CACHES L2 COMPARTILHADAS Em um esquema típico de cache de dois níveis em um SMP, cada processador possui uma cache L1 dedicada e uma cache L2 também dedicada. Nos últimos anos, tem crescido o interesse pelo conceito de uma cache L2 compartilhada. Em uma versão anterior do seu mainframe SMP, conhecido como geração 3 (G3), a IBM fez uso de caches L2 dedicadas. Em suas versões posteriores (séries G4, G5 e série z900), uma cache L2 compartilhada foi usada. Duas considerações definiram esta mudança:

1. Ao mudar de G3 para G4, a IBM duplicou a velocidade dos microprocessadores. Se a organização G3 fosse mantida, um aumento significativo de tráfego no barramento teria ocorrido. Ao mesmo tempo, houve um desejo de reutilizar o máximo possível de componentes G3. Sem uma atualização significativa do barramento, o BSN teria se tornado um gargalo.
2. Análises de cargas de trabalho típicas de mainframe revelaram um alto grau de compartilhamento de instruções e dados entre os processadores.

Estas considerações levaram os projetistas da G4 a considerar o uso de uma ou mais caches L2, cada uma sendo compartilhada por vários processadores (cada processador tendo um uma cache L1 dedicada no chip). À primeira vista, compartilhar uma cache L2 pode parecer uma ideia ruim. O acesso à memória a partir dos processadores deveria ser mais lento porque os processadores devem agora competir pelo acesso a uma única cache L2. No entanto, se uma quantidade suficiente de dados é, de fato, compartilhada por vários processadores, então uma cache compartilhada pode aumentar o rendimento ao invés de diminuí-lo. Dados que são compartilhados e encontrados na cache compartilhada são obtidos mais rapidamente do que se tivessem que ser obtidos pelos barramentos.

17.3 Coerência de cache e protocolo MESI

Nos atuais sistemas multiprocessadores, é comum haver um ou dois níveis de cache associados a cada processador. Esta organização é essencial para alcançar um desempenho razoável. No entanto, isso cria um problema conhecido como problema de ***coerência de cache***. Em essência, o problema é: várias cópias dos mesmos dados podem existir em caches diferentes simultaneamente e, se for permitido aos processadores atualizarem as suas próprias cópias livremente, isso pode resultar em uma imagem da memória inconsistente. No Capítulo 4, definimos duas políticas de escrita comuns:

- **Write-back**: operações de escrita são feitas normalmente apenas na cache. A memória principal é atualizada apenas quando a linha de cache correspondente é retirada da cache.
- **Write-through**: todas as operações de escrita são feitas na memória principal e na cache, garantindo que a memória principal sempre esteja válida.

É claro que uma política de write-back pode resultar em inconsistência. Se duas caches contêm a mesma linha, e a linha é atualizada em uma cache, a outra cache terá um valor inválido sem saber. Leituras subsequentes dessa linha inválida produzem resultados inválidos. Mesmo com a política de write-through, inconsistências podem ocorrer a não ser que outras caches monitorem o tráfego de memória ou recebam alguma notificação direta sobre a atualização.

Nesta seção, analisamos brevemente várias abordagens para o problema de coerência de cache e depois focamos na abordagem que é a mais usada: protocolo MESI, do inglês *Modified, Exclusive, Shared, Invalid*. Uma versão deste protocolo é usada nas implementações de Pentium 4 e PowerPC.

Para qualquer protocolo de coerência de cache, o objetivo é deixar que variáveis locais recém-usadas cheguem à cache apropriada e permaneçam aí durante várias leituras e escritas, enquanto o protocolo é usado para manter a consistência das variáveis compartilhadas que podem estar em várias caches ao mesmo tempo. Abordagens para coerência de cache geralmente têm sido divididas em abordagens por hardware e por software. Algumas implementações adotam uma estratégia que envolve tanto elementos de software quanto de hardware. Mesmo assim, a classificação em abordagens por software e por hardware ainda é instrutiva e comumente usada ao analisar as estratégias de coerência de cache.

Soluções por software

Esquemas de coerência por cache por software tentam evitar a necessidade de hardware adicional, circuitos e lógicas, contando com compilador e sistema operacional para lidar com o problema. Abordagens de software são atraentes porque a sobrecarga de detectar problemas potenciais é transferida do tempo de execução para o tempo de compilação e a complexidade de projeto é transferida do hardware para o software. Por outro lado, abordagens

de software em tempo de compilação geralmente devem tomar decisões conservadoras, levando à utilização ineficiente da cache.

Os mecanismos de coerência baseados em compiladores efetuam uma análise do código para determinar que itens de dados podem se tornar problemas se armazenados na cache; eles ainda marcam esses itens de maneira adequada. O sistema operacional ou hardware, então, evitam que esses itens indevidos sejam colocados em cache.

A abordagem mais simples é evitar que quaisquer variáveis de dados compartilhadas sejam colocadas na cache. Isto é conservador demais, porque uma estrutura de dados pode ser usada exclusivamente durante alguns períodos e pode ser efetivamente usada somente para leitura durante outros períodos. A coerência de cache se torna um problema apenas durante os períodos nos quais pelo menos um processo pode atualizar a variável e pelo menos um outro processado pode acessar a variável.

Abordagens mais eficientes analisam o código para determinar períodos seguros para variáveis compartilhadas. O compilador, então, insere instruções no código gerado para reforçar coerência de cache durante os períodos críticos. Uma série de técnicas tem sido desenvolvidas para efetuar a análise e para reforçar os resultados; veja análises em Lilja (1993[d]) e Stenstrom (1990[e]).

Soluções por hardware

Soluções baseadas em hardware são geralmente conhecidas como protocolos de coerência de cache. Estas soluções fornecem reconhecimento dinâmico em tempo de execução de condições de inconsistência potenciais. Como o problema é tratado apenas quando aparece de fato, há um uso mais eficiente de cache, o que leva a um desempenho melhor se comparado com a abordagem de software. Além disso, estas abordagens são transparentes ao programador e ao compilador, reduzindo o trabalho no desenvolvimento de software.

Esquemas de hardware diferem em uma série de particularidades, incluindo onde a informação sobre estado das linhas por dados é guardada, como essa informação é organizada, onde a coerência é reforçada e os mecanismos de reforço. Em geral, os esquemas por hardware podem ser divididos em duas categorias: protocolos de diretório e protocolos de detecção.

PROTOCOLOS DE DIRETÓRIO Protocolos de diretório coletam e mantêm a informação sobre onde as cópias das linhas residem. Normalmente, há um controlador centralizado que é parte do controlador da memória principal e um diretório que é guardado na memória principal. O diretório contém informação de estado global sobre o conteúdo de várias caches locais. Quando um controlador de cache individual faz uma requisição, o controlador centralizado verifica e emite comandos necessários para transferência de dados entre memória e caches e entre caches. Ele é responsável também por guardar a informação de estado atualizada; portanto, cada ação local que pode afetar o estado global de uma linha deve ser reportada para o controlador central.

Normalmente, o controlador mantém a informação sobre quais processadores têm uma cópia de quais linhas. Antes que um processador possa escrever em uma cópia local de uma linha, ele deve requisitar o acesso exclusivo para a linha ao controlador. Antes de conceder esse acesso exclusivo, o controlador envia uma mensagem para todos os processadores com uma cópia da cache dessa linha, forçando cada processador a invalidar a sua cópia. Depois de receber o reconhecimento de volta de cada processador, o controlador concede acesso exclusivo para o processador requisitante. Quando outro processador tenta ler uma linha que está exclusivamente concedida para outro processador, ele envia uma notificação de falha para o controlador. O controlador, então, emite um comando para o processador que guarda essa linha para que o processador escreva-a de volta na memória principal. A linha agora pode ser compartilhada para leitura pelo processador original e processador requisitante.

Esquemas de diretório tem a desvantagem de um gargalo central e de uma sobrecarga de comunicação entre os vários controladores de cache e o controlador central. No entanto, eles são eficientes em sistemas de grande escala que envolvem vários barramentos ou algum outro esquema complexo de interconexão.

PROTOCOLOS DE MONITORAÇÃO (Snoopy Protocols) Protocolos *snoopy* distribuem a responsabilidade de manter a coerência de cache entre todos os controladores de cache em um multiprocessador. Uma cache deve reconhecer quando uma linha que ela guarda é compartilhada com outras caches. Quando uma ação de atualização é feita em uma linha compartilhada na cache, ela deve ser anunciada para todas as outras caches por meio de um mecanismo de difusão (*broadcast*). Cada controlador de cache é capaz de "monitorar" na rede essas notificações de dispersão e reagir de acordo.

Protocolos de *snoopy* encaixam-se perfeitamente em um multiprocessador baseado em barramento, porque o barramento compartilhado fornece um meio simples para difusão e monitoramento. No entanto, como um dos

objetivos do uso de caches locais é evitar acessos ao barramento, cuidado deve ser tomado para que o tráfego de barramento aumentado para difusão e monitoramento não anule os ganhos do uso de caches locais.

Duas abordagens básicas para protocolo de detecção foram exploradas: write invalidate e write update (ou write broadcast). Com um protocolo de write invalidate, pode haver vários leitores, mas apenas um escritor ao mesmo tempo. Inicialmente, uma linha pode ser compartilhada entre várias caches para propósitos de leitura. Quando uma das caches deseja escrever na linha, ela primeiramente emite um aviso que invalida essa linha em outras caches, tornando a linha exclusiva para a cache que estará escrevendo. Uma vez a linha se tornando exclusiva, o processador proprietário pode fazer as escritas locais e baratas até que algum outro processador solicite a mesma linha.

Em um protocolo de write updates, pode haver vários escritores como também vários leitores. Quando um processador deseja atualizar uma linha compartilhada, a palavra a ser atualizada é distribuída para todas as outras e as caches que contêm essa linha podem atualizá-la.

Nenhum destes dois protocolos é superior a outro em todas as situações. O desempenho depende do número de caches locais e do padrão de leituras e escritas de memória. Alguns sistemas implementam protocolos adaptáveis que implementam ambos os mecanismos, write invalidate e write update.

A abordagem write invalidate é a mais usada em sistemas multiprocessadores comerciais, como Pentium 4 e PowerPC. Ela marca o estado de cada linha de cache (usando dois bits extras na marcação da cache) como modificada, exclusiva, compartilhada ou inválida. Por esta razão, o protocolo write invalidate é chamado de MESI.[1] No restante desta seção, analisamos o seu uso entre caches locais por meio de um multiprocessador. Para simplicidade da apresentação, não analisamos os mecanismos envolvidos em coordenação entre os níveis 1 e 2 localmente, assim como o tempo de coordenação pelo multiprocessador distribuído. Isso não adicionaria nenhum princípio novo, porém complicaria muito a discussão.

O protocolo MESI

Para fornecer a consistência de cache em um SMP, a cache de dados frenquentemente suporta um protocolo conhecido como MESI. Para o MESI, a cache de dados inclui dois bits de estado para cada *tag*, para que cada linha possa estar em um dos quatro estados:

- **Modificada**: a linha na cache foi modificada (diferente da memória principal) e está disponível apenas nesta cache.
- **Exclusiva**: a linha na cache é a mesma da memória principal e não está presente em nenhuma outra cache.
- **Compartilhada**: a linha na cache é a mesma da memória principal e pode estar presente em outra cache.
- **Inválida**: a linha na cache não contém dados válidos.

A Tabela 17.1 resume o significado dos quatro estados, e a Figura 17.7 mostra um diagrama de estado para o protocolo MESI. Tenha em mente que cada linha de cache tem os seus próprios bits de estado e, portanto, a sua própria instância do diagrama de estado. A Figura 17.7a mostra as transições que ocorrem por causa das ações iniciadas pelo processador associado a essa cache. A Figura 17.7b mostra as transições que ocorrem por causa dos eventos que são detectados no barramento comum. Esta apresentação de diagramas de estado separados para ações de iniciar processador e iniciar bar-

Tabela 17.1 Estado das linhas da cache MESI

	M Modificada	E Exclusiva	S (*shared*) Compartilhada	I Inválida
Esta linha da cache está válida?	Sim	Sim	Sim	Não
A cópia da memória está...	desatualizada	válida	válida	—
Há cópias em outras caches?	Não	Não	Talvez	Talvez
Uma escrita nesta linha...	não vai para barramento	não vai para barramento	vai para barramento e atualiza a cache	vai diretamente para barramento

1 Nota do tradutor: a letra S na sigla MESI vem do inglês *Shared* (compartilhada).

- Se nenhuma outra cache tem uma cópia da linha (limpa ou modificada), então nenhum sinal é retornado. O processador que iniciou lê a linha e passa a linha na sua cache de inválida para exclusiva.

LEITURA COM ACERTO (READ HIT) Quando uma leitura com acerto ocorre em uma linha que está atualmente na cache local, o processador simplesmente lê o item requerido. Não há mudança de estado: o estado permanece modificado, compartilhado ou exclusivo.

ESCRITA COM FALHA Quando ocorre uma escrita com falha na cache local, o processador inicia uma leitura de memória para ler a linha da memória principal contendo o endereço que faltou. Para este propósito, o processador emite um sinal no barramento que significa *leitura com intenção de modificar* (RWITM, do ingês *read-with-intent-to-modify*). Quando a linha é carregada, ela é imediatamente marcada como modificada. Em relação a outras caches, dois cenários possíveis antecedem o carregar da linha de dados.

Primeiro, alguma outra cache pode ter uma cópia modificada dessa linha (estado = modificado). Neste caso, o processador alertado sinaliza ao processador iniciante que outro processador tem uma cópia modificada da linha. O processador que iniciou entrega o barramento e espera. O outro processador obtém acesso ao barramento, escreve a linha de cache modificada de volta na memória principal e passa o estado da linha de cache para inválida (porque o processador que iniciou vai modificar esta linha). Subsequentemente, o processador que iniciou emite novamente um sinal RWITM para o barramento e depois lê a linha da memória principal, modifica a linha na cache e muda a linha para estado modificado.

O segundo cenário é quando nenhuma outra cache possui uma cópia modificada da linha requisitada. Neste caso, nenhum sinal é retornado e o processador que iniciou continua a ler a linha e a modificá-la. Enquanto isso, se uma ou mais caches possuem uma cópia limpa da linha no estado compartilhado, cada cache invalida a sua cópia da linha e se uma cache tiver uma cópia limpa da linha no estado exclusivo, ela invalida a sua cópia da linha.

ESCRITA COM ACERTO (WRITE HIT) Quando ocorre uma escrita com sucesso em uma linha que está atualmente na cache local, o efeito depende do estado atual dessa linha na cache local:

- **Compartilhada**: antes de efetuar atualização, o processador deve obter a propriedade exclusiva da linha. O processador sinaliza a sua intenção no barramento. Todo processador que tem uma cópia compartilhada da linha na sua cache passa-a de compartilhada para inválida. O processador que iniciou então efetua a atualização e passa a sua cópia da linha de compartilhada para modificada.
- **Exclusiva**: o processador já possui o controle exclusivo desta linha, então ele simplesmente efetua a atualização e passa a sua cópia da linha de exclusiva para modificada.
- **Modificada**: o processador já possui o controle exclusivo desta linha e a linha está marcada como modificada, então ele simplesmente efetua a atualização.

CONSISTÊNCIA DE CACHE L1-L2 Até agora descrevemos protocolos de coerência de cache em termos de atividade cooperativa entre caches conectadas ao mesmo barramento ou outro recurso de interconexão de SMP. Normalmente, estas caches são caches L2 e cada processador possui também uma cache L1 que não se conecta diretamente ao barramento e, portanto, não pode fazer parte de um protocolo de detecção. Assim, algum esquema é necessário para manter a integridade de dados entre ambos os níveis de cache e entre todas as caches na configuração SMP.

A estratégia é estender o protocolo MESI (ou qualquer protocolo de coerência de cache) para caches L1. Assim, cada linha na cache L1 inclui bits para indicar o estado. Basicamente, o objetivo é o seguinte: para cada linha que está presente na cache L2 e na sua cache L1 correspondente, o estado da linha L1 deve seguir o estado da linha L2. Uma forma simples de fazer isso é adotar a política de write through na cache L1; neste caso, a escrita direta é para a cache L2 e não para a memória. A política de write through de L1 força qualquer modificação em uma linha L1 para a cache L2 e assim a torna visível para outras caches L2. O uso da política de write through de L1 requer que o conteúdo de L1 seja um subconjunto do conteúdo L2. Isso, por sua vez, sugere que a associatividade da cache L2 seja igual ou maior que a associatividade de L1. A política de *write-through* de L1 é usada no IBM S/390 SMP.

Se a cache L1 tem uma política write-back, a relação entre as duas caches é mais complexa. Existem várias abordagens para manter a coerência. Por exemplo, a abordagem usada no Pentium II é descrita em detalhes em Shanley (2005[f]).

17.4 *Multithreading* e chips multiprocessadores

A medida mais importante de desempenho para um processador é a taxa em que ele executa as instruções. Isso pode ser expresso como:

$$\text{Taxa MIPS} = f \times IPC$$

onde *f* é a frequência de clock do processador, em MHz, e *IPC* (instruções por ciclo) é o número médio de instruções executadas por ciclo. De acordo com isso, os projetistas têm perseguido o objetivo de aumentar o desempenho em duas frentes: aumento de frequência de clock e aumento de número de instruções executadas ou, mais apropriadamente, o número de instruções completadas durante um ciclo do processador. Conforme vimos em capítulos anteriores, os projetistas aumentaram o IPC usando um pipeline de instruções e pipelines múltiplos paralelos de instruções em uma arquitetura superescalar. Com projetos de pipeline e pipelines múltiplos, o principal problema é maximizar a utilização de cada estágio do pipeline. Para melhorar o rendimento, os projetistas criaram mecanismos cada vez mais complexos, como executar algumas instruções em uma ordem diferente da forma que ocorrem no fluxo de instruções e começar a execução de instruções que podem nunca ser necessárias. Mas como foi discutido na Seção 2.2, esta abordagem pode estar alcançando o limite por causa da complexidade e dos problemas de consumo de energia.

Uma abordagem alternativa, a qual permite um grau mais alto de paralelismo em nível de instruções sem aumentar a complexidade dos circuitos ou consumo de energia, é chamada de *multithreading*. Basicamente, o fluxo de instruções é dividido em vários fluxos menores, conhecidos como *threads*, de modo que cada *thread* possa ser executada em paralelo.

A variedade de projetos específicos de *multithreading* realizada nos sistemas comerciais e nos experimentais é muito grande. Nesta seção, fazemos uma breve análise dos principais conceitos.

Multithreading implícito e explícito

O conceito de *thread* usado na discussão sobre processadores *multithread* pode ou não ser o mesmo que o conceito de *threads* de software em sistemas operacionais multiprogramados. Será útil definir os termos rapidamente:

- **Processo**: uma instância de um programa executando em um computador. Um processo engloba duas características principais:
  - **Posse do recurso**: um processo inclui um espaço de endereço virtual para guardar a imagem do processo; a imagem do processo é coleção de programa, dados, pilhas e atributos que definem o processo. De tempos em tempos, a um processador pode ser dada a posse (ou controle) de recursos, como memória principal, canais de E/S, dispositivos de E/S e arquivos.
  - **Escalonamento/execução**: a execução de um processo segue um caminho de execução (rastro) por um ou mais programas. Esta execução pode ser intercalada com a de outros processos. Assim, um processo possui um estado de execução (Executando, Pronto etc.) e uma prioridade de despacho, e é a entidade que é escalonada e despachada pelo sistema operacional.
- **Troca de processos**: uma operação que troca em um processador de um processo para outro, salvando todos os dados de controle do processador, registradores e outras informações do primeiro e substituindo-as com informações de processo do segundo.[3]
- ***Thread***: uma unidade de trabalho dentro de um processo que pode ser despachada. Ela inclui um contexto de processador (o qual inclui o contador de programa e o ponteiro de pilha) e sua própria área de dados para uma pilha (para possibilitar desvio de subrotinas). Uma *thread* executa sequencialmente e pode ser interrompida para que o processador possa se dedicar a outra *thread*.
- **Troca de *thread***: o ato de trocar o controle do processador de uma *thread* para outra dentro do mesmo processo. Normalmente, este tipo de troca é muito menos custoso do que uma troca de processo.

3 O termo ***troca de contexto*** é frequentemente encontrado em literatura e livros sobre SO. Infelizmente, embora a maior parte da literatura use este termo para se referir ao que é chamado aqui de troca de processo, outras fontes o usam para se referir à troca de *thread*. Para evitar ambiguidade, o termo não é usado neste livro.