torna menos intensa. Por outro lado, protocolos baseados em diretórios são escaláveis para grandes quantidades de núcleos, apesar de muitas vezes o seu tamanho limitar a escalabilidade.

Analisando esses *trade-offs* é possível pensar em unir o melhor dos dois mundos e criar um protocolo que realize a coerência tanto por *snooping* quanto por diretório. Zhao *et al* [22] projetaram um sistema que faz exatamente isso. Para uma arquitetura com CMPs Zhao *et al* criaram agrupamentos de núcleos que realizam coerência internamente com um protocolo baseado em *snooping*. Por sua vez, a coerência entre todos os agrupamentos é mantida por um protocolo baseado em diretório.

Um problema é que só faz sentido ter um agrupamento entre núcleos se os núcleos que compartilham o mesmo dado estiverem no mesmo agrupamento, caso contrário diversas requisições entre agrupamentos serão realizadas impedindo o ganho de um barramento *snooping* conectando núcleos próximos. No entanto, não há garantias que o sistema operacional tenha esse conhecimento e possa escalonar *threads* que compartilham o mesmo dado em núcleos próximos espacialmente. Para contornar este problema Zhao *et al* também propõem um mecanismo que cria os barramentos *snooping* dinamicamente.

Quando um núcleo faz uma requisição para uma linha de *cache* e ocorre um *miss* na *cache*, ao invés do núcleo enviar uma requisição para o diretório, ele primeiro envia uma requisição para o barramento *snooping*. Se algum outro núcleo possui essa mesma linha de *cache* ele a coloca no barramento e o núcleo requisitante pode obter seu valor como se fosse um protocolo *snooping* normalmente.

Caso nenhum dos outros núcleos pertencentes àquele agrupamento tenham a cópia requisitada, o núcleo requisitante envia uma requisição para o diretório e a partir disso requisições serão enviadas para os outros agrupamentos e se algum deles possuir o dado este será recuperado como em um protocolo baseado em diretório. Caso nenhum núcleo possua uma cópia válida o núcleo requisitante inicial busca o dado atualizado na memória.

É interessante observar que do ponto de vista do protocolo baseado em diretório cada agrupamento é como se fosse apenas um núcleo, pois dentro de um agrupamento a coerência é mantida por meio do protocolo *snooping*. Além disso, quando uma nova *thread* de aplicação é mapeada para um conjunto de núcleos o mecanismo cria um novo barramento apenas com esses núcleos.

Os resultados apontados por Zhao *et al* indicam que com a criação de agrupamentos dinâmicos com oito núcleos diminui a latência de falta em leituras em cerca de 20%. Dentro de cada agrupamento, onde a coerência é mantida por *snooping* a quantidade de vezes que um dado procurado está no agrupamento local é de aproximadamente 50% na média. Além disso, mais instruções por ciclo e menos tráfego é gerado o que implica em um menor consumo de energia, de maneira que para algumas aplicações a economia chega em 37%.

4.5 Protocolo Baseado em Diretório com Migração de Dados

Uma questão de projeto que é muitas vezes pouco abordada diz respeito à confiabilidade de uma arquitetura. Por exemplo, como um sistema pode se recuperar se um núcleo falhar ou mesmo se algum controlador de diretório falhar? Questões como estas também devem ser levantadas na definição de um projeto arquitetural e respondidas na implementação do hardware para que o sistema não fique inoperante e não ocorra uma perda de desempenho visível ao usuário.

Neste contexto, uma alternativa proposta no trabalho de X [11] foi o projeto de uma arquitetura híbrida, em que existe um protocolo de coerência baseado em diretórios e um mecanismo de coerência baseado em migração de computação. Os dois mecanismos podem trabalhar juntos ou separadamente e a ativação e desativação de um ou outro pode ser feita durante a execução do sistema.

O protocolo de diretório é como descrito anteriormente e o mecanismo de migração de computação é um esquema que torna o sistema mais confiável e é chamado de EM (*execution*

migration). Este esquema garante que nenhum dado a ser escrito esteja compartilhado entre as *caches* e isso faz com que não seja necessário diretórios. O EM consegue isso por transferir o contexto de execução de uma *thread* de um núcleo para outro sempre que algum dado solicitado por essa *thread* não se encontra na *cache* privada do núcleo original, mas está presente na de outro núcleo. Dessa forma, o EM transfere a execução do programa (*thread*) para os núcleos em que os dados solicitados estão presentes. Existem três conceitos relacionados a este protocolo, a saber:

1. *Core home*: núcleo que possui o dado que uma instrução de programa solicita.
2. *Core hit*: quando o núcleo que está executando o programa possui o dado solicitado.
3. *Core miss*: como o anterior, porém o núcleo não possui o dado.

Quando um núcleo C executa uma instrução que necessita de um dado que está em alguma posição de memória $addr$, o EM verifica quem é o núcleo *home* H para este endereço $addr$. Se o próprio núcleo C é o nó *home* ($C = H$), então ocorre um *core hit* e a requisição do dado continua normalmente para a hierarquia de memória. Caso C não seja o *home* ($C \neq H$), então ocorre um *core miss*. Neste último caso, o núcleo C para sua execução e carrega o contexto da *thread* que estava sendo executada para o núcleo H e o acesso à memória é então realizado localmente em H.

O mecanismo EM trabalha a nível de página. Caso algum outro protocolo de coerência seja usado este deve trabalhar a nível de blocos de *cache*, como por exemplo um protocolo de diretório. Desse modo, o EM não interfere no protocolo subjacente e a informação do núcleo que é *home* pode ser mantida na TLB.

Essa técnica de migração da computação permite que o sistema se mantenha operacional caso uma falha ocorra a algum diretório, controlador ou mesmo núcleo. Existem quatro cenários que *Khal et al* identificam como possíveis em uma arquitetura com o EM e um protocolo subjacente baseado em diretório:

1. **Diretório para EM**: o protocolo de diretório está ativado, mas o EM não. Neste cenário o intervalo de endereços que este diretório mantinha passa a ser controlado pelo mecanismo EM. Uma justificativa para isso é o caso de algum controlador de diretório falhar. Nesta situação o diretório não pode ser utilizado, porém o sistema pode continuar operando por meio do EM.
2. **Diretório para Diretório e EM**: o protocolo de diretório está ativado, mas o EM não. Este cenário é parecido com o anterior, existem blocos controlados pelo EM e outros pelo outro protocolo no mesmo diretório. Uma situação em que isso pode ser útil é

quando alguns blocos de *cache* são desabilitados por causa de erros. Desta forma, eles podem ser controlados pelo EM, enquanto os outros continuam sendo controlados pelo protocolo de diretório.

3. **Diretório e EM para EM**: os dois protocolos estão ativados. Este cenário é igual ao primeiro. Ele é útil caso algum controlador de diretório pare de funcionar.

4. **EM para Diretório e EM**: apenas o protocolo EM está ativado. Neste cenário os dois protocolos são ativados. Essa situação é interessante quando algum *link* de migração falha, então os blocos controlados por ele podem ser controlados pelo protocolo de diretório.

4.6 Um Sistema de Coerência para Escalar Muitos Núcleos

Outro problema relacionado aos protocolos de coerência é a questão da escalabilidade. Como descrito na Seção 3 protocolos baseados em *snooping* são pouco escaláveis devido ao uso de um barramento compartilhado. Protocolos baseados em diretórios conseguem escalar melhor, porém para sistemas com grandes quantidades de núcleos (acima de 100) também podem apresentar limitações de desempenho.

Martin *et al* [13] propõem um sistema de coerência baseado em diretórios que emprega diversas técnicas para torná-lo escalável para grande quantidades de núcleos. As técnicas sugeridas para tornar o sistema escalável foram rastreamento preciso de núcleos, projeto hierárquico de *caches* privadas e notificações explícitas de despejos.

Rastrear precisamente os núcleos pode ser feito com *caches* compartilhadas inclusivas. Neste tipo de *cache*, todo bloco em qualquer *cache* privada também está presente na cache compartillhada. Além disso, cada bloco de *cache* compartilhada armazena, além da informação de estado do bloco, a informação de quais núcleos possuem aquele bloco. Dessa forma, quando uma requisição de coerência chega ao controlador da *cache* compartilhada ele pode enviar as requisições (por exemplo invalidações) exatamente para os núcleos que possuem aquele bloco em suas *caches* privadas ou responder diretamente para o núcleo requisitante.

Um projeto de *cache* privada hierárquico consiste na criação de diversos agrupamentos de núcleos. Cada agrupamento possui um número k de núcleos com suas *caches* privadas e uma rede de interconexão que conecta todos os núcleos do agrupamento. Também conectada a esta rede está a *cache* do agrupamento que é compartilhada por todos os núcleos daquele agrupamento. Todos os agrupamentos são conectados por uma rede de interconexão dos agrupamentos, que por sua vez também conecta uma cache compartilhada para todos os agrupamentos.

Quando um núcleo modifica um bloco de *cache* que está compartilhado ele deve es-

crever este bloco de volta para a memória, o que é comumentemente chamado de despejo. Tal operação é realizada explicitamente, isso significa que antes de enviar o dado para a *cache* compartilhada o controlador da *cache* privada primeiro envia uma notificação de que fará isso, em seguida a *cache* compartilhada emite uma resposta a esta notificação e ao recebê-la o controlador envia o dado para a *cache* no nível superior.

A fim de demonstrar como o problema da escalabilidade pode ser resolvido, Martin *et al* analisaram o sistema proposto em cinco aspectos: tráfego, armazenamento, manutenção de inclusão, latência e energia. Cada um desses parâmetros foi avaliado para configurações com diferentes quantidade de núcleos para descobrir se cada um desses parâmetros diminuiriam o desempenho se a quantidade de núcleos fosse consideravelmente grande.

Para analisar o tráfego Martin *et al* mediram quantos *bytes* são transferidos para obter e abandonar um dado de bloco de *cache* para cada *miss* na *cache*. Eles demonstraram que quando núcleos que compartilham linhas de cache são rastreados precisamente, o tráfego por *miss* na *cache* é independente da quantidade de núcleos.

Para rastrear os núcleos precisamente é necessário que na *cache* compartilhada mais espaço seja requerido para cada bloco de *cache*. Por exemplo, um sistema com 1024 núcleos requeriria 128 *bytes* a mais em cada bloco de *cache* compartilhada, o que não é escalável. Entretanto, utilizando a técnica de projeto hierárquico com um número k de núcleos por agrupamento seria necessário para o exemplo anterior apenas k *bits* em cada bloco de *cache* compartilhada do agrupamento.

Referências

[1] ADVE, S. V., AND GHARACHORLOO, K. Shared Memory Consistency Models: A Tutorial. *Computer 29*, 12 (Dec. 1996), 66–76.

[2] AGARWAL, A., SIMONI, R., HENNESSY, J., AND HOROWITZ, M. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the Annual International Symposium on Computer Architecture* (Los Alamitos, CA, USA, 1988), IEEE Computer Society Press, pp. 280–298.

[3] ASHBY, T. J., DIAZ, P., AND CINTRA, M. Software-Based Cache Coherence with Hardware-Assisted Selective Self-Invalidations Using Bloom Filters. *IEEE Transactions on Computer 60*, 4 (Apr. 2011), 472–483.

[4] BAER, J.-L. *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*, 1st ed. Cambridge University Press, New York, NY, USA, 2009.

[5] CHAIKEN, D., FIELDS, C., KURIHARA, K., AND AGARWAL, A. Directory-Based Cache Coherence in Large-Scale Multiprocessors. *Computer 23*, 6 (June 1990), 49–58.

[6] CUESTA, B. A., ROS, A., GÓMEZ, M. E., ROBLES, A., AND DUATO, J. F. Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks. In *Proceedings of the Annual International Symposium on Computer Architecture* (New York, NY, USA, 2011), ACM, pp. 93–104.

[7] EGGERS, S. J., AND KATZ, R. H. Evaluating The Performance of Four Snooping Cache Coherency Protocols. *SIGARCH Computer Architecture News 17*, 3 (Apr. 1989), 2–15.

[8] HASHEMI, B. Simulation and Evaluation Snoopy Cache Coherence Protocols with Update Strategy in Shared Memory Multiprocessor Systems. In *Proceedings of the International Symposium on Parallel and Distributed Processing* (may 2011), pp. 256 –259.

[9] HENNESSY, J., AND PATTERSON, D. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2011.

[10] KAYI, A., AND EL-GHAZAWI, T. An Adaptive Cache Coherence Protocol for Chip Multiprocessors. In *Proceedings of the Second International Forum on Next-Generation Multicore/Manycore Technologies* (New York, NY, USA, 2010), IFMT '10, ACM, pp. 4:1–4:10.

[11] KHAN, O., LIS, M., SINANGIL, Y., AND DEVADAS, S. DCC: A Dependable Cache Coherence Multicore Architecture. *IEEE Comput. Archit. Lett. 10*, 1 (Jan. 2011), 12–15.

[12] LAWRENCE, R. A survey of cache coherence mechanisms in shared memory multiprocessors, 1998.

[13] MARTIN, M. M. K., HILL, M. D., AND SORIN, D. J. Why On-chip Cache Coherence is here to stay. *Communications on the ACM 55*, 7 (July 2012), 78–89.

[14] NAEEM, A., CHEN, X., LU, Z., AND JANTSCH, A. Realization and Performance Comparison of Sequential and Weak Memory Consistency Models in Network-on-chip Based Multi-core Systems. In *Proceedings of the Design Automation Conference* (jan. 2011), pp. 154 –159.

[15] OLUKOTUN, K. *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*, 1st ed. Morgan and Claypool Publishers, 2007.

[16] PATTERSON, D. A., AND HENNESSY, J. L. *Computer Organization and Design: The Hardware/Software Interface*, 4th ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[17] PATTERSON, D. A., AND HENNESSY, J. L. *Computer Arquitecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.

[18] SORIN, D. J., HILL, M. D., AND WOOD, D. A. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[19] STALLINGS, W. *Computer Organization and Architecture - Designing for Performance (8. ed.)*. Pearson / Prentice Hall, 2006.

[20] STENSTRÖM, P. A Survey of Cache Coherence Schemes for Multiprocessors. *Computer 23*, 6 (June 1990), 12–24.

[21] TARTALJA, I., AND MILUTINOVIÁC, V. Classifying Software-Based Cache Coherence Solutions. *IEEE Software 14*, 3 (May 1997), 90–101.

[22] ZHAO, H., JANG, O., DING, W., ZHANG, Y., KANDEMIR, M., AND IRWIN, M. J. A Hybrid NoC Design for Cache Coherence Optimization for Chip Multiprocessors. In *Proceedings of the Annual Design Automation Conference* (New York, NY, USA, 2012), ACM, pp. 834–842.