Guion del grupo T20-Clase 27-04-2020

Disclamer: Aprovecharé parte del guión del grupo 40 ya que esencialmente hemos explicado lo mismo en la parte de introducción. También aprovecharé parte del guion del grupo 10, porque también hay un parte que hemos explicado muy parecido. Lo digo por si pasáis el antiplagio :-) . Habrá una parte en la que nosotros hicimos un ejemplo más sencillo para el tema de coherence problem, pero también os añadiré un ejemplo más completo (si no lo entendéis ya lo repasamos en clase) donde se trabaja con más conceptos a la vez. He intentado describir en más detalle las tablas que aparecen para que podáis entender qué se está haciendo.

Primera parte:

L'objectiu del Tema 4 és entendre els fonaments de les arquitectures multiprocessador de memòria compartida, que donin suport a l'execució d'un model de programació de memòria compartida tipus OpenMP; això implica dues coses: 1) que la memòria és el component que permet als processadors compartir dades (variables escalars i estructurades); i 2) que han d'existir mecanismes que permetin sincronitzar aquesta compartició de dades per tal d'evitar "data races". Aquests dos aspectes són els que estudiarem en aquest tema.

Slide 6

No podem començar a parlar d'arquitectures multiprocessador sense abans repassar l'arquitectura uni-processador que ja coneixeu. En aquesta slide a la dreta teniu un diagrama en què es mostra un processador i la seva jerarquia de memòria. Si recordeu de l'assignatura d'AC tant en el processador com en la jerarquia de memòria els arquitectes de computadors han fet tot el possible per millorar el seu rendiment, reduint el temps de cada tipus d'operació.

Slide 4

Per exemple pel que fa a processador, passant de l'execució lineal (escalar) en la que una nova instrucció no comença fins que l'anterior ha acabat, a una execució segmentada en la qual les instruccions superposen la seva execució en base a dividir la seva execució en etapes; en acabar la primera etapa d'una instrucció ja es pot començar l'execució de la següent instrucció, aconseguint així un primer grau de paral·lelisme en l'execució d'instruccions. Si cada etapa triga un cicle d'execució del processador, s'aconsegueix executar 1 instrucció per cicle. I anant un pas més enllà, a l'execució superescalar en la que a cada cicle es pot iniciar l'execució de múltiples instruccions independents. En aquesta slide, amb 8 instruccions executades simultàniament.

Tornant a la slide 6

També d'EC i AC podeu intentar recordar: 1) el perquè de la jerarquia de memòria, motivada per allò del "gap" creixent entre la velocitat del processador i de la memòria que es mostra en la gràfica de l'esquerra; 2) el perquè de tenir un o més d'un nivell de memòria cache, segurament relacionat amb el fet que el temps d'accés a una memòria està relacionat amb la seva capacitat (nombre de paraules de memòria); i 3) el perquè de les caches d'instruccions i dades separades en el primer nivell de la jerarquia, relacionat amb els tipus d'accés que es

fan.

Slide 7

I per què funciona la jerarquia de memòria? Quins són els dos principis en què es basa el seu funcionament? Els principis de localitat temporal (la probabilitat de tornar a referenciar l'adreça actual és molt alta) i espacial (la probabilitat de referenciar adreces properes a l'actual és molt alta). La localitat temporal ens suggereix que val la pena llavors guardar el contingut de l'adreça actual en una memòria més propera amb accés més ràpid doncs benaviat es tornarà a utilitzar; la localitat espacial ens suggereix que quan accedim a una paraula aprofitem i accedim a un bloc d'adreces consecutives, el que s'anomena una línia (unitat de transferència entre nivells de la jerarquia). A l'accedir a un nivell de la jerarquia podem trobar l'adreça que volem accedir (hit), o no trobar-la (miss) i llavors ens toca anar-la a buscar a un nivell més llunyà.

Slide 8

I per acabar aquest recordatori, els 4 algorismes o polítiques que defineixen el funcionament d'un nivell de la jerarquia de memòria: política d'emplaçament (una línia de memòria principal (MP) en quina línia de la cache es pot guardar?: directe, totalment associatiu o associatiu per conjunts), política de reemplaçament (si la cache està plena, quina línia trec de la cache per desar la nova?: FIFO, LRU, ...), polítiques d'escriptura en cas de hit (write-through o copy-back) i en cas de miss (write-allocate i write-no-allocate). Després farem un exemple on recordarem algunes d'aquestes polítiques de funcionament.

Slide 11

Passem a les arquitectures multiprocessador. En aquest curs parlarem de tres tipus d'arquitectures multiprocessador, un d'ells amb memòria compartida centralitzada (que anomenarem SMP o UMA, amb el que començarem avui) i dos d'ells amb memòria distribuïda, un dels quals ens proporcionarà memòria compartida (que anomenarem NUMA i que veurem el dia vinent) i un altre que no ens servirà per executar models de memòria compartida tipus OpenMP (i que s'anomenen multiprocessadors amb pas de missatges, arquitectures clúster o multicomputadors, que si tenim temps comentarem en el tema següent).

Slide 12

Comencem així amb les arquitectures SMP (Symmetric Multi-Processor) que us comentaven en el vídeo d'aquesta setmana. Bàsicament 2 o més processadors idèntics connectats a una memòria centralitzada a través d'una xarxa d'interconnexió (anomenada bus, tothom hi està connectat).

Slide 13

Que permet la xarxa d'interconnexió? Doncs que qualsevol processador pugui accedir a qualsevol adreça de memòria. Com? Amb les instruccions normals de load (ld: lectura d'una adreça de memòria) i store (st: escriptura a una adreça memòria). Un altra propietat de les arquitectures SMP és que el temps d'accés a memòria principal és constant, és a dir, l'accés qualsevol adreça de memòria principal sempre és el mateix, independentment de l'adreça i del processador que realitza l'accés. Per això aquestes arquitectures també

s’anomenen UMA (Uniform Memory Access).
La principal limitació de les arquitectures UMA és el nombre de processadors que directament es poden connectar a una única memòria principal. El temps a memòria és constant sempre que no hi hagi conflictes en l’accés a la xarxa d’interconnexió, és a dir, que quan vulgui accedir-hi no hi hagi cap altre processador accedint-hi.

Slide 14

Dues solucions per disminuir la possibilitat de conflictes: 1) afegir memòria cache, que farà que només hàgem d'anar a memòria principal quan fem un miss a la cache (cosa que per localitat hauríem de fer poc); i 2) tenir múltiples bancs independents de memòria que es puguin accedir simultàniament a través de la xarxa d’interconnexió, servint així les línies de cache que necessiten processadors diferents.
De fet podríem haver ficat la cache de dues maneres possibles: cache privada (tal com mostra el dibuix de la slide 15, cada processador té la seva pròpia jerarquia de cache) o cache compartida (en la que tots els processadors comparteixen la mateixa jerarquia de cache). En el vídeo que heu vist aquesta setmana comenta el perquè és millor l'opció de caches privades, ja que redueixen el temps d’accés i eviten els conflictes en l’accés a la xarxa d’interconnexió. Però ... quin problema ens presenta l’arquitectura UMA amb caches locals? El problema de la coherència de memòria.

Slide 15

Ens mostra el problema de la coherència de memòria. Tres processadors que accedeixen a la variable foo en l’adreça de memòria X, tant per llegir com escriure. Cada lectura, per part d’un dels processadors, crea una nova copia en la seva cache privada. Si qualsevol d’ells fa una escriptura, com que l’accés és un hit (i assumim copy-back), la memòria principal no s’actualitza, provocant una incoherència de memòria. Si després un altre processador torna a llegir, llegirà de memòria un valor obsolet. Amb el copy-back recordeu que la memòria principal només s’actualitza quan la línia es reemplaça de la cache, com passa per exemple en aquest cas quan el primer processador elimina la variable que està a l’adreça X de la seva cache per donar cabuda a una variable en l’adreça Y.

En clase hemos tratado con el siguiente ejemplo:

Ejemplo:
int x=5, y1=0; // Inicialización en memoria.

```
#pragma parallel num_threads(2)
#pragma omp single
{
    x = x+3; //(1)

    #pragma omp task shared(x,y1)
    {
        int z1;
        z1 = (x*25);    // qué valor de x lee, 5 o 8?
        x = z1;         // el valor de x lo verá el thread máster?
        y1= y1+1;       // (3) el valor de y1 lo verá el thread máster?
     }
     while (y1==0);     // (2) verá el thread máster un cambio en el valor?
     x = x*4;           // qué valor tendrá x?

}
```

Es un programa sencillo que habre una región paralela con dos threads. Como trabajamos con tasks, hemos decido hacer un single. En este caso vimos que el thread master (single hace que sólo comience un thread del pool de threads) empezará a ejecutar. Suposimos que habían dos threads: el thread máster y otro más, y que cada thread se ejecutaba en un procesador, cada uno con sus caches privadas. Además, para este ejemplo, no había ningún tipo de soporte de coherencia de cache todavía!.

Bien, el thread máster será el primero que leerá el valor de x (y toda la línea de cache donde está) en la instrucción del código (1) y actualizará su valor (en su cache!!!) . Eso implicará que el valor de x será 8 en la cache del procesador donde está el thread máster. En la memoria, en cambio, continuará siendo 5, que es como se había inicializado la variable global. Notad que x y y1 son globales, y además se ha hecho explícito que son compartidas en la task que crea el master thread.

El thread máster creará la task y ya sabemos que la ejecución puede hacerse cuando el thread (el idle normalmente, y en este caso es el otro thread- thread 1) se de cuenta que hay algo a hacer y empiece a ejecutar la tarea.

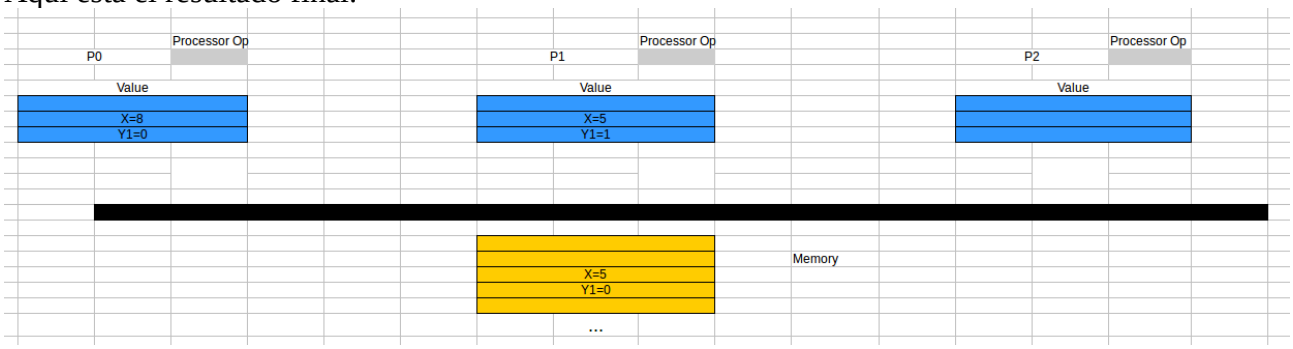
Puede pasar que antes que thread 1 empiece con la tarea, el thread máster llegue al while, y se quede allá mientras que y1 valga 0 (línia de código (2)).Se supone que cuando el thread 1, cuando ejecute la tarea pondrá ese valor y1 a 1 (línea de código (3))... però ... NO!!!

Sin soporte de coherencia lo que va a pasar es que el thread máster leerá y1 (y la línea de memoria donde está), la traerá a su cache en la primera lectura, y después, cada vez que quiera consultar el valor de y1 en el bucle while producirá un hit en su cache (la y1 está en su cache) y no necesitará ir a memoria principal. ¿Qué es lo que pasará entonces? Que no verá ninguna actualización del valor por parte del thread1????

Veamos, que pasa. El thread 1 comienza a ejecutar la tarea (task) y comienza a realizar las operaciones. En clase fuimos contestando las preguntas... qué valor de x (y línea de cache) leerá? Pues leerá la línea que se encuentra en memoria principal, cuyo valor sigue siendo 5. AI!!!! se dará cuenta el otro thread (el máster) que alguien va a leer x y que le tiene que pasar? NO!!! bueno... no, si no hay ningún soporte hardware que nos garantice una coherencia entre las copias que puede haber de las líneas de memoria en las diferentes caches. Lo que pasará si no hay coherencia es que el thread 1 tendrá x=5 en su cache, mientras que el thread máster ya tiene un valor de 8. Son incoherentes!

Pero lo peor no es eso, lo peor es que esta incoherencia hará que el programa tampoco acabe. El thread 1 leerá y1 (y la línea donde se encuentra) y se la copiará a su cache. Cuando le añada un valor a 1 **(línea de código (3))**, ese valor se actualizará.... dónde? En SU CACHE!!!! por consiguiente, el thread máster NUNCA verá que y1 pasa a 1... y se quedará infinitamente esperando a que se produzca el cambio. Sólo cuando el thread 1 tenga que reemplazar la línea de cache donde está y1, y haga copy-back en memoria principal... el thread máster se dará cuenta de esa actualización, pero no es seguro.

Os he colgado la hoja libreoffice para poder ir viendo como se actualizaban los valores de x e y1. Aquí está el resultado final:



Este es el problema de coherencia de caches. El procesador tiene que poder indicar a la cache que hacer, y por otro lado escuchar lo que hacen los demás procesadores. Necesitamos soporte del hardware. Necesitamos un mecanismo para propagar los cambios de una cache al resto de caches.

Ejemplos de otras clases con mucho más detalles:

Ejemplo:

Supongamos el siguiente código:

```
int negatius=0;
...
#pragma omp taskloop grainsize(2)
for (i=0; i<12; i++) {
        if (a[i]<0) {
            #pragma omp atomic
            negatius += a[i];
        }
```

Que se ejecuta en dos procesadores P0 y P1, cada uno con una memoria cache de 3 líneas. Supongamos además que los elementos están almacenados en memoria como se muestra a continuación, así como la asignación de iteraciones a procesadores con la indicación de si el valor es positivo o negativo.

Aclaraciones del dibujo:

a[16] viene a indicar el vector y sus 16 posiciones... sólo se muestran las 12 primeras y el resto tienen valores negativos. Los «+» y «-» indican si el valor guardado en la posición 0, 1, 2,... es positivo o negativo.

Memory line (block) indica en qué linea de memoria principal se encuentra.

Task assigned.. significa a qué thread se le da esa tarea y supondremos que thread 0 va al procesador 0 y thread 1 al procesador 1.

La cache con la que trabajaremos es full associative (una línea de memoria puede ir cualquier entrada libre de la cache). La cache tiene 3 líneas de 16 bytes cada línea, total: 48 bytes de cache.

a[16]	0	1	2	3	4	5	6	7	8	9	10	11	negatius
	+	+	+	+	+	-	-	-	+	+	+	+	
memory line (block)	line 0				line1				line2				line3
task assigned to processor ...	1		0		0		1		0		1		

Cache size=48 bytes
Line size = 16 bytes
Fully associative placement, random replacement, copy-back, write-allocate

cache 0

line	data		D/V
0			
1			
2			

cache 1

line	data		D/V
0			
1			
2			

Si no se hace nada para mantener la coherencia de cache, y si analizaos una possible ejecución del programa, tendríamos el comportamiento siguiente:

Cada fila de la tabla indica la ejecución de una instrucción de nuestro bucle negatius+=a[i] por parte de un thread (0 o 1). El thread 0 o 1 lo sabemos gracias a la columna P, de procesador. Para cada fila observamos el elemento accedido del vector a (element), la línea de memoria que se trae (memory line), a qué cache de procesador va (0 o 1 en cache 0/1), la entrada-número de línea de la cache donde va (cache line), si se produce hit o miss (hit/miss) y el valor que tiene ese elemento en la memoria principal (value [MP]). Para la parte de la variable negatius (que se encuentra en la línea de memoria principal 3) tenemos esecialmente el mismo tipo de información.

P	a[16]						negatius (memory line 3)					
	element	memory line	cache 0/1	cache line	hit/miss	value [MP]	cache 0/1	cache line	hit/miss	value [0]	value [1]	value [MP]
1	0	0	1	0	m (r)	0						0
0	2	0	0	0	m (r)	2						0
1	1	0	1	0	h (r)	1						0
0	3	0	0	0	h (r)	3						0
0	4	1	0	1	m (r)	4						0
1	6	1	1	1	m (r)	-6	1	2	m (r) h (w)		0 -> -6	0
0	5	1	0	1	h (r)	-5	0	2	m (r) h (w)	0 -> -5		0
1	7	1	1	1	h (r)	-7	1	2	h (r) h (w)		-6 -> -13	0
1	10	2	1	2 (*)	m (r)	10	copy-back due to replacement of cache line 2 in processor 1					0 -> -13
1	11	2	1	2	m (r)	11						-13
0	8	2	0	0 (**)	h (r)	8						-13
0	9	2	0	0	h (r)	9						-13
				(*) random replacement selected line 2 in processor 1								
				(**) random replacement selected line 0 in processor 0								

Cuando aparece un valor→valor significa actualización de valor. Si hay un (*) o (**) significa una anotación de reemplazo.

Si vais mirando lo que pasa veréis que no es correcto ya que en caso de miss de lectura de negatius, el procesador siempre lee de memoria principal el mismo valor (valor 0) y en caso de hit de escritura no actualiza el contenido de la memoria; sólo se actualiza en caso de reemplazo que es cuando se realiza el write-back (o copy-back en el ejemplo).

Slide 16

Como vimos, hay dos mecanismos que se pueden utilizar para mantener la coherencia entre los procesadores.

Los métodos para propagar los cambios en una cache al resto los podemos clasificar en:

a) write-update: un procesador cada vez que modifica un valor de una variable en memoria compartida, de la que tiene copia en cache, lo comunica a TODOS los procesadores que comparten memoria y fuerza a que todos actualicen sus valores, si estos ya tenían una copia también en sus caches. Esta alternativa tiene sentido cuando desde un procesador se realiza una escritura y todos los siguientes accesos desde otros procesadores son únicamente para lectura. Pero como vimos en clase esto puede provocar mucho tráfico para mantener la coherencia de cache.

b) write-invalidate: un procesador cada vez que modifica un valor de una variable en memoria compartida, de la que tiene copia en cache, fuerza a que todos los procesadores que comparten memoria (con copia de una misma línea de memoria en sus caches) invaliden las copias de esta variable (y línea de cache). El nuevo valor es actualizado cuando se necesite. Cómo? Pues el procesador intentará acceder, tendrá un miss en cache y pedirá otra vez el dato a memoria principal.

En protocolos write-update, si un dato es leído y nunca más es utilizado, siguientes accesos a ese dato estarán continuamente actualizando su valor, aunque no se utilice, generando gasto innecesario de ancho de banda y latencias adicionales. Esta ineficiencia con protocolos write-invalidate no sucedería, dado que solo se invalida una vez. Hasta que no se vuelve a utilizar no se actualiza el valor en las caches.

Veamos cómo quedaría el ejemplo anterior si el comportamiento de las memorias fuera:

Con protocolo write-update vemos que tanto las caches como la memoria principal se van actualizando:

P	a[16]						negatius (memory line 3)					
	element	memory line	cache 0/1	cache line	hit/miss	value [MP]	cache 0/1	cache line	hit/miss	value [0]	value [1]	value [MP]
1	0	0	1	0	m (r)	0						0
0	2	0	0	0	m (r)	2						0
1	1	0	1	0	h (r)	1						0
0	3	0	0	0	h (r)	3						0
0	4	1	0	1	m (r)	4						0
1	6	1	1	1	m (r)	-6	1	2	m (r) h (w)		0 -> -6	**-6**
0	5	1	0	1	h (r)	-5	0	2	m (r) h (w)	-6 -> -11	**-11**	**-11**
1	7	1	1	1	h (r)	-7	1	2	h (r) h (w)	**-18**	-11 -> -18	**-18**
1	10	2	1	2 (*)	m (r)	10						-18
1	11	2	1	2	m (r)	11						-18
0	8	2	0	0 (**)	h (r)	8						-18
0	9	2	0	0	h (r)	9						-18

Con protocolo write-invalidate se van invalidando las copias que puedan tener las caches:

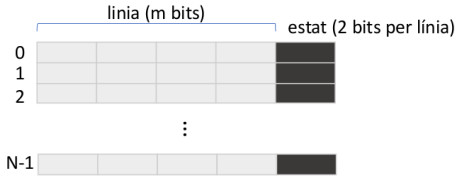
P	a[16]						negatius (memory line 3)					
	element	memory line	cache 0/1	cache line	hit/miss	value [MP]	cache 0/1	cache line	hit/miss	value [0]	value [1]	value [MP]
1	0	0	1	0	m (r)	0						0
0	2	0	0	0	m (r)	2						0
1	1	0	1	0	h (r)	1						0
0	3	0	0	0	h (r)	3						0
0	4	1	0	1	m (r)	4						0
1	6	1	1	1	m (r)	-6	1	2	m (r) h (w)		0 -> -6	**0 -> inval**
0	5	1	0	1	h (r)	-5	0	2	m (r) h (w)	-6 -> -11	**-6 -> inval**	inval
1	7	1	1	1	h (r)	-7	1	2	m (r) h (w)	**-11 -> inval**	-11 -> -18	inval
1	10	2	1	2 (*)	m (r)	10	copy-back (replacement of line 2 in P1)			inval	10	**inval ->18**
1	11	2	1	2	m (r)	11				inval	11	-18
0	8	2	0	0 (**)	h (r)	8				8	11	-18
0	9	2	0	0	h (r)	9				9	11	-18

Slide 17

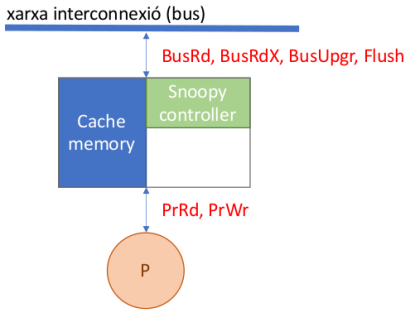
En ambdós casos la xarxa d'interconnexió tipus bus és el mecanisme que permet fer el broadcast d'aquestes comandes i estableix l'ordre en què les coses es veuen per tots els elements implicats. Que vol dir això? Que quan un processador vol fer un accés avisa a tots els altres i a la memòria (del update o del invalidate) i la resta de processadors i la memòria estan escoltant ("tafanejant", millor dit) contínuament el que es diu per la xarxa. Aquestes accions de comunicació pel bus les fa l'anomenat "snoopy cache controller" (snooping=tafanejar).
Anem a detallar el mecanisme de snoop amb write-invalidate, que és el més adient per ser equivalent al copy-back en un sistema uni-processador.

Slide 19

És un protocol distribuït en el sentit que cada cache guarda informació de l'estat de la seva línia (Shared S: tinc còpia de la línia, i potser hi ha més copies en altres caches; Modified M: només jo tinc còpia de la línia i està “dirty”, modificada; Invalid I: he tingut la línia però l’han invalidat en algun moment). Com que són 3 estats necessitem 2 bits per codificar-los. Així als m bits que ocupen les dades de cada línia li haurem d’afegir 2 bits per codificar l'estat.



I suposem les següents comandes que pot generar el processador cap a la memòria cache i el snoopy emetre/escoltar pel bus (totes elles explicades en la slide):



Para cada acceso a memoria, el procesador puede generar dos eventos: PrRd si se trata de lectura o PrWr si se trata de una escritura. El protocolo reacciona, escribiendo en el bus las transacciones: BusRd indicando lectura del dato, BusRdX / BusUpgr indicando escritura del dato (miss o hit respectivamente) y Flush, indicando que el dato se actualiza en memoria con la única copia válida de la cache. El diagrama de estados que describe funcionamiento del protocolo se presenta en la slide 20.

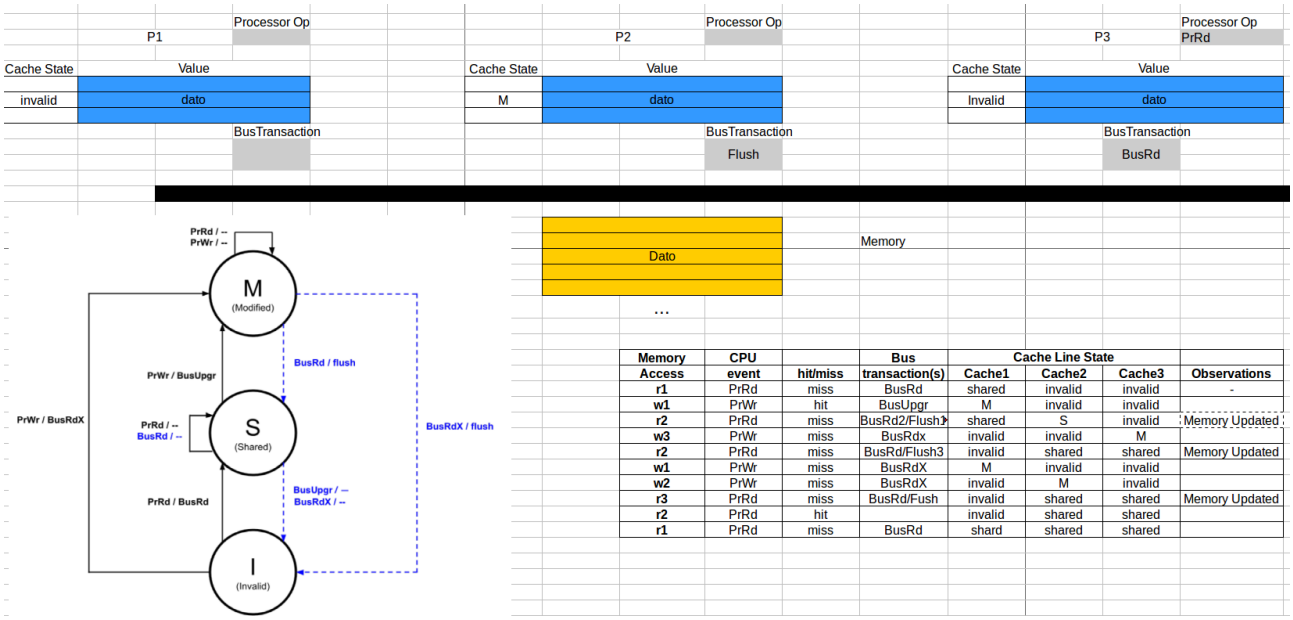
Posibles optimizaciones del protocolo Snoopy MSI tendrían como objetivo reducir el tráfico en el bus y los accesos a memoria. Por ejemplo, obtener el valor actualizado de una variable directamente desde la última cache que tiene una copia válida de la variable (slides 21-23).

Es importante mirar la optimización que se realiza en el caso del MESI. Su objetivo es que si vamos a hacer una lectura de un dato, y ningún otro procesador tiene una copia de ese dato en sus caches, el estado de la copia pasa a EXCLUSIVE. Este estado implica que tiene una copia limpia (no actualizada) del dato y ningún otro procesador tiene copia. Eso implica que si este procesador con la copia limipia exclusiva (exclusive estado) tiene que hacer una actualización del dato, no necesita realizar ninguna transacción en el BUS para indicar esa actualización ya que sabe que no hay más copias de esa posición de memoria. En el caso de MSI sí que es necesario ya que no teníamos información de exclusividad de la copia LIMPIA. El resultado es que MESI se ahorra una transferencia por el bus para mantener la coherencia de cache en una actualización de una línea de cache en estado exclusive.

Tanto en MSI como MESI, el estado M (modified) también denota exclusividad pero en este caso es copia DIRTY.

Para practicar el MSI hicimos el problema 1 . En el mensaje del Racó he añadido la hoja libreoffice con dos hojas dentro: una sin solución por si queréis practicarlo otra vez desde 0 y la otra con el resultado de la foto finish (cuando ya se acabó el ejercicio).

Al final llegamos a esto:



Memory Access	CPU event	hit/miss	Bus transaction(s)	Cache Line State Cache1	Cache2	Cache3	Observations
r1	PrRd	miss	BusRd	shared	invalid	invalid	-
w1	PrWr	hit	BusUpgr	M	invalid	invalid	
r2	PrRd	miss	BusRd2/Flush1	shared	S	invalid	Memory Updated
w3	PrWr	miss	BusRdx	invalid	invalid	M	
r2	PrRd	miss	BusRd/Flush3	invalid	shared	shared	Memory Updated
w1	PrWr	miss	BusRdX	M	invalid	invalid	
w2	PrWr	miss	BusRdX	invalid	M	invalid	
r3	PrRd	miss	BusRd/Fush	invalid	shared	shared	Memory Updated
r2	PrRd	hit		invalid	shared	shared	
r1	PrRd	miss	BusRd	shard	shared	shared	

No hicimos el apartado de MESI. Podéis intentar ver como se debería modificar el resultado final para que contemple un protocolo MESI en vez de MSI.

Slide 24 (no explicada en clase...)

La granularidad más fina de un dato que se puede mantener en forma “coherente” es una línea de cache o bloque de memoria. 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).

El proximo día explicaremos este problema en detalle y realizaremos el Problema 4 como ejercició práctico. Os cuelgo también una hoja libreoffice donde podéis intentar solucionar el problema.