Guió i material per classe no presencial PAR grup 20
Dilluns 15/03/2021, 10:00-12:00

L'objectiu del Tema 3 és entendre els fonaments de les arquitectures multiprocessador de memòria compartida, que donen suport a l'execució de models de programació de memòria compartida tipus OpenMP; en altres paraules, entendre com el sistema de memòria permet als processadors compartir dades (variables escalars i estructurades) i també entendre d'on surten els overheads de data sharing que hem vist en el Tema 2.

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 ben aviat 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 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?: *random*, *FIFO*, *LRU*, ...), polítiques d'escriptura en cas de hit (*write-through* si escric a cache i MP o *copy-back*, quan només escric a cache actualizant MP en cas de reemplaç) i en cas de miss (*write-allocate* si porto la línia a cache i *write-no-allocate* quan escric directament a MP sense portar la línia a cache).

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 darrer tema del curs).

Slide 13 (Resum de la vídeo lesson 4)

Comencem així amb les arquitectures SMP (Symmetric Multi-Processor) que us comentaven en la vídeo lesson 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**). 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**). Evidentment, 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.

Per disminuir la possibilitat de conflictes s'afegeix la memòria cache, que farà que només hagem d'anar a memòria principal quan fem un miss a la cache (cosa que per localitat hauríem de fer poc). Però ... quin problema ens presenta l'arquitectura UMA amb caches locals? **El problema de la coherència de memòria** (vídeo lesson 4).

Slide 14 (no explicada)

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.

Slide 15 (no explicada)

En els vídeos heu vist les dues solucions al problema: 1) convertir les caches en write-through, de manera que cada escriptura actualitzi el contingut de memòria principal i totes les altres copies a memòries cache locals; aquesta solució s'anomena **write-update**. I la 2), l'anomenada **write-invalidate**, amb la que quan és es fa una escriptura s'invaliden totes les còpies que hi ha a altres caches locals, incloent-hi la còpia de la memòria principal. Heu fet un parell de quizzes i veig que heu agafat bé la idea en general.

Repàs qüestionari Atenea

No s'observen massa problemes amb les qüestions. En quizzes de la vídeo lesson 4 hi ha un exemple de "no coherència"/"coherència" i un exemple per diferenciar "write update" de "write invalidate". He mirat i pràcticament tots ho heu respost bé, potser no a la primera vegada però si a la segona.

Anem a fer un exemple diferent per acabar d'entendre el problema. Donada la següent seqüència d'accessos a memòria, omplir la taula amb els continguts de cache, indicant si es cache/miss, supossant protocol write-update:

Memory Access	hit/miss	Cache line value			Observations
		cache1	cache2	cache3	
r1 (0)	m	**0**			from MP
w1 (1)	h	**1**			MP updated
r2	m	1	**1**		from MP
w3 (2)	m	2	2	**2**	MP and caches updated
r2	h	2	**2**	2	
w1 (5)	h	**5**	5	5	MP and caches updated
w2 (3)	h	3	**3**	3	MP and caches updated
r3	h	3	3	**3**	
r2	h	3	**3**	3	
r1	h	**3**	3	3	

I en cas de ser write-invalidate la taula quedaria:

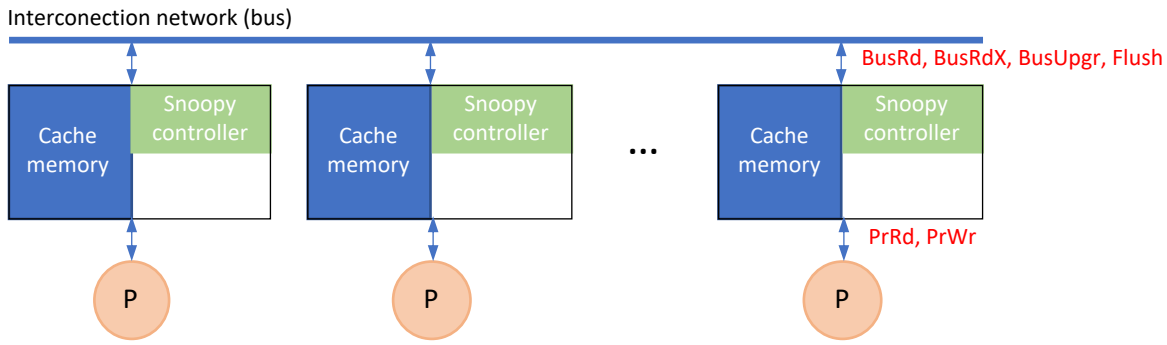
Memory Access	hit/miss	Cache line value			Observations
		cache1	cache2	cache3	
r1 (0)	m	**0**			from MP
w1 (1)	h	**1**			MP invalidated
r2	m	1	**1**		from cache1, MP updated
w3 (2)	m	x	x	**2**	from MP, other invalidated
r2	m	x	**2**	2	from cache3, MP updated
w1 (5)	m	**5**	x	x	from MP, other invalidated
w2 (3)	m	x	**3**	x	from cache1, other invalidated
r3	m	x	3	**3**	from cache2, MP updated
r2	h	x	**3**	3	
r1 (0)	m	**3**	3	3	from MP

L'actualització (write-update) o bé la invalidació (write-invalidate) ha d'utilitzar algun mecanisme de comunicació. Un pot ser aprofitant que hi ha un bus al qual estan connectades totes les caches i processadors (broadcast-based – snooping) i cada cop que es fa una invalidació o update es comunica amb broadcast a tot el sistema. L'altre possibilitat és intentar evitar aquest broadcasts a TOT el sistema, intentant guardar la informació de coherència en un directori, que queda distribuït (ja ho veurem) i que ajuda a evitar broadcasts massius.

Slide 17

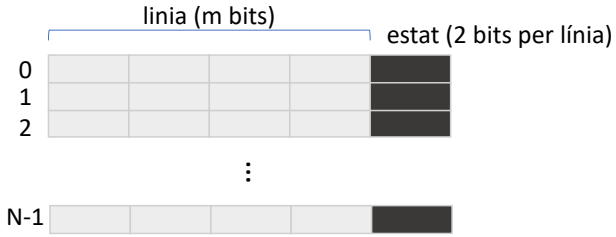
Ara ens centrarem als protocols de coherència amb broadcast-based (snooping). En ambdós casos (write-update o write-invalidate) la xarxa d'interconnexió tipus bus és el mecanisme que permet fer el **broadcast** d'aquestes comandes i estableix l'ordre en què les coses es veuen per tots els elements implicats. 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 contínuament el que es diu per la xarxa. Aquestes accions de comunicació pel bus les fa l'anomenat "snoopy cache controller".

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): PrRd/PrWr, BusRd/BusRdX, BusUpgr/Flush.

Slide 20

En aquesta slide es resumeix el funcionament del protocol snoopy write-invalidate en forma de graf d'estats, amb els tres possibles estats i les transicions entre ells provocades per les comandes que emet el processador cap a la cache o les que escolta pe bus. I en cada transició, el que el snoopy emet pel bus.

El protocol bàsic assumeix que en cas de no haver-hi copies de la mateixa línia en cap cache, o d'estar la línia "clean" en les caches que en tenen copia, és la MP qui ha de proporcionar la línia al processador que la demana. Només en cas d'estar "dirty", és a dir, en estat M en una (i només una de les caches), llavors és aquesta cache qui l'ha de proporcionar. Quan es fa flush s'envia la dada al processador que la demana i s'aprofita per actualitzar el contingut de la línia a MP.

Sabent el protocol, podem continuar l'exemple d'abans afegint-hi les comandes de coherència que es generen (**problema 1 del Tema 3)**:

Access	CPU event	Bus transaction	Status cache P_1	Status cache P_2	Status cache P_3
R1	$PrRd_1$ (miss)	$BusRd_1$ (*)	- → S	-	-
W1	$PrWr_1$ (hit)	$BusUpgr_1$	S → M	-	-
R2	$PrRd_2$ (miss)	$BusRd_2$ / $Flush_1$ (**) (+)	M → S	- → S	-
W3	$PrWr_3$ (miss)	$BusRdX_3$ (*)	S → I	S → I	- → M
R2	$PrRd_2$ (miss)	$BusRd_2$ / $Flush_3$ (**) (+)	I	I → S	M → S
W1	$PrWr_1$ (miss)	$BusRdX_1$ (*)	I → M	S → I	S → I
W2	$PrWr_2$ (miss)	$BusRdX_2$ / $Flush_1$ (**) (+)	M → I	I → M	I
R3	$PrRd_3$ (miss)	$BusRd_3$ / $Flush_2$ (**) (+)	I	M → S	I → S
R2	$PrRd_2$ (hit)	..	I	S	S
R1	$PrRd_1$ (miss)	$BusRd_1$ (*)	I → S	S	S

Observation: (*) line coming from main memory; (**) line coming (flushed) from another cache; (+) MP updated.

Trobareu un vídeo (opcional) a Atenea per aquesta setmana on s'explica el protocol MSI, per si us ha quedat algun dubte de l'explicació de classe.