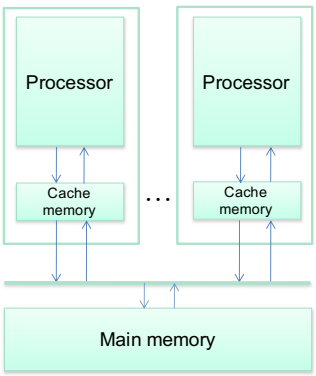
Guió i material per classe no presencial PAR grup 20 Dilluns 22/3/2021, 10:00-12:00

Repàs qüestionari Atenea sobre UMA coherence (I)



In a UMA (Uniform Memory Access time) multiprocessor, there are two or more identical processors connected to a single shared main memory through an interconnection network; this network allows any processor to access any memory location.

F In a UMA system, processors access to shared memory using instructions that are different from those used to access local memory (i.e. cache).

T In a UMA system with private (local) caches, the time to access main memory on a miss is independent of which processor is doing the access and which memory address is being accessed.

F In a UMA system with private (local) caches, the coherence protocol guarantees that for each line in main memory there can only exist a single copy of the line in one of the private cache memories of the system, independently of the number of processors available.

T In a UMA system with private (local) caches, a write-update coherence policy ensures that all copies in private caches of a memory line are updated.

T In a UMA system with private (local) caches and write-update coherence, the access to a memory address previously accessed by the same processor always results in a cache hit, unless the cache replacement algorithm decided to replace the line that contains that address.

F In a UMA system with write-invalidate MSI, the BusRdX invalidation command is generated on the bus every time a processor performs PrWr on a valid line in its cache memory.

T In a UMA system with write-invalidate MSI, when a processor performs a PrRd followed by a PrWr to an address residing in an uncached (or invalid) memory line, the associated snoopy needs to place two coherence commands on the bus, even when no other cores have accessed the memory location in between.

F In a UMA system with write-invalidate MSI, the BusRdX does not need to read the memory line that contains the address being accessed since the processor asks for the line with the purposes of modifying (writing to) it.

T In a UMA system with the simplest write-invalidate MSI protocol explained in class, main memory is responsible for providing a memory line when one of the private caches asks for it unless the line is in M state in another private cache.

F The implementation of the simplest write-invalidate MSI protocol explained in class requires two bits to track the state of each word in each cache line. Therefore, if the cache has capacity for N lines, each one storing M words, MSI requires N*M*2 bits to keep its contents coherent.

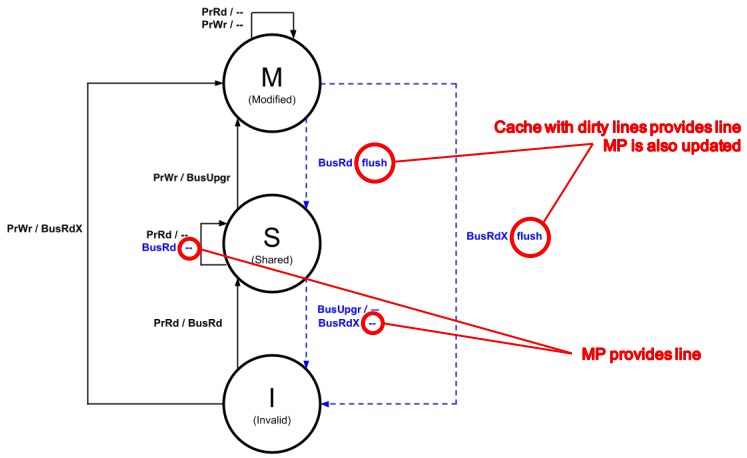
Repàs qüestionari Atenea sobre UMA coherence (II)

Access	CPU event	Bus transaction	Status cache P_0	Status cache P_1	Observations
w0	$PrWr_0$	$BusRdX_0$	- → M	-	miss, line from memory
r0	$PrRd_0$	--	M	-	hit
w0	$PrWr_0$	--	M	-	hit
r1	$PrRd_1$	$BusRd_1$ / $Flush_0$	M → S	- → S	miss, line from P_0
r0	$PrRd_0$	--	S	S	hit
w1	$PrWr_1$	$BusUpgr_1$	S → I	S → M	hit, invalidation P_1→P0
w0	$PrWr_0$	$BusRdX_0$ / $Flush_1$	I → M	M → I	miss, line from P_1, invalidation
r1	$PrRd_1$	$BusRd_1$ / $Flush_0$	M → S	I → S	miss, line from P_0

En resum: 2 BusRdX, 1 BusUpgr, 2 BusRd, and 3 Flush commands. 2 cache line invalidations, but only one of these two invalidations has an associated Flush command

Slides 21-23 (opcional, no explicat a classe)

Segur que més d'un de vosaltres s'haurà quedat pensant que el protocol MSI explicat no és massa òptim. 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 (flush). 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.



Per tant ja veiem dues possibles optimitzacions:

- **Protocol MSIF:** no cal llegir de MP si hi ha copia de la línia en una de les caches. Però. quina cache proporciona la línia? MSIF afegeix un nou estat F (Forward), que és una variant de l'estat S però amb responsabilitat de proporcionar la línia quan la demanen. Última línia que rebi la línia es queda en estat F (per localitat temporal).
- **Protocol MOSI:** No cal escriure a MP quan es fa flush, així l'escriptura es fa ràpida. El snoopy l'escriurà a MP quan la línia es reemplaci (és a dir, s'elimini de la seva cache), de mentre serà la responsable de proporcionar-la, estat O (Owner) que és també una variant de l'estat S.

Una altra optimització és pel cas de les variables privades o d'ús per part d'un sol processador. **Protocol MESI** afegeix l'estat E (Exclusive), variant també de l'estat S, que indica que només existeix una còpia de la línia: quan es porta la línia de MP es fica en estat E; si algú altre la

demana després, passa a estat S. Si el mateix processador l'escriu i està en estat E no cal informar a ningú, ja que no existeixen altres còpies a cache.

Podríem modificar la taula del problema 1 per veure com quedaria en cas de les tres optimitzacions anteriors. Per exemple, en el cas de MESI només canviarien les dues primeres línies, la resta quedarien igual:

Access	CPU event	Bus transaction	Status cache P_1	Status cache P_2	Status cache P_3
R1	$PrRd_1$ (miss)	$BusRd_1$ (*)	- → E	-	-
W1	$PrWr_1$ (hit)	--	E → M	-	-
R2	$PrRd_2$ (miss)	$BusRd_2$ / $Flush_1$ (**) (+)	M → S	- → S	-
...					

Pel cas de MSIF quedaria:

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

I finalment pel cas de MOSI quedaria:

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 → O	- → S	-
W3	$PrWr_3$ (miss)	$BusRdX_3$ / $Flush_1$ (**)	O → I	S → I	- → M
R2	$PrRd_2$ (miss)	$BusRd_2$ / $Flush_3$ (**)	I	I → S	M → O
W1	$PrWr_1$ (miss)	$BusRdX_1$ / $Flush_3$ (**)	I → M	S → I	O → I
W2	$PrWr_2$ (miss)	$BusRdX_2$ / $Flush_1$ (**)	M → I	I → M	I
R3	$PrRd_3$ (miss)	$BusRd_3$ / $Flush_2$ (**)	I	M → O	I → S
R2	$PrRd_2$ (hit)	..	I	O	S
R1	$PrRd_1$ (miss)	$BusRd_1$ / $Flush_2$ (**)	I → S	O	S

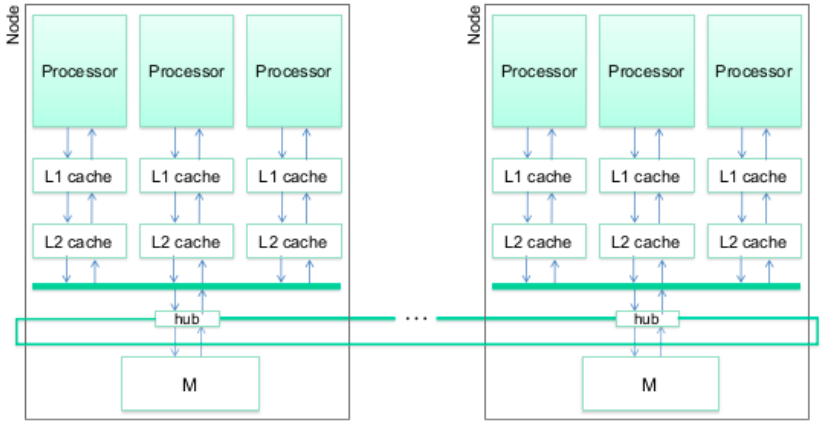
Slide 11

Avui continuarem explorant les arquitectures paral·leles de memòria compartida, en concret les anomenades **arquitectures NUMA (Non-Uniform Memory Architectures)**, en contrapartida a les **arquitectures UMA (Uniform Memory Access)** que vàrem presentar a la classe anterior. En ambdues arquitectures la memòria s'accedeix amb instruccions de load (lectura) i store (escriptura).

Recordem que el tret principal de les arquitectures UMA era la memòria global centralitzada que s'accedia a través de la xarxa d'interconnexió tipus bus; entre les dues permetien que qualsevol processador pogués accedir a qualsevol posició de memòria amb un temps constant. Però aquesta arquitectura té com principal limitació l'escalabilitat, és a dir, el nombre màxim de processadors que es poden connectar. Imagineu-vos que la cache privada de cada processador té un hit ratio del 90%, que vol dir que 1 de cada 10 accessos a memòria ocupen

el bus. A grans trets, només per donar una idea, si connectem més de 10 processadors el bus es congestionarà, augmentant el temps d'execució.

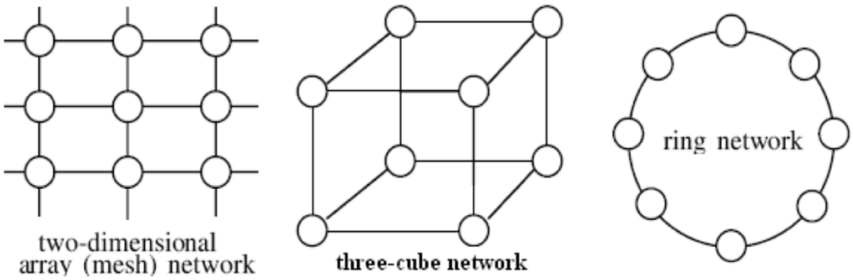
La idea és expandir el sistema amb múltiples nodes UMA interconnectats formant un sistema de memòria físicament distribuïda però lògicament compartida, amb coherència a nivell de cache.



Anem a veure tot això amb més profunditat.

Slide 36 (repàs vídeo lesson 5)

Les **arquitectures NUMA** es basen a distribuir físicament la memòria entre els diferents processadors, de manera que cada processador té una porció de la memòria total del sistema accessible de forma local, amb un temps d'accés petit. És el que anomenarem NODE: processador, jerarquia de cache i memòria principal. Quan el processador hagi d'accedir a posicions de memòria residents en altres processadors, ho haurà de fer a través de la xarxa d'interconnexió més o menys complicada, però que no és un bus al qual estan tots els nodes penjats. Per exemple:



El que està clar és que el temps d'accés passa a ser variable en funció de el lluny o a prop que es trobi la memòria que conté l'adreça que es vol accedir. Per tant, ja veieu que el temps d'accés a memòria ara serà no uniforme, depenent de quin processador fa l'accés i a quina posició de memòria accedeix.

I evidentment, tot això amb **coherència de memòria**, és a dir, que poden existir múltiples còpies de línies de memòria en les caches privades del processador, assegurant que un processador sempre podrà accedir a la versió més actualitzada de cadascuna de les línies de memòria. Com que la memòria no està centralitzada ni existeix un bus que tots els nodes comparteixen accés, la coherència no es pot mantenir amb un mecanisme de snooping com fèiem en el cas de UMA. Amb l'arquitectura NUMA la coherència es manté amb un mecanisme més escalable que s'anomena **directori**. El directori s'encarrega de guardar l'estat de cada línia

de MP, i per fer-lo escalable està dividit en **slices**, cada node amb un slice del directori que guarda la informació de coherència per les seves línies de MP.

El **hub** que apareix en cada node és qui s'encarregarà d'enviar comandes i línies entre nodes per tal de mantenir la coherència de cache.

Slide 39

Anem a veure més en detall el contingut del directori. El directori conte **una entrada per cada línia de memòria** en la memòria principal del node. En aquesta entrada es guarda: **l'estat de la línia**, que pot ser U (Uncached, no hi ha cap còpia de la línia en tot el sistema), S (Shared, hi ha una o més còpies de la línia) o M (Modified, existeix una única còpia de la línia en algun node i aquesta està modificada). Són 3 estats, per tant amb dos bits suficients per codificar-lo. A més a més, per saber quin o quins nodes tenen copia en estat S o M, l'entrada conté una tira de bits amb tants bits com nodes en total tenim en el sistema. Aquests bits s'anomenen **bits de presència**.

En el vídeo que heu mirat aquests dies, de fet només fa servir 1 bit d'estat per dir si la línia està Dirty o no. Amb els bits de presència ja pot saber si la línia està en estat U (tots a zero) o en estat S (algun bit de presència a 1 i el bit D a zero). Nosaltres per simplificar l'explicació farem servir MSU i així no haver d'anar parlant de bits a 0 i a 1.

Observeu que el directori requereix una part important de memòria per emmagatzemar la informació. En la slide teniu quin % de la memòria de dades representa el directori en funció del nombre de nodes del sistema. Per exemple, per 256 nodes necessitem 256+2 bits. Si les línies són de 64 bytes, això representa un 258/(64*8)*100=50,4%, és a dir, necessitem un 50,4% mes de memòria.

Repàs qüestionari Atenea sobre NUMA directory entry

Assume a directory-based multiprocessor system with four processors, each with 24 GB of main memory and with the corresponding directory associated. Memory lines are 64 bytes long. Data coherency is maintained using a write-invalidate protocol.

How many entries exist in each one of the four directories?
#entries = size memory ÷ line size = 24 GB ÷ 64 B = 24 x 2^{30} ÷ 2^{6} = 24 x 2^{24} = 3 x 2^{27}

Assuming the explanation in the video lessons, how many bits are needed in each directory entry? **Presence:** 1 bit per processor = 4 bits. **Valid**: 1 bit (Dirty / Clean). In total 5 bits.

Observeu que amb el MSU de les slides que us acabo de comentar serien 2 bits d'estat, per tant 4+2=6 bits).

Slide 40

Anem a veure com funciona el protocol de coherència basat en directori i MSU. Imagineu-vos ara que un processador qualsevol vol accedir a una posició de memòria x. La primera pregunta que ens hauríem de fer és: en quin NUMA node es troba la línia de memòria que conté la variable x? És a dir, **com es determina en quin processador (memòria MP del processador) es guarda una línia de memòria?** Normalment la decisió la fa el sistema operatiu, de manera que el primer processador que accedeixi a una línia de memòria serà la que l'emmagatzemarà;

i això considerarem que ja no canvia durant l'execució del programa. A aquest node l'anomenarem **home node**.

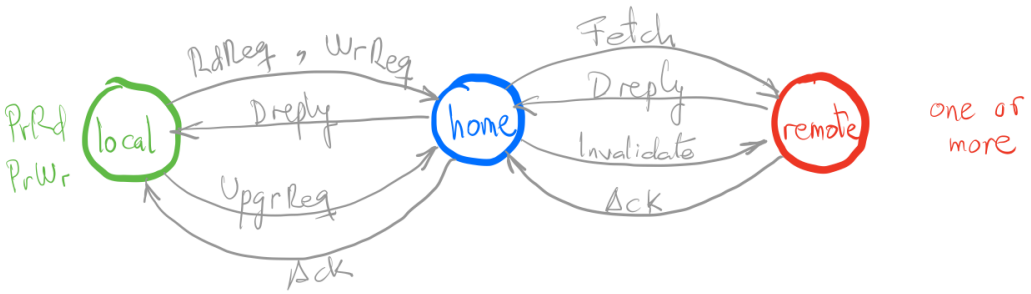
Per mantenir la coherència ara intervindran dos tipus de nodes addicionals al home node (que com acabem de dir és el node que conté, en la seva MP, l'adreça de memòria a la qual es vol accedir): **node local**, que és el node que conté el processador que inicia l'accés a memòria i l'accés li provoca un hit o miss en la seva jerarquia de memòria; i **nodes remots**, que són els nodes que contenen en la seva cache una còpia valida de la línia a la qual es vol accedir, podent estar aquestes còpies netes (estat S, sense modificar) o brutes (estat M, modificades).

En cas d'iniciar-se un accés a memòria en un node local que pugui afectar la coherència de memòria, el node local haurà de contactar primer amb el node home per dir-li el tipus d'accés que vol fer, que al seu torn contactarà, si és necessari, amb els nodes remots. Però sempre s'haurà d'anar primer al node home.

Slide 41

En aquesta slide es mostren les comandes de coherència en un protocol simplificat, molt equivalents a les que vàrem veure per MSI. RdReq, WrReq i UpgrReq que envia el node local al node home. El node home li contestarà amb Dreply per enviar-li la dada que demana (en cas de RdReq i WrReq) o Ack per confirmar que ja s'ha fet la comanda si no necessita enviar la dada (UpgrReq). Però abans de contestar, el node home haurà d'interactuar amb els nodes remots per invalidar-los la còpia de la línia que tenen (Invalidate) o per reclamar-los l'única còpia de la línia valida (Fetch). Els nodes remots de nou contestaran amb una confirmació de què ja s'ha fet (Ack) o amb la dada (Dreply), respectivament.

Anem a intentar entendre la seqüència de comandes que s'originarien en un accés qualsevol a memòria. El següent graf mostra les possibles comandes i entre quin i quin node poden succeir.



Slide 42

I en aquesta slide un possible exemple. Imagineu-vos 3 processadors, el 1 el node local, el 2 el node home (també amb còpia a la seva cache) i el 3 el node remot també amb còpia a la seva cache. El directori inicialment indica estat S amb còpies a nodes 2 i 3. El node 1 vol fer una petició d'escriptura que resulta en un miss, ja que no està a la seva cache. Envia WrReq al node 2 (home), que després de consultar el directori invalida la seva còpia de cache i envia la comanda d'Invalidate al node 3, que respon amb un Ack. Un cop invalidades les copies, canvia l'estat del directori a M indicant només copia en el node 1. Finalment envia la línia al node local fent un Dreply.

Slide 43

I de fet el sistema pot combinar una arquitectura UMA amb múltiples processadors dins del node, i una arquitectura NUMA entre nodes, tots els processadors compartint memòria. I això

voldrà dir que s'haurà de combinar la coherència UMA dins d'un node i la coherència NUMA entre nodes. En la slide es veuen tres exemples de variables var-a, var-b i var-c, en diferents estats i copies en les caches locals dels nodes NUMA.

Back to Slide 25

Anem a veure què ens causa el tràfic de coherència que estem estudiant. El tràfic de coherència apareix perquè els processadors comparteixen la informació que hi ha a les línies de cache. Dos tipus de compartició: **true i false sharing**.

El **true sharing** és inevitable, simplement causada perquè els processadors estan treballant conjuntament per resoldre un problema i per això necessiten compartir informació (variables).

Slide 26

Exemple de true sharing, causat per l'accés a la variable *result*. Es pot reduir la compartició de la variable, i per tant, reduir el tràfic de coherència? Sí ... privatitzant la variable i només actualitzant al final una única vegada en lloc de a cada iteració del bucle.

I el **false sharing**? Anem directament a veure un exemple.

Slides 27 i 28

En aquesta slide es mostra un altre exemple en el qual s'està accedint als dos camps x i y d'un struct f. Tenim dues tasques, una d'elles llegint f.x i l'altre escrivint f.y. Quin problema hi ha? Tot i que les dues tasques no comparteixen la x i la y, el protocol de coherència entra en joc, ja que els dos camps del struct conviuen en la mateixa línia de cache. Cada escriptura que fa la tasca 2 (la que escriu sobre f.y) invalida la còpia de la línia que té la tasca 1 (la que només llegeix f.x), per després tornar-li a proporcionar la línia quan la tasca 1 la torna a necessitar. Fixeu-vos que en principi haurien de ser tot hits. Això s'anomena **false sharing**.

El false sharing el podem eliminar si fem que cadascun dels camps del struct estigui en una línia de cache diferent. I això es pot aconseguir ficant-hi un padding, un camp no utilitzat, amb una mida suficient per separar els dos camps `x` i `y`.

Jump to Slide 45

Per acabar el tema, comentar què més provoca tràfic de coherència en les arquitectures NUMA. Està clar que els overheads associats a true i false sharing ara encara són més elevats. De fet si recordem el **model de data sharing** del Tema 2, teníem que $t_{data\_sharing} = t_s + m \cdot t_w$, on t_s era el temps d'start-up del accés remot i t_w el transfer time per cada byte.

El component t_s és degut al temps de localitzar la línia que conté la dada (en el home node) i activar tot el protocol de coherència. I el t_w és el que triguen les dades en viatjar des del node que té les dades (home si clean o remote si dirty) fins al node local.

I aquests overheads es poden veure agreujats si les dades no estan ben inicialitzades. Per què? Recordeu first touch! Mireu els dos escenaris en la slide. En el de l'esquerra totes les dades queden inicialitzades en el node 0 (que es converteix en el home de les línies corresponents). En canvi en l'escenari de la dreta les dades s'inicialitzen en paral·lel i cada processador es fa home d'unes quantes línies dels vectors a i b. Això és important?

Slide 46

Doncs sí, ja que després a l'accedir als vectors a i b tindrem o no més o menys tràfic de coherència. I en la slide es mostra una regió paral·lela amb dos bucles que fan un determinat càlcul. Com veieu en el cas de la dreta tots els accessos són locals, sense cap mena de tràfic de coherència; en canvi en l'escenari de l'esquerra hi haurà

- Primer bucle: RdReq al accedir al vector a i WrReq al accedir al vector b, dels nodes P_1, P_2 i P_3 cap al node home (P_0).
- Segon bucle: UpgrReq al escriure sobre el vector a, de nou dels nodes P_1, P_2 i P_3 cap al node home (P_0). L'accés al vector b no provoca tràfic de coherència.

Problemes per la ultima classe després de setmana santa i abans del control

Problema 10 del Tema 2 (data sharing overheads)
Problema 6 del Tema 3 que presenta problemes de false sharing, però requereix una solució una mica diferent a l'explicada aquí.
Problema 4 del Tema 3 on es treballa el protocol NUMA