Guión del grupo T20-Clase 4-05-2020

Slide 24-25

Como comentamos la clase pasada cuando hablamos de acceder a un dato y traérnoslo estamos hablando de una línea de cache, que es la granularidad más fina de la que se puede mantener coherencia. Sin embargo, una línea de cache no es lo “mínimo” que se puede direccionar. Por lo tanto, hablaremos de **“true sharing”** cuando desde dos procesadores distintos se accede a la misma dirección de memoria, y hablaremos de **“false sharing”** cuando desde dos procesadores distintos no se accede a la misma dirección de memoria pero sí a la misma línea de cache.

False sharing resulta ser una ineficiencia, cuando al menos uno de los accesos desde procesadores diferentes a la misma línea de cache (pero no la misma dirección de memoria) es para escritura. Estas acciones dispararan el protocolo de coherencia aun cuando ni siquiera se estén leyendo/escribiendo la misma dirección de memoria.

Este tipo de ineficiencias no genera programas incorrectos, pero sí programas más ineficientes y en consecuencia se deben evitar. Una técnica para evitarlo es “padding”. Donde se agrega artificialmente una distancia (se reserva un espacio de memoria sin utilidad para el programa) entre las dos variables responsables del “false sharing”, para que pasen a estar en diferentes líneas de cache (slide 25). Al estar en líneas diferentes no puede producirse, entre ellas, el problema de «false sharing».

En el ejemplo hay dos tareas que se pueden ejecutar en paralelo. La estructura f tiene dos campos: x e y, los dos de tipo int, que en los procesadores actuales ocuparían 4 bytes cada uno, un total de 8 bytes. Por consiguiente, la estructura puede caber tranquilamente en una única línea de cache y podemos suponer que estarán en la misma línea si consideramos que la estructura f está alineada a tamaño de línea (empieza con una dirección múltiplo de tamaño de línea).

La primera tarea realiza un bucle y va actualizando la variable compartida «s» con los valores que lee del campo x de la estructura f. En el primer acceso a f.x (asumiremos, por ejemplo, que f.x se guarda en la dirección 0x0400004), habrá un miss que hará que el snoopy provoque un BusRd y se copia la línea de memoria donde guarda esa estructura y campo (asumiremos, por ejemplo, que se guarda en una linea de memoria con la dirección 0x0400000, y ocupa 16 bytes, pudiendo guardar hasta 4 enteros de tipo int por línea), en la cache del thread que está ejecutando esa task.

La segunda task realiza un acceso a f.y (que si f.x asumimos que se guardaba en 0x0400004, este campo se guardará en la dirección 0x0400008, 4 bytes después que la f.x). En este caso hace una lectura y una escritura sobre el campo:

```
f.y = f.y + 1
```

Esto implica una lectura, que provocará un PrRd, seguido por un BusRd por parte del snoopy, para que se copie la línea de memoria en la cache del procesador donde se ejecute la tarea. Recordad que f.x y f.y forman parte de la misma estructura guardada en la misma línea de memoria que empieza en 0x0400000. Posteriormente, se querrá incrementar en 1 f.y, y por consiguiente se producirá un PrWr, seguido de un BusUpgr por parte del snoopy. Este BusUpgr afecta a la coherencia a las copias de la línea de memoria guardada en 0x0400000.

Por consiguiente, qué le sucede a la copia que había en el procesador que tenia una copia de la línia de memoria guardada 0x0400000 y que había accedido a f.x, con dirección 0x0400004? Siguiendo el protocolo MSI, su snoopy verá un BusUpgr en el bus contra la línea de la que tiene copia y la invalidará.

Por consiguiente, observamos que aunque los dos threads acceden a datos con direcciones de memoria distintas : f.x con 0x0400004 y f.y con 0x0400008, ambos campos están afectados por el sistema de coherencia, que aplica a líneas de cache. Cuando esto pasa, cuando accesos a distintas posiciones de memoria caen en la misma línea de memoria, siendo alguno de estos accesos de escritura, se dice que se ha producido false sharing. Si hubieran accedido al mismo dato con la misma dirección de memoria se hubiera producido true sharing.

¿Cómo podemos evitar este false sharing entre los dos threads del ejemplo? Lo que proponemos para este caso es hacer que los dos campos f.x y f.y esten mapeados en líneas de cache distintas. Para ello introducimos un campo extra que nos separe adecuadamente estos dos campos, es decir, estamos añadiendo padding. El padding que necesitamos en el ejemplo con líneas de 16 bytes, es de 3 int (12 bytes), con ello el campo f.x estará mapeado en la dirección 0x0400004 y el campo f.y en la dirección 0x0400010.

Problema 4 : False sharing.

Slide 26-29

Los sistemas multicores, que los podéis encontrar en la mayoría de portátiles, sobremesas y móviles actuales, surgen de la necesidad de aumentar el rendimiento actual de los procesadores, aprovechando la tecnologia existente que permite el aumento en el número de transistores que caben en un chip. En la gráfica de la Slide 26 se muestra el crecimiento en el número de transistores en un chip en el tiempo.

Hasta el año más o menos 2004, la forma de aumentar el rendimiento de los procesadores de uso más común era aumentando su frecuencia. Pero eso implicó que se consumía más y además, en un momento dado, que el rendimiento del procesador no fuera linealmente más rápido, e incluso peor. Es por ello que se optó por tener más de un core en el mismo procesador, con menor frecuencia a la de los procesadores en aquellos momentos. Teniendo más de un core ayuda a explotar el paralelismo, con un consumo menor y menos problemas de calentamiento del procesador. De ahí el interés general de saber explotar esos procesadores multicore usando paralelismo en nuestras aplicaciones.

Estos multicores tienen una jerarquía de memoria en la que se comparte la memoria principal y el LLC (last level of cache). El resto de niveles de cache son privados.

En este caso, si tenemos varios multicores conectados a una memoria principal a través de un bus, el sistema de coherencia se mantiene a nivel del LLC, seguramente guardando información extra a la que hemos hablado hasta el momento para saber qué cores tienen copia en su cache privada también, de las líneas que se tienen en el LLC compartido entre cores.

Así, podemos tener varios multicores conectados al bus. Hasta cuantos? Si aumentamos el número llegará un momento que el sistema no escalará debido a la contención en los accesos a memoria

principal y el mantenimiento de coherencia a través del bus (en el video se hablaban de máximo 8-16 cores).

Slide 30

En esta transparencia se muestran alternativas para poder coseguir escalar el sistema. En este caso cada multicore tiene una porción de memoria más cercana. Además, entre los diferentes módulos de memoria y procesadores hay algun tipo de conexión que nos permita tener la visión de memoria compartida, aunque físicamente distribuida entre los procesadores.
Estos sistemas, además, pueden permitir que la comunicación para mantener la coherencia de cache se haga punto a punto entre los procesadores que tengan copia de datos. Estos sistemas de los que estamos hablando son sistemas donde accesos a la memoria pueden tardar diferentes tiempos, es decir, pueden ser no uniformes. De ahí que se hable de NUMA: Non Uniform Memory Architectures.

Slide 32

Esta transparencia muestra el esquemático de un sistema NUMA. Cada procesador se conecta a un «trozo» de memoria por medio de un HUB. De forma generalizada podríamos tener varios cores (o multicores) conectados a un HUB por medio de un bus. Este hub se conecta a la memoria local y a otros hub (Dibujos de ejemplos de esto último en la Slide 33 y 38).

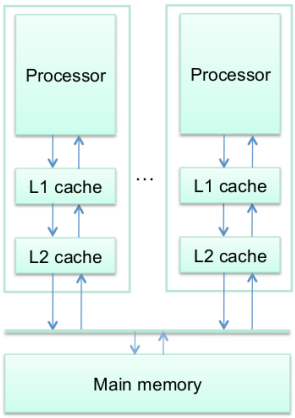
Los hubs permiten redirigir las transacciones para mantener la coherencia de cache, además de transferir los datos de memoria.

Normalmente a todo el sistema, con los diferentes hubs, procesadores y bancos de memoria se les da el nombre de una arquitectura NUMA. En boada tenemos 5 nodos de este tipo de arquitectura. Cada nodo boada consta de dos NUMA nodes (recordad el lstopo que hicistes). Se le llama NUMAnode a cada pack formado por un hub, el banco de memoria conectado a este hub, y el core/cores/multicores con o sin bus conectados a ese hub.

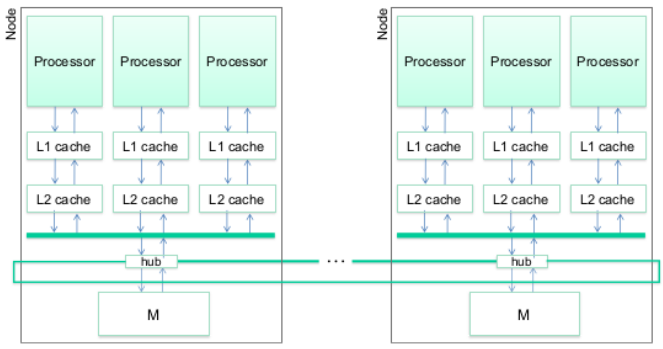
Extensión de las transparencias. Esta parte se ha realizado para ayudar a la comprensión de cómo se guarda la información de coherencia en sistemas UMA y en sistemas NUMA.

Esquema general:

UMA:



NUMA:



UMA:

¿Cómo se organiza la memoria y donde se ponen los datos?

- La memoria está centralizada. Un único espacio de direcciones. Todos acceden via un bus con el mismo tiempo de acceso.
- La memoria está organizada en bloques de memoria (líneas).
- Cuando un programa accede a una línea de memoria se traerá una copia de la línea a su cache y la mantendrá mientras que no la tenga que reemplazar o no se la invaliden. Por lo tanto, los procesadores pueden tener copias de líneas de memoria en sus caches.

¿Cómo mantienen la coherencia entre las caches?

- Broadcast en el bus + Snoopy
- Usando políticas de escritura Write-Update o Write-Invalidate. En adelante nos centraremos en Write-Ivalidate

¿Qué información se necesita para mantener la coherencia de cache?

- Información de Coherencia a nivel de cada Cache:
  - Cada cache guarda información de coherencia para cada una de sus lineas
  - Cada línea puede tener uno de los siguientes estados: M(odified),S(hared), I(invalid)
  - El número de bits que necesitamos en cada línia para guardar el estado son 2 bits (con dos bits podemos guardar 4 posibles.
- La memoria no mantiene información de coherencia. Tampoco tiene información de qué procesadores tienen copia ya que 1) todos los procesadores que pueden contener copia estan conectados en el mismo bus y 2) las peticiones de lecturas y escrituras les llegan a todos y por consiguiente cada uno de los procesadores puede actuar adecuadamente para mantener coherente su cache.

¿Qué pasa cuando un programa pregunta donde está un dato?

- Si un programa accede a un dato, donde estará el dato?
  - O bien está en memoria principal o bien está en la cache.
  - Si está en la cache puede estar en dos estados diferentes: S o M. Si una entrada de la cache está inválida no sabemos que dato estaba allí.

Aquí tenemos un ejemplo de una arquitectura UMA con tres procesadores con su nivel de cache (y los bits de state per indicar cómo se cuentran cada entrada – línia):

En la figura tenemos un dato que está compiado y modificado en una línia de cache del procesador 1. En ese caso, el dato y línea de memoria principal donde se encuentra no es válido. Las otras caches tienen indicado invalid para indicar que sus copias fueron invalidadas por la escritura en el procesador 1.

NUMA:

¿Cómo se organiza la memoria y donde se ponen los datos?

- La memoria está físicamente distribuida. Hay también un único espacio de direcciones. Ahora cada sub-sistema (NUMANode en la figura) tiene una parte de memoria accesible. Todos acceden via un bus al mismo hub pero no a resto del sistema NUMA.
- La memoria está organizada en bloques de memoria (líneas), igual que antes, pero la memoria total es la suma de partes que estan físicamente distribuidas en los diferentes NUMANodes.
- Cuando un thread accede a una línea de memoria se traerá una copia de la línea a su cache y la mantendrá mientras que no la tenga que reemplazar o no se la invaliden. Por lo tanto, los procesadores pueden tener copias de líneas de memoria en sus caches.
- **¿Dónde se encuentra un dato en un sistema con memoria distribuida físicamente?** Quizás el dato no se encontrará en la parte de memoria del NUMAnode donde el thread se está ejecutando y tendrá que pedirla a otro NUMANode – El NUMANode home del dato-línea de memoria donde se encuentra.
- **¿Cómo sabe donde esta guardado un dato el programa si no está en su parte de memoria más cercana?** El Sistema Operativo sabe donde se guarda cualquier dirección de memoria y puede hacer la petición del dato. De hecho, la primera vez que se accede un dato en un programa por un threaad, el Sistema Operativo decide donde colocar el dato. En una política first-touch (el primero que lo toca), el sistema operativo decide ponerlo en la memoria del NUMAnode donde se está ejecutando el thread que accedió al dato.

¿Cómo mantienen la coherencia entre las caches y cómo se sabe donde estan las copias de una línea de memoria?

- Hay un directorio distribuido para gestionar la coherencia de cache **entre** NUMAnodes, básicamente para saber si hay copias de las líneas de memoria principal, si las hay de qué tipo (compartida o modificada), y quien tiene las copias (para poder gestionar la coherencia de las copias en cache que puedan haber en otros NUMAnodes).
- **Dentro** de cada NUMAnode podemos mantener el mismo sistema de snoopy para ver que pasa por el bus hasta el HUB y cada cache mantiene el estado de sus líneas.
- Usando políticas de escritura Write-Update o Write-Invalidate. En adelante nos centraremos en Write-Ivalidate

¿Qué información se necesita para mantener la coherencia de cache?

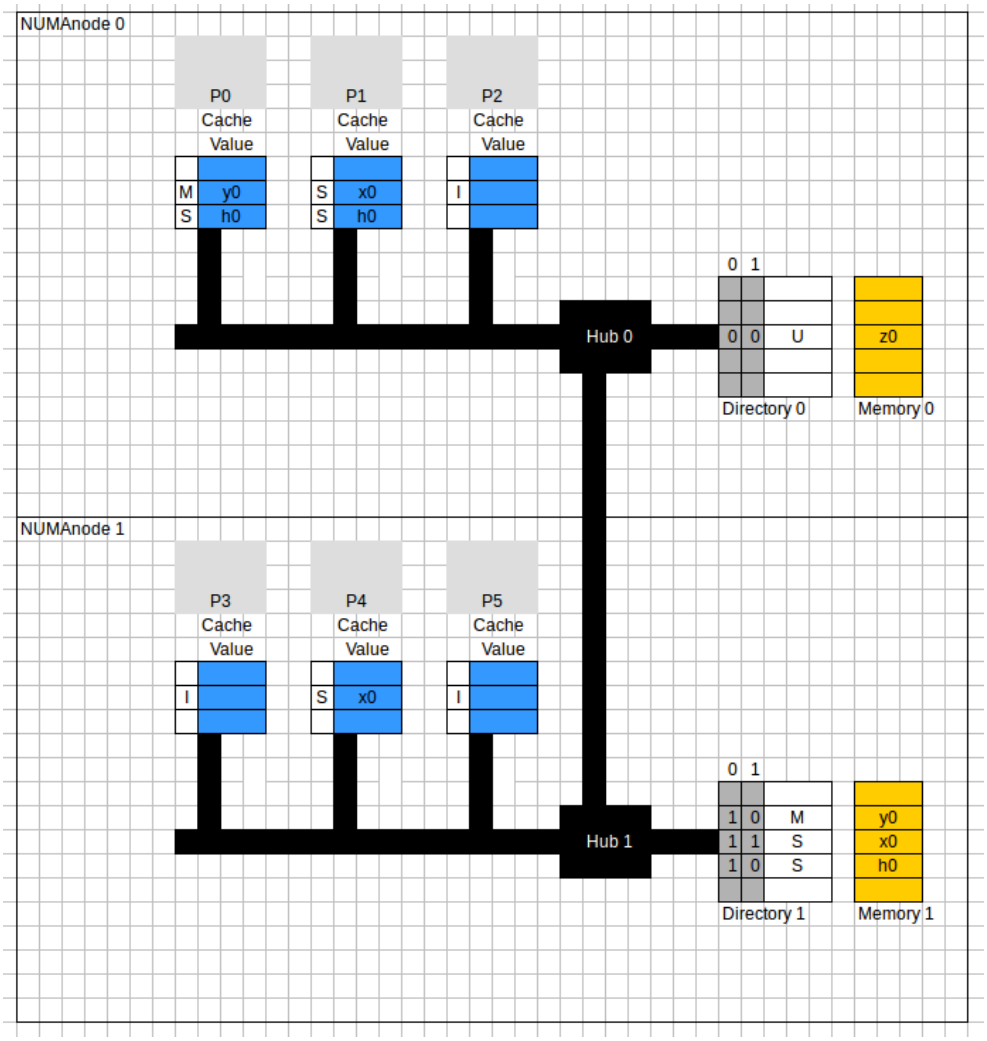
- Directorio: este guarda para cada línea de memoria:
  - Bits de status de si hay copia o no de esta línea de memoria en las caches de algun NUMAnode y de qué tipo son las copias.
    - Si no hay copia alguna se dice que la línea de memoria está Uncached.
    - Y si tiene copia si está shared o modified.
    - Hay tres possibles estados, por consiguiente, con 2 bits ya es suficiente.
  - Quien la tiene la/s copia/s?
    - Bits de presencia:

        - Para guardar esta información tenemos un vector de bits donde cada posición del vector guarda la información de si hay copia (1) o no (0) en el NUMANode que corresponda a esa posición. NUMANode i le corresponde posición i del vector.
        - Tendremos tantos bits como NUMANodes.
    - El directorio gestiona los accesos a las líneas de memoria, centralizando y ordenando las escrituras a una misma posición de memoria.
- Información de Coherencia a nivel de cada Cache:
    - Cada cache guarda información de coherencia para cada una de sus lineas
    - Cada línea puede tener uno de los siguientes estados: M(odified),S(hared), I(invalid)
    - El número de bits que necesitamos en cada línia para guardar el estado son 2 bits (con dos bits podemos guardar 4 posibles.
- La memoria mantiene información de coherencia gracias al directorio, qué guarda información de si hay copias o no y, en caso de haberlas, donde.

¿Qué pasa cuando un programa pregunta donde está un dato?

- Si un programa accede a un dato, donde estará el dato?
    - O bien está en memoria principal o bien está en la cache.
    - Si está en la cache puede estar en dos estados diferentes: S o M. Si una entrada de la cache está inválida no sabemos que dato estaba allí.
- Si no está en cache, donde voy a buscarla?
    - El Sistema Operativo le indicará en que NUMANode se guardó el dato en memoria (le NUMANode home de ese dato).
    - Se hará una petición a ese NUMANode de forma directa para que le pueda dar una copia de la línea de memoria y tambien para informarle que esta cache tendrá una copia.
    - Si la petición que se hace al NUMANode home es de escritura o actualización, el **NUMANode HOME** puede usar la información del vector de bits de presencia para enviar mensajes de invalidación a todos aquellos NUMANodes que tenían una copia.

Aquí tenemos un ejemplo de una arquitectura NUMA con dos NUMAnodes:



Mostramos:

- Un NUMA Node home (NUMAnode 1) de los datos x0, y0 y h0.
- Otro NUMA node home (NUMAnode 0) para el dato z0.

¿Qué observamos?

- Y0 (la línea donde está) tiene una copia modificada en el NUMANode 0. Eso lo sabemos gracias al directorio que indica M y en los bits de presencia hay un 1 en el bit correspondiente al NUMANode 0. También sabemos que dentro del NUMANode 0 sólo puede haber una cache con una copia modificada.
- x0 (la línea de memoria donde está) tiene dos NUMANodes con copias shared (límpias), una en cada NUMANode. Eso lo sabemos porque su entrada en el directorio tiene una S y los dos bits de presencias están a 1. Dendtro de cada NUMANode con copia puede haber una o más copias de la misma línea. Pasar al siguiente ejemplo.
- h0 (la línea de memoria donde está) tiene un NUMAnode que tiene copia límpia. Cuidado! Aquí vemos que el NumaNode 0 tiene copia gracias a que el bit de presencia tiene un 1, pero dentro del NUMAnode 0 hay dos copias límpias en dos caches diferentes. Esto es para que os déis cuenta que la información de presencia es por NUMAnode y no por core/cache. Sabemos que hay copia en el NUMANode 0 de h0, pero dentro de ese NUMANode puede haber más de una copia en más de una cache.
  - Eso es un problema? No, lo que nos interesa saber es que NUMAnodes tienen copias, dentro de cada NUMANode, ya se gestionaran las invalidaciones cuando sea necesario.
- Z0 (la línia de memora donde está) tiene su NUMANode home en el NUMANode 0, pero no hay ninguna copia, por eso está uncached.

Nomenclatura: Los NUMANodes que tienen copias de un dato se les llama NUMAnodes remotes (Owner en el caso de la copia modificada y Reader en el caso de la copia limpia). El NUMANode que empieza un acceso se le llama NUMANode local.

Seguimos con las slides.

Slide 33-34

En estos sistemas hay varios actores que participan en los mensajes de coherencia. Estos actores tienen que ver con los Numa nodes.

El numa node donde se guardan los datos en memoria la primera vez que se acceden se le llama Numa node home. ¿Dónde se escribe este dato? Normalmente la política que se utiliza es first-touch, en la que el sistema operativo coloca la memoria asociado a un dato accedido por primera vez en una localización cercana al thread que la accedió.

Los otros dos actores que intervienen en el mantenimiento de la cache coherence son los nodos «Remote» y «Local» Numa Nodes. Remote Numa node es aquel que contiene alguna copia de la línea de memoria accedida y puede ser Owner (contienen copia modificada) o Reader (contiene una copia limpia de la línea de memoria accedida). Local Numa node es el que efectua el acceso a la línea de memoria.

Como podéis ver pueden haber hasta tres tipos de Numa Nodes que pueden intervenir en el mantenimiento de la coherencia de cache.

Para mantener el sistema de coherencia en este caso se suele usar un sistema de directorios ya que escalar un sistema de coherencia basado en broadcast es complicado (Slide 34). **Sin embargo, en un sistema basado en broadcast por el bus, el bus podia controlar el orden de las operaciones de actualización y modificación de alguna dirección de memoria. Era en el momento de petición de acceso al bus el que permitía garantizar un orden: si había algun procesador realizando alguna acción de actualización sobre una dirección de memoria, el resto de procesadores no podían actuar sobre la misma dirección. Como se garantizará ese orden de acceso usando directorios?**
Primero recordemos que el directorio contiene información de coherencia de las líneas de memoria de una zona de memoria principal (NO tiene información de coherencia DE LAS CACHES). Cada NUMAnode home contiene la información de coherencia correspondiente a las líneas de memoria que se guarden en su zona de memoria local (decidido, por ejemplo, por el protocolo first touch del sistema operativo). Recordado esto, el directorio, o mejor dicho, la parte del directorio que es home Numa node de una dirección de memoria garantizará el orden para las escrituras que van a la misma dirección de memoria. ¿Cómo puede conseguirlo? Esto se consigue porque cualquier acceso de actualización (UpgrReq) o de escritura (WrReq) que se origine sobre una dirección de memoria que el directorio guarda, originada la petición en cualquier punto del sistema, llegará a este directorio para actualizar el estado de esa línia de memoria principal. Por consiguiente, como los NUDAnode home centralizan en su directorio las peticiones de actualización/escritura sobre las líneas de memoria que gestiona, puede determinar el orden de éstas. Gracias a que el directorio está distribuido entre las diferentes porciones de la memoria compartida, pero físicamente distribuida, este control del orden es por porciones de memoria, evitando excesiva contención sobre un único directorio.

El sistema de directorio guarda el estado de coherencia por línea de memoria principal. Cuidado!!! esta información SÓLO está en el Home Numa Node donde se guarda por primera vez la línea de memoria. Así, cuando un Local Node quiere realizar un acceso, SÓLO tendrá que comunicarse con el Home Numa Node. Si hay más de un Numa Node haciendo una petición al Home Numa Node , este garantizará el orden. Para mantener la coherencia habrá comunicaciones punto a punto entre los diferentes Numa Nodes, evitando realizar boradcast a todo un sistema con mucho procesadores. Por otro lado, las caches de cada procesador continuan teniendo información del estado de coherencia de sus líneas de cache. Lo único que ahora la gestión global del orden de las escrituras queda más distribuido ya que no se hace un broadcast a todo el bus para que todos los procesadores del sistema se enteren.

Slide 35

Qué es lo que se guarda en un directorio para dar soporte a la coherencia de cache? Y donde se guarda?

Hay un trozo de directorio para cada porción de memoria. Es decir, una porción del directorio por Numa Node. El directorio contiene una entrada por cada línea de memoria principal. Cada entrada del directorio guarda información del estado de las copias existentes de esa línea de memoria principal: estado de coherencia y quienes tienen copia. La información de quienes tienen copia puede guardarse de distintas formas. La que os presentamos consiste en un bit de información para cada Numa Node. Normalmente se acaba teniendo un vector de bits con un 1 o un 0 para indicar si hay copia o no en cada Numa Node. Los estados de los que disponemos en las transparencias son M (Modified) , S (Shared), U (uncached), y por consiguiente sólo necesitamos 2 bits. En el video sólo hablaba de un bit de dirty y los bits de presencia.

- **Si el estado de la línea de memoria es M en el directorio, eso significa que sólo hay una copia válida de la línea de memoria en alguna cache del sistema. Esta cache que contiene una copia modificada estará dentro de algun procesador de algun NUMAnode. En los bits de presencia SÓLO habrá un bit a uno, y nos indicará en qué NUMAnode se encuentra la copia. CUIDADO! Te indica en qué NUMAnode, pero no en que cache! Recordad el ejemplo de arriba.**
  La cache que tenga esa copia tendrá el estado M si es que se sigue un protocolo MSI dentro de cada NUMAnode.
- **Si el estado de la línia de memoria es S (Shared), eso significa que hay una o más caches con copias limpias (no modificadas) de esa línea de memoria. Pero cuidado! En los bits de presencia del directorio se tiene información por NUMAnode. Por consiguiente, tendremos tantos bits a unos como NUMAnodes con alguna copia limpia de la línea de memoria; en alguna de las caches de los procesadores en aquellos NUMAnodes. Dicho de otro modo, ¿Podemos tener más copias limpias de la línea de memoria que bits a unos del vector de presencia? SI! Por ejemplo, podríamos tener dos copias limpias en dos caches de en un mismo NUMAnode. Sólo tendríamos el bit a 1 para ese NUMAnode pero en cambio tenemos dos copias. Cada una de las copias tendrá el estado de coherencia S también, si seguimos el protocolo MSI dentro de cada NUMAnode.**
- **Si el estado en una entrada del directorio es U, esto significa que la dirección de memoria que guarda esa entrada NO TIENE NINGUNA COPIA EN LAS CACHES. Por consiguiente, en los bits de presencia de la entrada del directorio no habrá ningún bit a 1. Todos seran ceros. En este caso, como no hay copias en caches, no aplica hablar de cómo esta el estado de la copia en cache :-) .**

Finalmente, intentemos aclarar que el estado de coherencia en las caches, siguiendo el protocolo MSI, guarda la información de esa copia local. Este estado puede coincidir con el estado guardado en la entrada del directorio : M y S. Pero, por otro lado, podemos tener el estado I en las caches, pero no en el directorio. Al igual pasa que en el directorio podemos tener U y este estado no tiene sentido en las caches. El I en una copia en cache nos indica que la copia que tenemos en la cache no es válida. Esto puede pasar porque nunca se leyó nada, o porque se invalidó siguiendo el protocolo de coherencia MSI.
En cambio, es el SO el que decide qué se guarda y qué es válido en las memorias principales. En el directorio de memoria tenemos información válida para las líneas de memoria guardadas en esa parte de la memoria. Por otro lado, para saber si tenemos copias o no de esa línea de memoria que se guarda en el directorio, necesitamos del estado U para indicar que una línea de memoria no tiene copias en las caches.

Slide 36

Para mantener la porción del directorio que está en un Numa Node, éste recibe peticiones de los diferentes Numa Nodes (NUMANode Local según la nomenclatura comentada antes). Estas peticiones pueden ser de lectura (RqReq), de lectura con intención de escribir (WrReq) y de actualización de una copia ya existente (UpgrReq) .

Para el caso de RqReq y WrReq el home node acabará enviando una transacción Dreply con la línea de memoria limpia. Es posible que antes de poder hacer Dreply del dato tenga que pedirlo (transacción Fetch) a otro Numa Node (Remote-Owner) si el estado para esa línea de memoria en el directorio era M.

Para el caso de UpgrReq sólo envia un Ack al local numa node (numa node que hizo la petición) . En este caso, al igual que para el WrReq, el home numa node puede tener que enviar mensajes de invalidación a los Numa Nodes que tengan copias de la línea de memoria afectada. Los Remote Numa nodes (Readers) contestarán a las invalidaciones con ACKs.

A modo de ejemplo mirar el dibujo de la slide 37:

Para seguirla un poco pensad que el home nuda nome es el del procesador 2. Hay dos copias de la línea accedida: una en la cache del procesador 2 que está en el Home Numa Node y otra en la del procesador 3 (uno de los dos Remote-Reader Numa Nodes). La coherencia en cache se tiene que mantener dentro de cada Numa Node, por ejemplo, siguiendo el protocolo visto MSI. De ahí que estas caches tengan bits de estado de coherencia y en el momento de iniciar la petición, las dos copias estaban en S (shared).

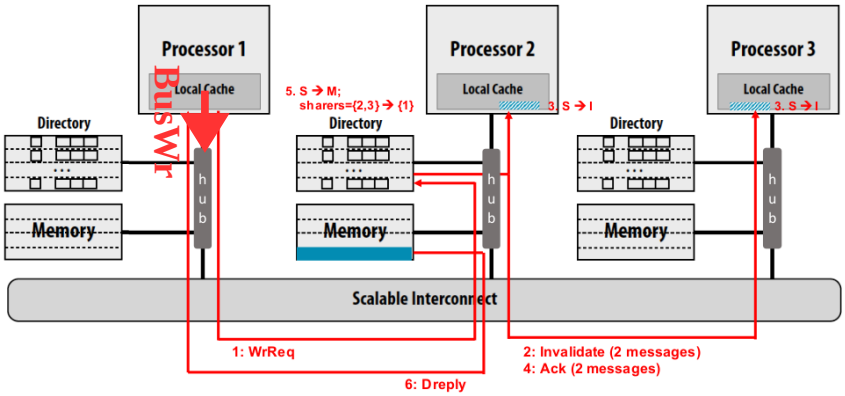
Al principio, el directorio guarda la información de que la línea está en estado S (shared) y que la lista de NumaNodes que tienen una copia es {2,3}, que hacen referencia a los numa nodes donde estan los procesadores 2 y 3 respectivamente.

A partir de aquí empieza el baile con 6 acciones importantes. Todas ellas estan numeradas e indican el orden en el que se realizan todas las acciones en esta arquitectura NUMA con tres Numa nodes.

A continuación explicamos un poco cada una de las acciones:

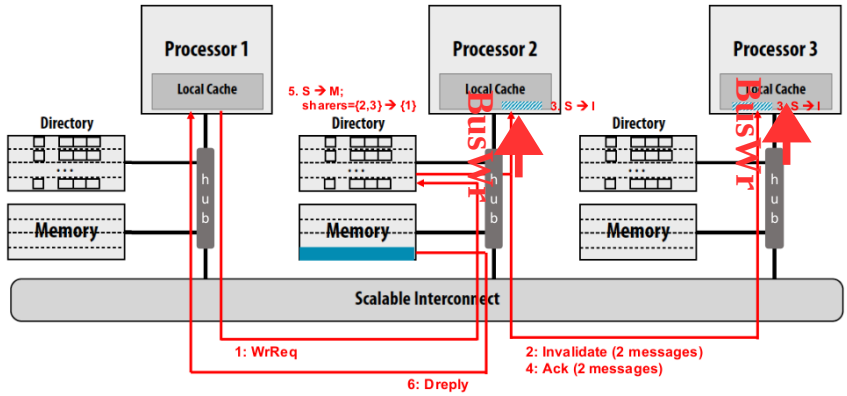
1 – Local Numa Node es el que inicia la petición de lectura con intención de escritura. En este caso es el Numa node con procesador 1. El procesador 1 hizo un write miss en su cache ya que no tiene copia. **Este envió un BusWr que le llegó al hub de este Numa Node, el cual no tiene esta dirección y por consiguiente consulta con el OS quien tiene esa dirección física de memoria**

accedida. El SO sabe cuál es el home Numa node, y el hub del Numa node 1 redirecciona la petición enviando un mensaje de WrReq al Numa node 2 para que le de una copia limpia de la línea de memoria y también para informarle que quiere escribir sobre ella. El Home Numa Node de esa línea de memoria (Numa Node 2), antes de poder hacer Dreply, debe invalidar cualquier copia que haya en el sistema NUMA.



2 – El Home Numa Node **(Numa node 2)** ve que tiene copias en dos Remote (Reader) Numa Nodes (Numa nodes 2 y 3) y les envia un mensaje de invalidación.

3 - Los dos Numa Nodes, localmente, hacen los pasos para indicar a todos los procesadores locales que tienen que invalidar las copias que dispongan. **Eso es fácil de hacer si los hubs de estos dos Numa Nodes envian un BusWr o BusUpgr a través del bus que comparten con los procesadores dentro de sus Numa Node. Todos los procesadores al ver ese BusWr invalidarán sus copias en sus caches.**



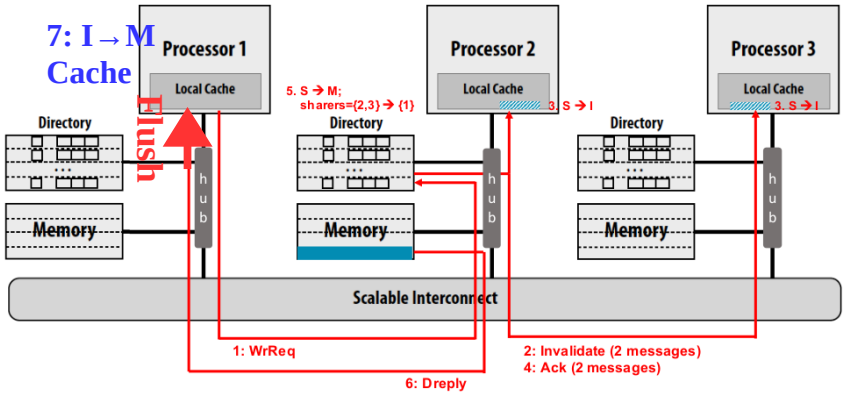
4 – Una vez los Remote Numa nodes han invalidado todas las copias existentes en su Numa Node, pueden enviar un ACK al Home Numa Node.

5 – El Home Numa Node cuando recibe todos los ACKs pasa el status de la línea de memoria principal en el directorio de Shared a Modified. Además actualiza la lista de shaders poniendo que sólo el Numa Node con el procesador 1 tiene copia (pone un 1 en el bit del Numa Node 1 y 0 en el resto)

6 – Finalmente, el Home Numa Node contesta al Local Numa Node con la copia de la línia de memoria que había pedido **(Dreply). El local Numa node (Numa Node con procesador 1) se convierte en el Numa Node Owner de esa línia de memoria.**

7 – **El hub del Numa Node Owner (Numa Node 1) transforma ese Dreply en un Flush que envía al bus que va al procesador 1 para que la cache de este procesador pueda actualizar el**

dato, y ponga su línea de cache en estado Modified. Nota: dentro de los Numa nodes de este ejemplo sólo hay un Procesador conectado al bus, pero podríamos tener sistemas con más de un procesador escuchando en el bus, y por consiguiente, todas las caches que tuvieran copias tendrían que actuar para mantener la coherencia de esas copias.



Slide 38

Ejemplo de un sistema donde se ven las estructuras hardware para dar soporte a la coherencia de cache tanto en los directorios adosados a las memorias locales, como en las caches de cada procesador dentro de los Numa Nodes.

Hay tres Numa Nodes y 2 procesadores por Numa Node. Hay un directorio para cada Numa Node y porción de memoria principal (main memory – MM). Cada directorio contiene unos bits de estado y 3 bits de presencia : uno para cada Numa Node que hay. Cada cache también tiene bits de estado de coherencia. Los tres Numa Nodes se comunican con una red de interconnexión no definida.

Problema 6: a, b, c y d (hecho en clase)

Próximo dia seguiremos con sincronización. Seria bueno que intentárais hacer los quiz.