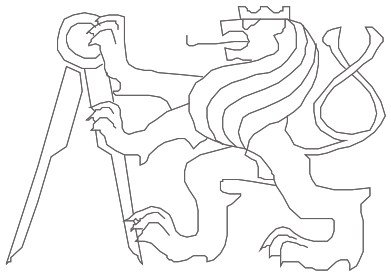
Pokročilé architektury počítačů

Paměťová konzistence



České vysoké učení technické, Fakulta elektrotechnická

Rekapitulace

- Moderní procesory pracují podstatně rychleji než hlavní paměť, na kterou jsou připojeny. Řešení?
  - Mezi procesor a paměť se vloží skrytá/é paměť/i.
- Omezením počtu přístupů do paměti se výkon výrazně zvýší. Výkon pozitivně ovlivní i minimum „cache misses“ – data nejsou ve SP, ale to není předmětem našeho dnešního zájmu.
- Dnešní téma: paměť musí být **konzistentní**,
- tedy když jiný procesor v SMP systému změní obsah své paměti, musí se to projevit i v obsahu **všech dotčených pamětí i SP**!

Terminologie dnešního tématu podrobněji

- Pravidla pro provádění paměťových operací,
  - Paměťová koherence
    - Pravidla pro přístupy k paměťovým místům
  - Paměťová konzistence
    - Pravidla pro všechny paměťové operace
- zajištění sekvenční konzistence,
- slabší modely paměťové konzistence
  - Konzistence dosahovaná s pomocí synchronizace, resp. synchronizačních proměnných.

Definice koherentní paměti (obvyklá)

- Řekneme, že paměťový systém multiprocesorového systému je **koherentní**, jestliže výsledek jakéhokoli provádění programu je takový, že pro každé paměťové místo je možné sestavit myšlené sériové pořadí čtení a zápisů k tomuto paměťovému místu a platí:

- 1. Paměťové operace k tomuto paměťovému místu pro každý proces jsou provedeny v pořadí, ve kterém byly spuštěny tímto procesem.
- 2. Hodnoty vrácené každou operací čtení jsou hodnotami naposledy provedené operace zápis do tohoto paměťového místa vzhledem k sériovému pořadí.

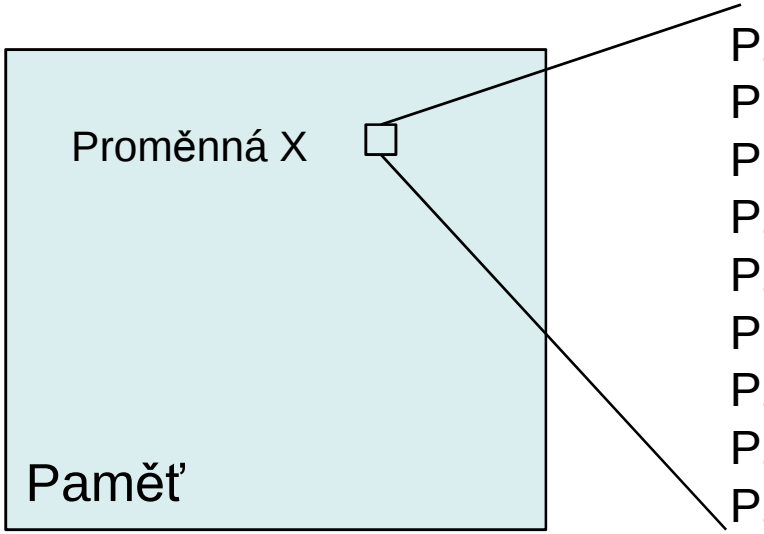
Koherence

Proces P1:

```
X=0;
if(X ==0) {
  y=fun();
  X = 1;
}
```

Proces P2:

```
X=0;
while(X ==0)
  { ; }
X = 2;
```



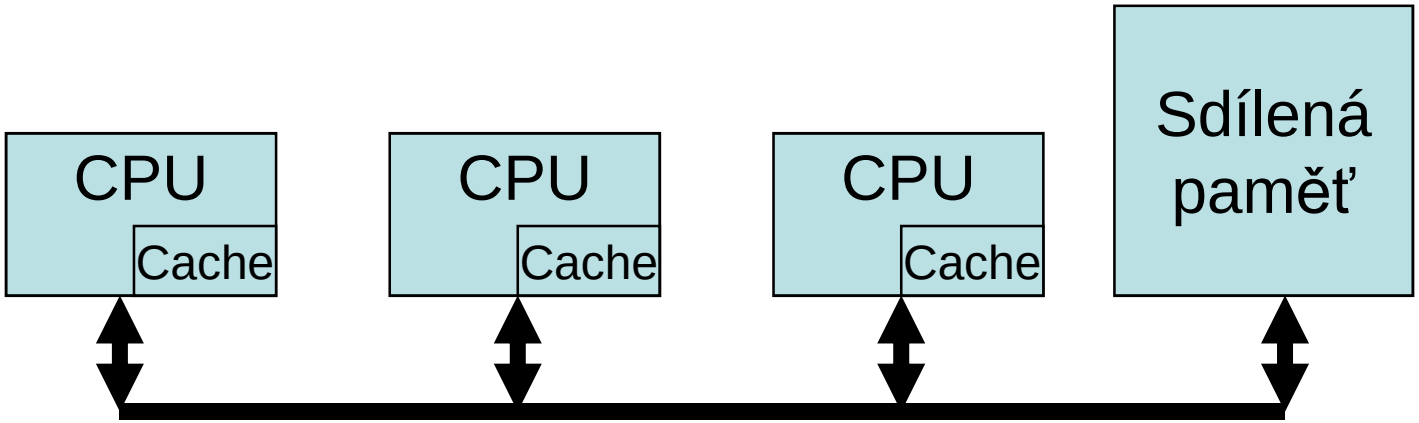
P2: X=0;	P2: read(X)
P1: X=0;	P2: X=0;
P1: read(X)	P1: X=0;
P2: read(X)	P1: read(X)
P2: read(X)	P2: read(X)
P1: X=1;	P1: X=1;
P2: read(X)	P2: read(X)
P2: read(X)	P2: read(X)
P2: X=2;	P2: X=2;

Máme jistotu, že když P2 uvidí X==2, bude funkce fun() volaná procesem P1 se všemi důsledky vykonána?

Konzistence

- **Konzistence** oproti koherenci specifikuje v jakém pořadí jednotlivé procesy spouštějí své paměťové operace, či jak se toto pořadí jeví ostatním procesům.
- Uvažuje sekvenční pořadí všech paměťových operací.
- Koherence uvažuje myšlené sekvenční pořadí pouze vůči jednotlivým paměťovým místům, nikoli mezi přístupy do různých paměťových míst.
- Konzistence definuje co je korektní chování sdílené paměti z pohledu čtení a zápisů

Na tomto MP systému simulujte provedení programu



Sdílená sběrnice

Počáteční hodnoty oba procesy: `x=0, y=0`

```
P1:          P2:
x = 1;       while(y==0) {;}
y = 1;       print(x);
```

Očekáváme, že *print(x) vytiskne 1.*

Stačí koherence k ***rozumnému*** chování sdílené paměti?

- Proměnná *f* indikuje, že proměnná *x* byla změněna.
- Paměťová koherence ovšem nijak nespecifikuje v jakém pořadí jednotlivé procesy P1 a P2 spouštějí své paměťové operace (read, write) a nijak nespecifikuje v jakém pořadí uvidí P2 zápisy do x a f.
- Koherence pouze zajistí, že P2 se nakonec dozví nové hodnoty *x* a *f,* ale nijak nespecifikuje v jakém pořadí tyto nové hodnoty obdrží.

- Proto ani na počítači s koherentním paměťovým systémem není vyloučeno, že P2 vytiskne starou hodnotu *x* (tj. 0).

Koherence ke „konzistenci“ nestačí.

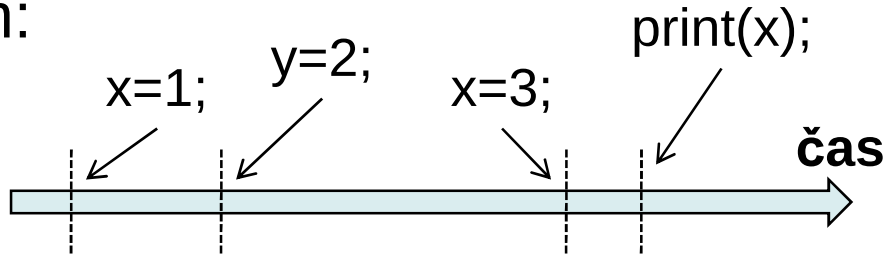
Koherence skrytých pamětí je nezbytná k zajištění datové (paměťové) konzistence v multiprocesorovém systému.

- Koherence – co vrátí operace čtení (jakou hodnotu)
- Konzistence – kdy bude zapsaná hodnota vrácena čtením.

Striktní konzistence

- Jednoprocesorový systém:

```
x = 1;
y = 2;
x = 3;
print(x);
```



(**Jakékoliv čtení z paměti z adresy *x* vrátí hodnotu uloženou při posledním zápisu na adresu *x*.**)

- Víceprocesorový systém:
  - podmiňuje existenci přesného globálního času ve všech uzlech a okamžité propagování změn
  - nerealistický až absurdní požadavek

Sekvenční konzistence (sequential consistency)

Minulou přednášku jsme si řekli:

- Definice (Lamport, 1979): “Počítač je sekvenčně konzistentní, jestliže je výsledek provádění programu stejný, jako kdyby operace na všech procesorech byly provedeny v nějakém sekvenčním pořadí a operace každého jednotlivého procesoru se objevují v této posloupnosti v pořadí daném jejich programem.”

- Sekvenční konzistence je slabší model než striktní konzistence, avšak implementovatelná…
- Jestliže procesy běží na různých procesorech, je povoleno libovolné prokládání jejich instrukcí, avšak s podmínkou, že všechny procesy vidí stejné pořadí změn paměti. Změny nejsou propagovány okamžitě, je pouze zaručeno pořadí (následek nepředchází příčinu).

Postačující podmínky pro zajištění SC

- Každý procesor P(i) spouští paměťové operace v programovém pořadí.
- Procesor P(i), který spustí operaci Write, nespustí další paměťovou operaci dříve, než se tato dokončí.
- Procesor P(i), který spustí operaci Read, nespustí další paměťovou operaci dříve, než se tato dokončí a než se dokončí operace Write (globally visible), jejíž hodnotu vrací operace Read. -> write atomicity
- It is important that compiler should not change the order of memory operations – many optimizations that are commonly employed for uniprocessors violate this/these condition/s.

Sekvenční konzistence (sequential consistency)

Předpokládejme, že na začátku platí a=0, b=0, c=0.

P1:

```
a=1;
print(b,c);
```

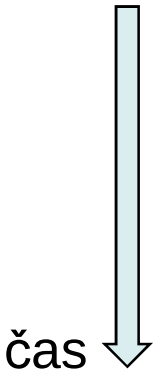
P2:

```
b=1;
print(a,c);
```

P3:

```
c=1;
print(a,b);
```

Může to dopadnout takto:




```
a=1;
b=1;
c=1;
print(b,c);
print(a,c);
print(a,b);
```

Výstup: 111111

```
a=1;
print(b,c);
b=1;
print(a,c);
c=1;
print(a,b);
```

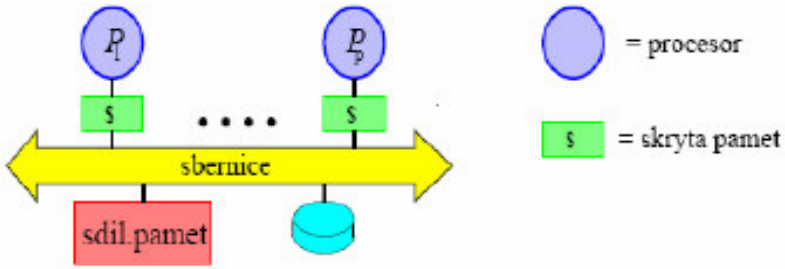
Výstup: 001011

atd.

Existuje 6! různých variací proložení instrukcí, ale ne všechny splní podmínku sekvenční konzistence.

Zajištění konzistence v SMP se sdílenou pamětí?

Sběrnicově založené SMP a koherence a konzistence



- Sdílená sběrnice je **excelentní** zařízení pro implementaci koherenčních skrytých pamětí a sekvenčně konzistentní paměti.!!!!
- To je jeden z důvodů, proč sběrnicové systémy SMP
  - jsou levná a výkonná rozšíření jednoprocesorových PC,
  - dominují na trhu serverů a přibývají na významu i na trhu desktopů,
  - jsou stavebními kameny pro rozsáhlejší systémy (DMP, svazky).

Sběrnice zajišťuje *serializaci* paměťových operací, *propagaci* výsledků a za určitých podmínek i *atomicitu* paměťových operací (záleží na protokolu sb.).

Myšlené pořadí operací požadované v definicích koherence a konzistence je pořadím těchto operací na sběrnici.

Architektura počítačů

Zdroj: Bečvář

Závěr

- Pro synchronizaci v paralelních počítačích se sdílenou pamětí je vhodné definovat *sekvenčně konzistentní paměťový systém*.
- Synchronizační operace jsou *vzájemné vyloučení, dvoubodová synchronizace* a *synchronizační bariéra*.
- Základem implementace synchronizačních operací jsou atomické RMW primitivy.
- V ISA procesorů se vyskytují RMW instrukce T&S, SWAP, F&I, C&S
- Novější procesory podporují tvorbu RMW primitiv pomocí instrukcí LL a SC, které umožňují efektivní implementaci synchronizačních operací v systémech se skrytými pamětmi

Architektura počítačů

Konzistence – otázka synchronizace

Předpokládejme dva procesy P1 a P2:

P1: A = A+1; P2: A = A +2;

Pokud by operace součtu byly atomické, pak A bude mít hodnotu A+3.

```
P1:   load R1, A        P2:   load R1, A
      addi R1,R1,1              addi R1,R1,2
      store R1,A                store R2,A
```

Možná sekvence, která produkuje A+1 jako výsledek:

```
P1:   load R1, A
                        P2:   load R1, A
                              addi R1,R1,2
                              store R1,A
      addi R1,R1,1
      store R1,A
```

Konzistence – otázka synchronizace

Předpokládejme dva procesy P1 a P2:

P1: A = A+1; P2: A = A +2;

Řešením je zámek na sdílenou proměnnou A.

```
while(!acquire(lock)) { čekací algoritmus }
výpočet nad sdílenými daty
release(lock)
```

Protože několik procesů může chtít získat (acquire) zámek současně, proces získávání zámku musí byt atomický.

• čekací algoritmus: ***busy waiting*** nebo ***blocking waiting***. Busy waiting – neustále zkouší získat zámek, blokující čekání – proces uspí sám sebe, uvolní procesor; vzbudí se až bude zámek uvolněn. Taktéž kombinace obou technik..

Konzistence – otázka synchronizace

Předpokládejme dva procesy P1 a P2:

P1: A = A+1; P2: A = A +2;

Problém získání zámku lze v nejjednodušším případě řešit synchronizační sdílenou proměnnou (uloženou v paměti), která může mít hodnotu 0 (zámek je volný) nebo 1 (obsazen).
Problém získání zámku pak spočívá v otestování na 0 a nastavení na 1. Toto však musí být **nedělitelné**!

Potřebujeme tedy instrukci, která:
přečte, modifikuje a zapíše hodnotu do paměti bez interference.

ISA: ***test-and-set*** - všechny dnešní procesory ji podporují; je nejjednodušším případem atomické operace

• zobecněním *test-and-set* je exchange-and-swap a compare-and-swap

Konzistence – otázka synchronizace

Předpokládejme dva procesy P1 a P2:

P1: A = A+1; P2: A = A +2;

Použitím ***test-and-set*** by mohl náš program pro P1 vypadat takhle:

```
loop: test-and-set R2, lock     // testuje lock a nastaví do R2
      bnz R2, loop              // pokud R2 není 0 vrátí se na loop
      load R1, A
      addi R1, R1, 1
      store R1, A
      store R0, lock            // uvolní zámek zapsáním 0. R0=0.
```

Konzistence – otázka synchronizace

Předpokládejme dva procesy P1 a P2:

P1: A = A+1; P2: A = A +2;

Dalším typem atomické operace je fetch-and-xx operace (fetch-and-increment, fetch-and-add, fetch-and-store,…). Program pak bude vypadat takhle:

P1: fetch-and-inc A; P2: fetch-and-inc A;
fetch-and-inc A;

Konzistence – otázka synchronizace

Předpokládejme dva procesy P1 a P2:

P1: A = A+1; P2: A = A +2;

Dalším alternativou je instrukční pár ***load-locked*** (ll) (nebo **load-link, load-linked, load-and-reserve**) a ***store-conditional*** (sc)

- Instrukce ll vrací aktuální hodnotu uloženou v paměti, sc pak uloží novou hodnotu jenom pokud nebyla nikým jiným modifikována – atomická operace je úspěšná – implementace vyžaduje *linked register*..

```
loop:   ll R1, A          // A načte do R1, adresa A do linked reg.
        addi R1, R1, 1
        sc R2, A          // R2 = linked register == adresa(A) ? 1 : 0
        bz R2, loop
```

- IBM PowerPC, DEC Alpha, MIPS, ARM, IA-64

Další modely konzistence

- *Kauzální konzistence (causal consistency)*
  - Zápisy, které jsou potenciálně kauzálně vázané, musí být viděny všemi procesy ve stejném pořadí. Souběžné zápisy mohou být viděny v různém pořadí.
  - Rozlišuje události, které jsou potenciálně závislé a které nikoliv.
  - Slabší než sekvenční konzistence

- *PRAM konzistence (PRAM consistency* = pipelined random access memory consistency*) = FIFO konzistence*
  - Zápisy prováděné jedním procesem jsou viděny ostatními procesy v tom pořadí, ve kterém byly prováděny, avšak zápisy různých procesů mohou být viděny různými procesy různě.
  - Slabší než kauzální konzistence

Další modely konzistence

Slabá konzistence (weak consistency)

- **Pozn.** Termín slabá konzistence se někdy používá pro všechny ostatní modely konzistence slabší než je model striktní konzistence.
- Motivační příklad: Proces v kritické sekci čte/zapisuje v rychlé smyčce hodnoty nějakých proměnných. Ostatní procesy nemají důvod jednotlivé zápisy vidět, proto není nutné, aby byly všechny propagovány. Proces necháme ukončit kritickou sekci a poté zajistíme rozeslání změn všem ostatním procesům.

1. Přístup k synchronizačním proměnným je sekvenčně konzistentní – všechny přístupy k synchr. proměnným jsou viděny jinými procesy v tom samém pořadí. (Všechny ostatní přístupy mohou být viděny různými procesy v různém pořadí.)
2. Přístup k synchronizační proměnné není povolen dokud neskončí všechny předchozí zápisy.
3. Před přístupem k datům musí být dokončeny všechny předchozí přístupy k synchronizačním proměnným

- Příklad: PowerPC má instrukci Sync, OpenMP používá flush

Další modely konzistence

Slabá konzistence (weak consistency)

- Programátor tedy definuje oblasti, kde **paměťové operace nad sdílenými proměnnými můžou být libovolně přeuspořádány**. Ty oddělí bariérami.
- Všechny datové operace (se sdílenými proměnnými) PŘED bariérou se musejí dokončit před vstupem do bariéry
- Všechny datové operace ZA bariérou musejí čekat na dokončení bariéry
- Bariéry jsou viděny v programovém pořadí

- Příklad. *Nechť A = Flag = 0*

proces P1:	**proces P2:**
A = 3;	
flush;	while (Flag != 1) {;}
Flag = 1;	... = A;

Když P2 vidí Flag=1 je garantováno, že P2 bude číst A s hodnotou 3, dokonce i když paměťové operace P1 před flush-em a za flush-em jsou přeuspořádány hardwérem nebo kompilátorem.

Další modely konzistence

- *Uvolňovací konzistence (release consistency)*
  - Existence dvou synchronizačních operací – acquire (požadavek) a release (uvolnění). Před zápisem uzel musí získat, pak uvolnit..
  - Kritická sekce…
  - Systém poskytuje tuto konzistenci pokud všechny zápisy určitého uzlu jsou viděny všemi ostatními uzly po uvolnění objektu (release) a před tím, než někdo jiný o něj požádá (acquire).
  - Dva přístupy:
    - pilný(*eager)* – všechny koherenční akce jsou vykonány na release
    - líný (lazy) – všechny koherenční akce jsou vykonány až při příštím acquire
  - Jestliže jsou požadavky a uvolnění správě spárované, výsledek jakéhokoliv výpočtu je ekvivalentní s sekvenčně konzistentní pamětí.
  - Release konzistenci (taky weak konzistenci, atd.) řadíme mezi modely relaxed consistency

Další modely konzistence

- Relaxed consistency
  - Připomeňme si, že sekvenční konzistence má dva požadavky: programové pořadí (pro operace čtení a zápisu) a atomicitu zápisu. Všechny modely, které uleví na těchto požadavkách jsou hodnoceny jako relaxed consistency. Co to přináší? Možnost paralelního běhu..

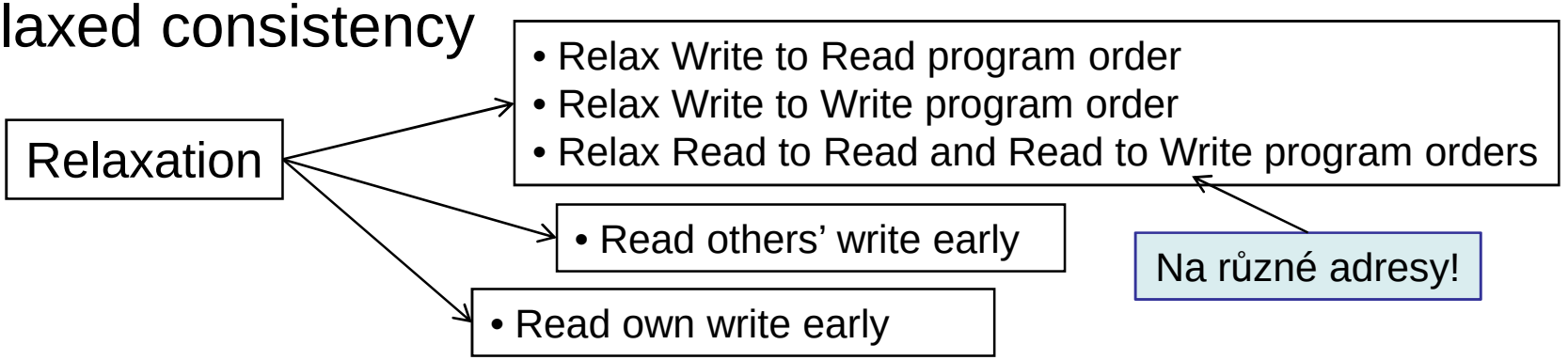
  - V programu bude napsáno:

```
Instr.1:   load R1, A
Instr.2:   load R2, B
…
Instr.N:   store R1, A
```

    Vadí, pokud bychom dokončili instrukci č.2 před instrukcí č.1?
    A co instrukce č.N ???

Další modely konzistence

- Relaxed consistency



Relaxation	W — R Order	W — W Order	R — RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					√	
IBM 370 [14]	√					serialization instructions
TSO [20]	√				√	RMW
PC [13, 12]	√			√	√	RMW
PSO [20]	√	√			√	RMW, STBAR
WO [5]	√	√	√		√	synchronization
RCsc [13, 12]	√	√	√		√	release, acquire, nsync, RMW
RCpc [13, 12]	√	√	√	√	√	release, acquire, nsync, RMW
Alpha [19]	√	√	√		√	MB, WMB
RMO [21]	√	√	√		√	various MEMBAR's
PowerPC [17, 4]	√	√	√	√	√	SYNC

Zdroj: Adve and Gharachorloo: Shared Memory Consistency Models, WRL Research Report.

Další modely konzistence

- Relaxed consistency

  Mezi tyto modely také řadíme:

  - Total Store Ordering (TSO) - SPARC: operace čtení může být dokončena před dřívějším zápisem na jinou adresu, ale čtení nemůže vrátit hodnotu zapsanou jiným procesorem do té doby než všechny procesory vidí zápis (procesor vrací hodnotu vlastního zápisu před tím než ji ostatní vidí)
  - Processor Consistency (PC): čtení může být dokončeno před tím než dřívější zápis (libovolným procesorem na libovolné místo) je viditelný všem. Pozn.: Existují různé definice této konzistence..

Relaxation	Example Commercial Systems Providing the Relaxation
W — R Order	AlphaServer 8200/8400, Cray T3D, Sequent Balance, SparcCenter1000/2000
W — W Order	AlphaServer 8200/8400, Cray T3D
R — RW Order	AlphaServer 8200/8400, Cray T3D
Read Others' Write Early	Cray T3D
Read Own Write Early	AlphaServer 8200/8400, Cray T3D, SparcCenter1000/2000

Some commercial systems that relax sequential consistency.

Zdroj: Adve and Gharachorloo: Shared Memory Consistency Models, WRL Research Report.

Dnes aktuální otázky:

- Implementace koherence a konzistence v systémech **bez sběrnice** metodou broadcastu

- Implementace koherence a konzistence v rozsáhlých systémech na bázi directories

Připomínám:

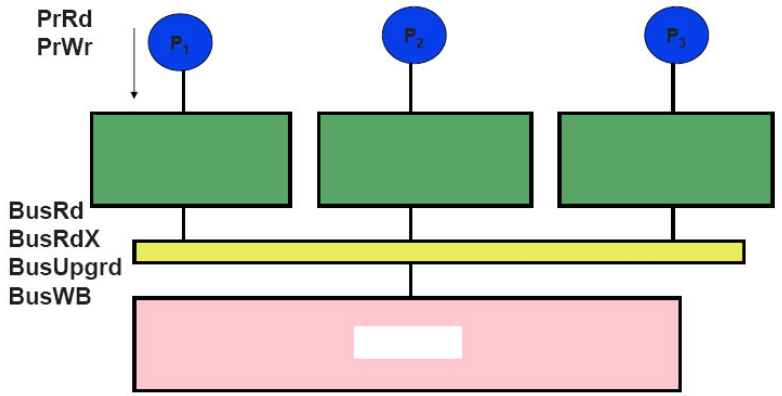
- Implementaci MESI v systému se SP (cache) a split-transaction sběrnicí jsme už probírali.

MESI – Opakování. Pohled přes události

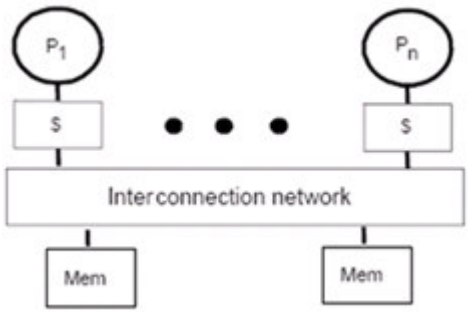
- Události - strana procesoru:
  - PrRd – Processor Read,
  - PrWr – Processor Write.
- Události – transakce na sběrnici:
  - BusRd – čtení bloku do cache (reakce na PrRd na blok ve stavu **Invalid**),
  - BusRdX – čtení se získáním eXclusivity, bude se zapisovat (reakce na PrWr na blok ve stavu **Invalid**)
  - BusUpgrd – upgrade práva na zápis, zneplatnění ostatních kopií (reakce na PrWr na blok ve stavu **Shared**)
  - BusWB – zápis bloku ve stavu **Modified** do paměti při nahrazení jiným blokem. Jde o úklidovou operaci a neovlivňuje koherenci, neboť pouze mění místo, kde je uložena aktuální hodnota daného bloku.

Split transaction sběrnice je mezistupněm k síťově orientovanému Request-Response protokolu:

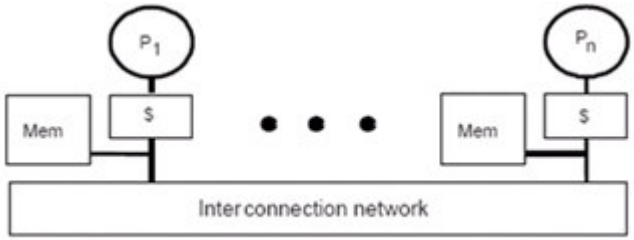
- BusRd
  - 1. Žádost – pošli broadcast všem procesorům a hlavní paměti, zdali nemají kopii daného bloku.
  - 2. Odpověď – všechny procesory provedou snooping a odpoví zdali mají kopii daného bloku (S signál), pokud má jeden Dirty (Modified/Owned) kopii, pošle zpět i data. Následně odpoví hlavní paměť s hodnotou bloku.
- BusUpgrd
  - 1. Žádost – pošli broadcast všem procesorům ať si zneplatní kopii daného bloku.
  - 2. Odpověď - všechny procesory provedou snooping a odpoví v okamžiku, kdy provedou zneplatnění (resp. zneplatnění je uloženo do fronty)
- BusRdX
  - Podobně jako BusRd + navíc zneplatnění.



Implementace koherence a konzistence paměti v systémech bez sběrnice metodou **Broadcast**



nebo



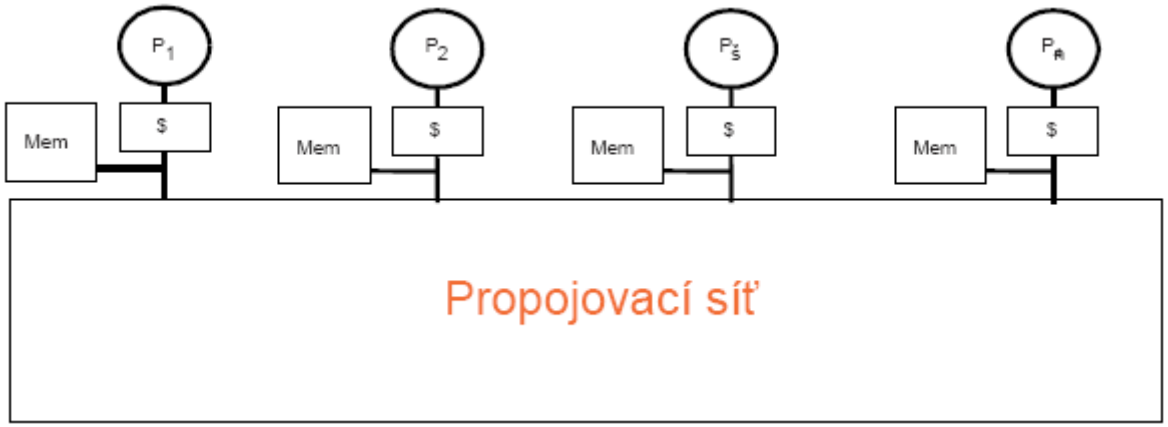
Problémy v systémech s nesběrnicovým propojením

- Více žádostí může být v běhu současně
  - Co když 2 procesory současně provedou BusRdX nebo BusUpgrd ?
  - Co když odpovědi a žádosti dorazí k různým procesorům v různém pořadí?

- Řešení:
  - Serializace (jinak synchronizace) požadavků (nutná kvůli koherenci a konzistenci) – jako na sběrnici … Ale sběrnici nemáme ….
- Místo serializace: Domovský uzel – Home Node,

  - Hlavní paměť, pokud je centralizovaná, nebo
  - Procesor u kterého je část hlavní paměti, ve které je daný blok.

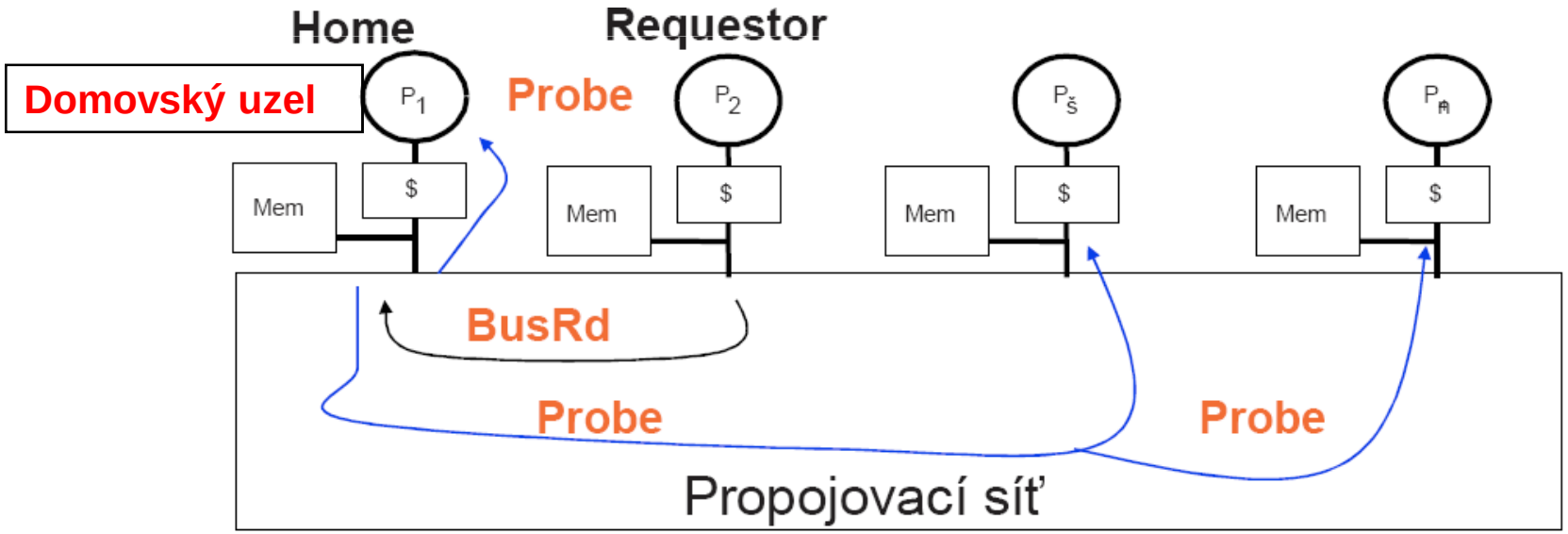
Příklad systému (a lá AMD Hypertransport)



- Hlavní paměť je distribuovaná, části jsou u jednotlivých procesorů (systém NUMA), ale je globálně fyzicky adresovaná a SP (cache) jsou koherentní.
- OS se snaží, aby maximum stránek bylo v lokální paměti.
- **Domovský uzel** – uzel u kterého je část hl. paměti, kam patří daný blok,
- **Vlastník** (owner) – uzel, který má aktuální kopii (ve stavu Modified nebo Owned).

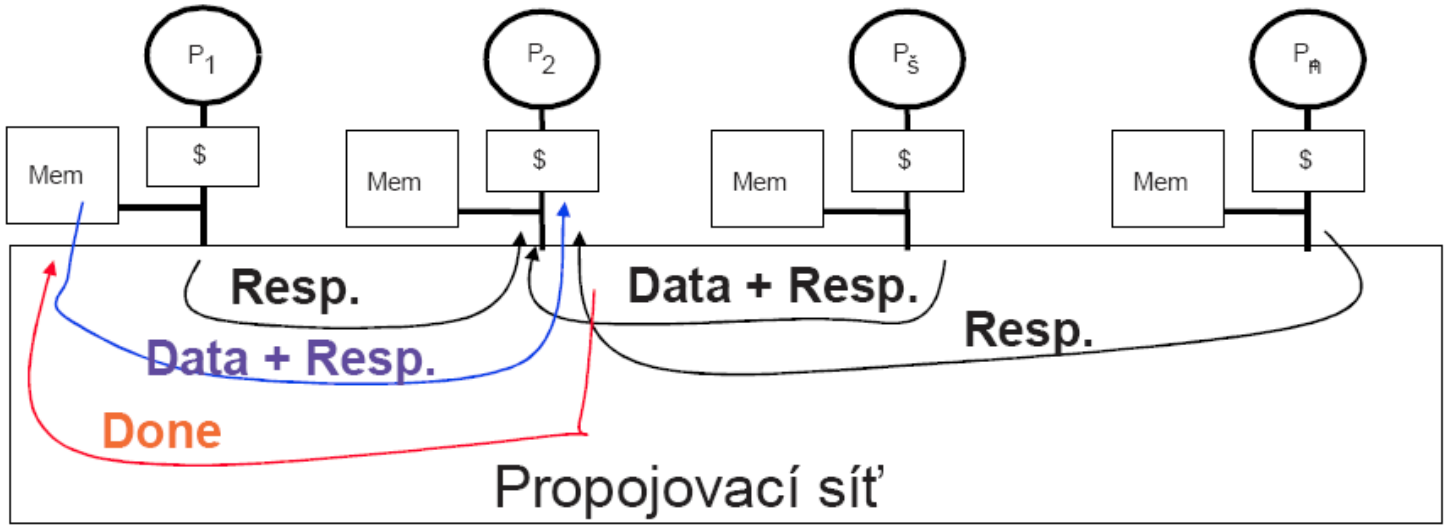
Příklad - P2: BusRd na položku, kterou má jenom P1

- P2 pošle žádost BusRd směrem k domovskému uzlu P1
- Paměť u P1 začne číst hodnotu bloku a P1 rozešle broadcast všem procesorům (P1,P3,P4) zdali nemají hodnotu žádaného bloku (mohou ji mít ve stavu E nebo M...)
- Této žádosti se říká Probe.



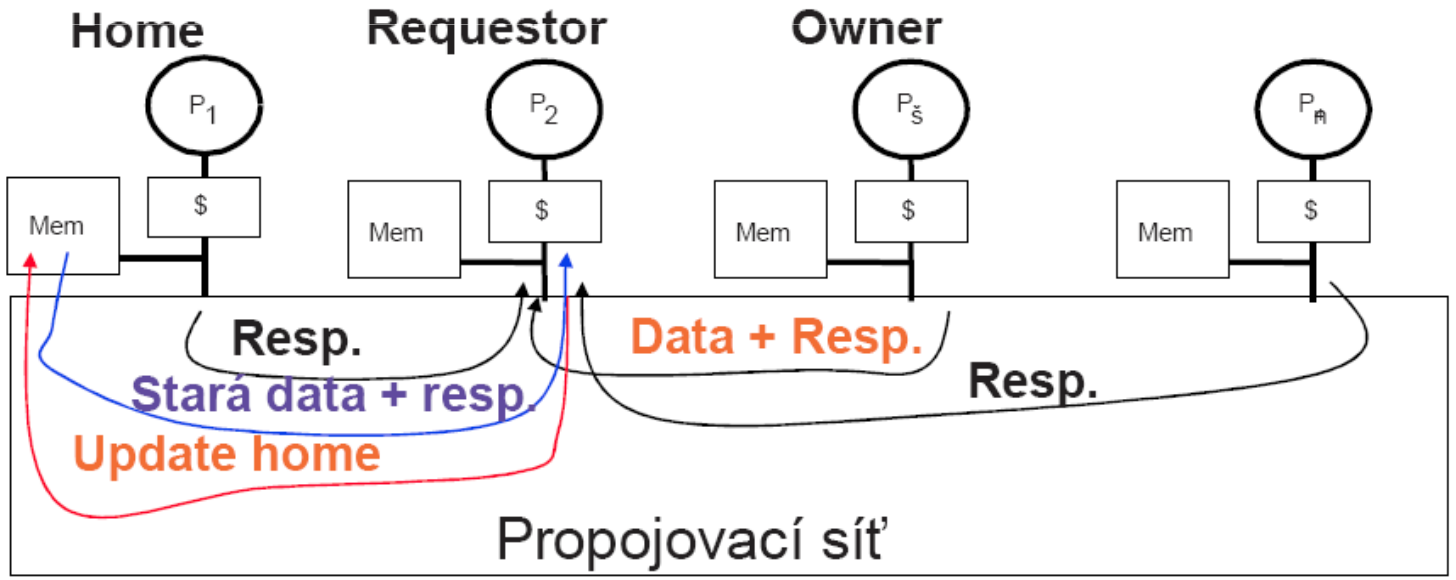
Pokračování příkladu

- Slídící P1, P3 a P4 pošlou **Resp.** (Snooping odpověď) žadateli P2. Pokud mají blok ve stavu Exclusive nebo Shared – mohou poslat i data. P2 nemusí čekat na data z hlavní paměti, pokud mu přijdou data dříve od jiného procesoru.
- Hlavní paměť u domovského uzlu pošle Data P2.
- P2 potvrdí domovskému uzlu Done (dokončení operace, umožňuje např. zastavit čtení z hl. paměti, pokud P2 už získal data od někoho dříve).



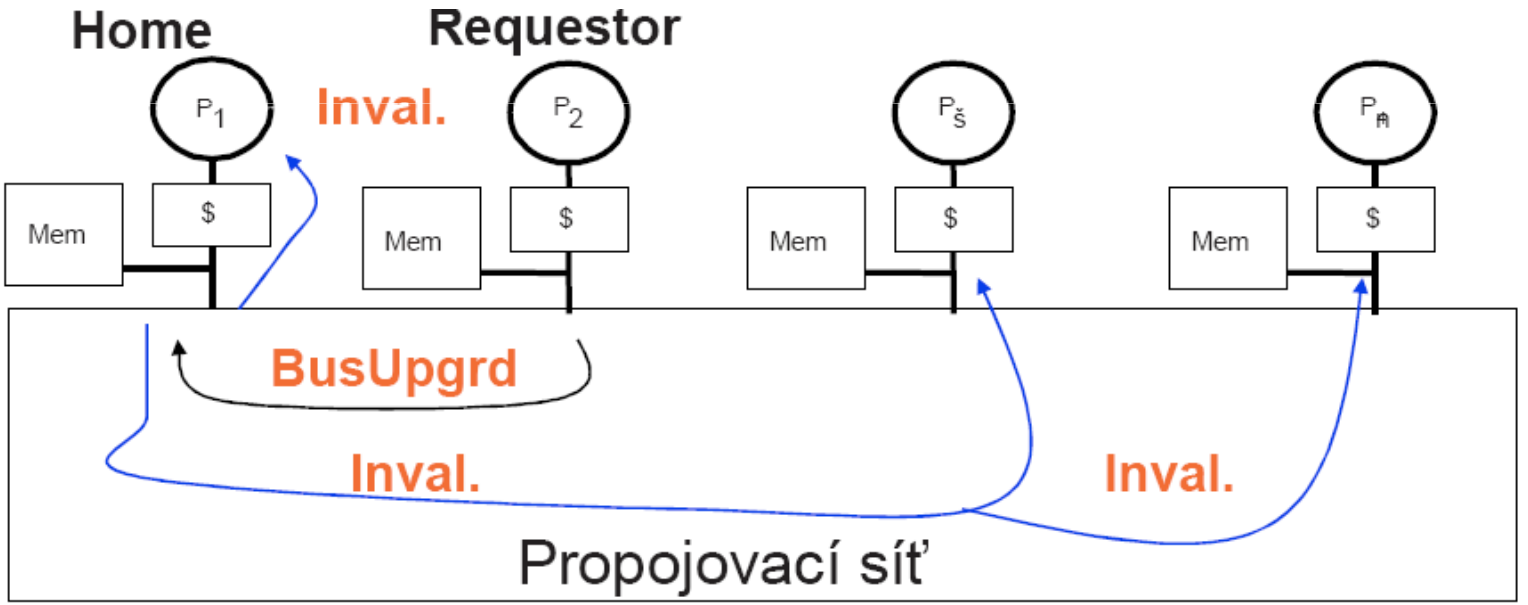
Co když má P3 hodnotu Modified (MESI protokol)?

- P3 pošle aktuální hodnotu bloku Data žadateli P2.
- P2 musí ignorovat Data (Stará data) z hlavní paměti, dokud nedostane Resp. (Snooping odpověď) od všech procesorů !
- P2 pošle aktuální data domovskému uzlu (jen jde-li o MESI protokol, Pokud jde o MOESI, nový stav bloku v P3 bude Owned a domovský uzel se nebude obnovovat (Update)).



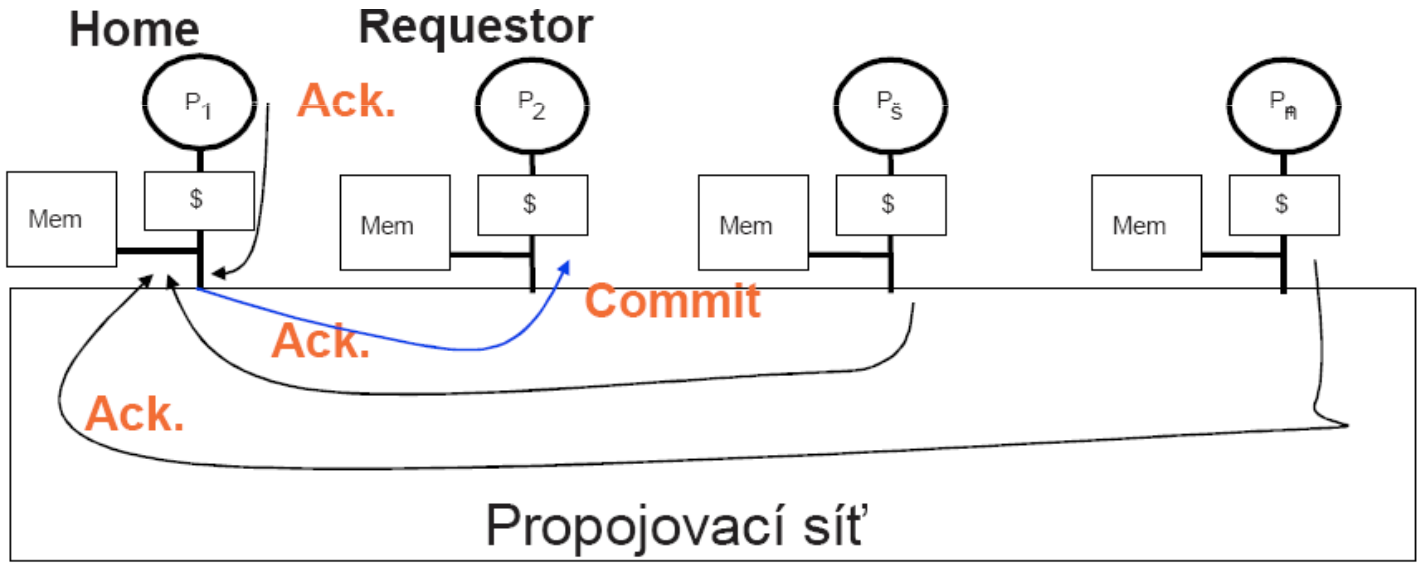
P2 chce zapisovat do bloku (Shared) - BusUpgrd

- P2 pošle žádost BusUpgrd směrem k domovskému uzlu P1
- Domovský uzel pošle Invalidate ke všem Pi v systému.



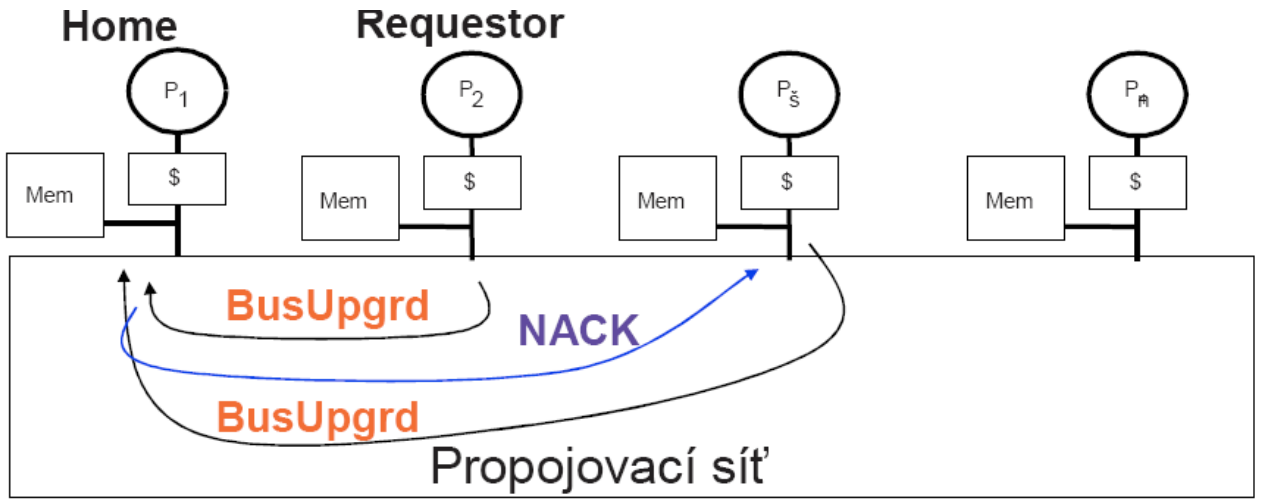
P2 chce zapisovat do bloku (Shared) - BusUpgrd

- P1, P3 a P4 pošlou potvrzení zneplatnění (Ack.) Domovskému uzlu
- Domovský uzel pošle potvrzení (Commit) žadateli P2 (pokud šlo o BusRdX pak i data)



Konflikt: P2 a P3 pošlou současně BusUpgrd (BusRdX)

- Domovský uzel je zodpovědný za serializaci přístupů k paměť. místu. Jeden z požadavků vyhraje, druhému je odeslán NACK (Negative Acknowledgement, příznak: opakujte akci později). V našem případě to později zkusí procesor P3.
- Podobně jsou řešeny i případy:
  - 1. BusRd přijde v okamžiku, dokud nedorazily všechny ACK na zneplatnění od předchozího BusUpgrd / BusRdX
  - 2. BusUpgrd/ BusRdX dorazil dokud nedorazily všechny ACK na zneplatnění od předchozího BusUpgrd / BusRdX



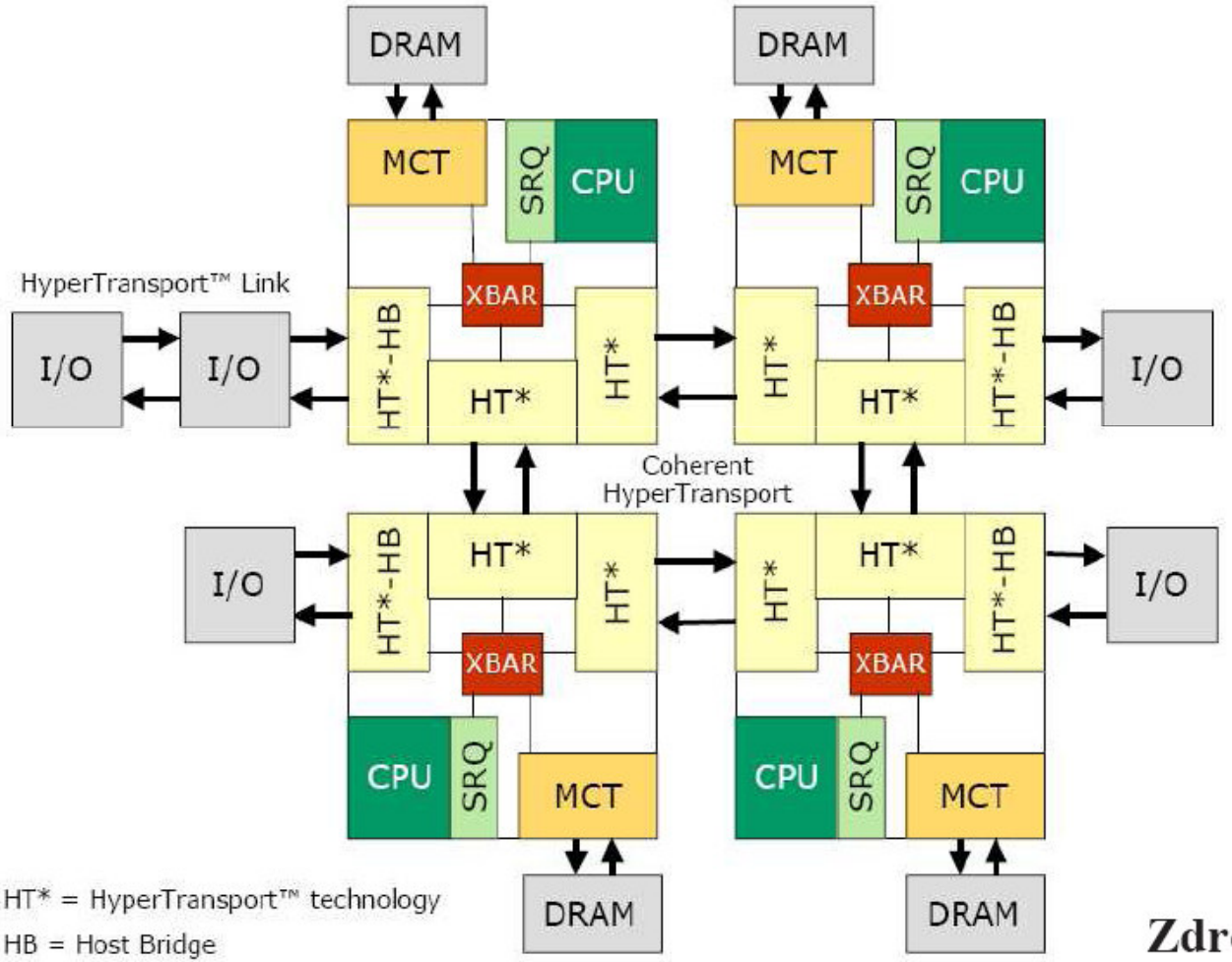
Všimli jste si?

- Kde na předchozích slajdech jsme hovořili o konzistenci?

Distribuované koherentní systémy na bázi Broadcastu

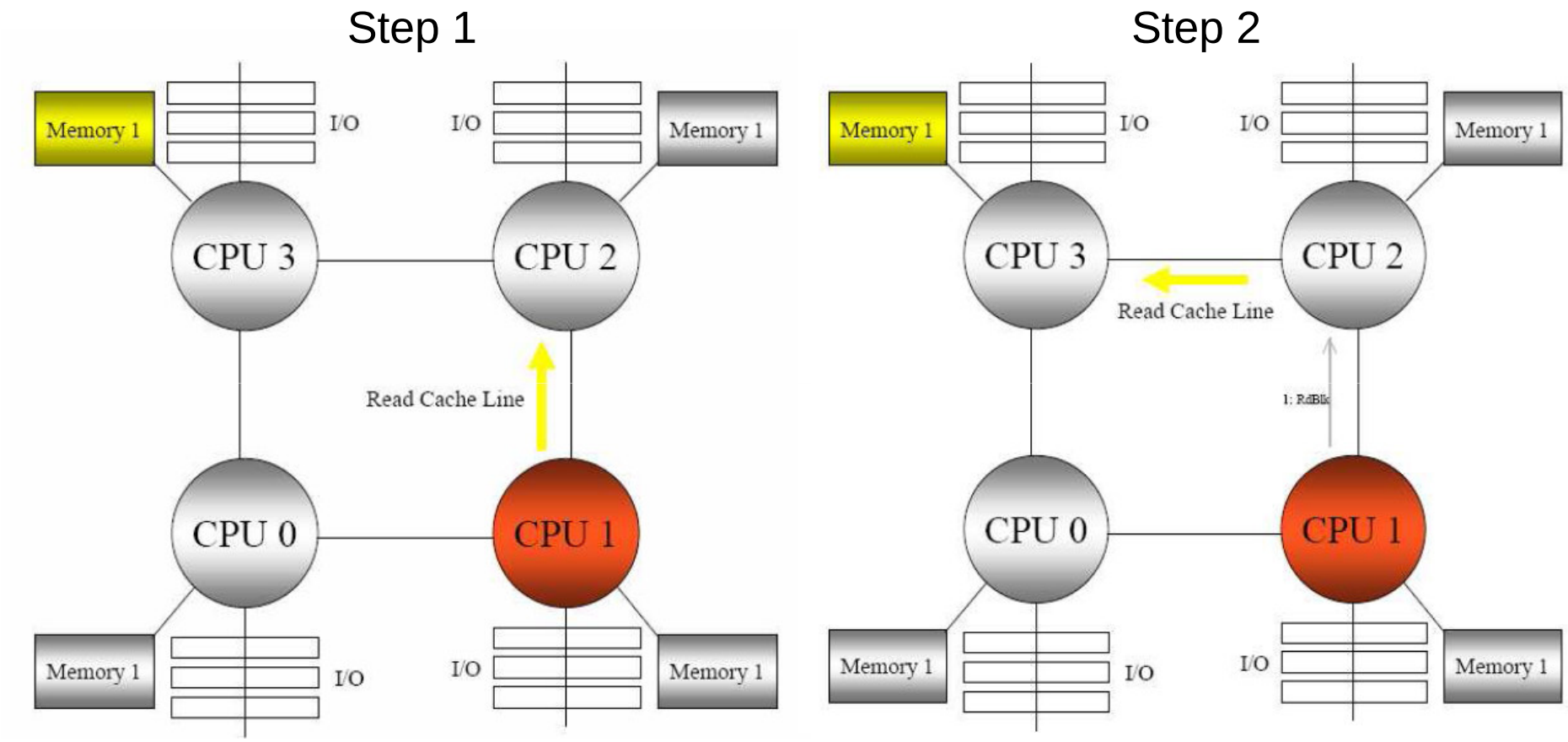
- Přirozené rozšíření sběrnicových systémů. Nevyžadují se velké změny v návrhu procesorů/čipových sad.
- Nevýhodou je obtížná škálovatelnost. Ale je o něco lepší než u sběrnic.
- Koherence zajištěna serializací přístupů čtení/zápis k danému paměťovému místu u Domovského uzlu.
- Sekvenční konzistence je splněna, pokud:
  - Procesory pouští Read/Write v programovém pořadí.
  - Procesor nespustí další paměťovou operaci, dokud nedostal Commit od Domovského uzlu na předchozí operaci Write.
  - Domovský uzel nedovolí aby následující Read dostalo novou hodnotu, dokud předchozí Write nedoběhl vzhledem ke všem procesorům (nedorazily ACK od všech procesorů) A pravidla pro buffery mezi SP (cache) – odpověď na BusRd nesmí předběhnout Invalidation, které přišly dříve.

Takhle vypadá AMD Quad – Coherent HyperTransport

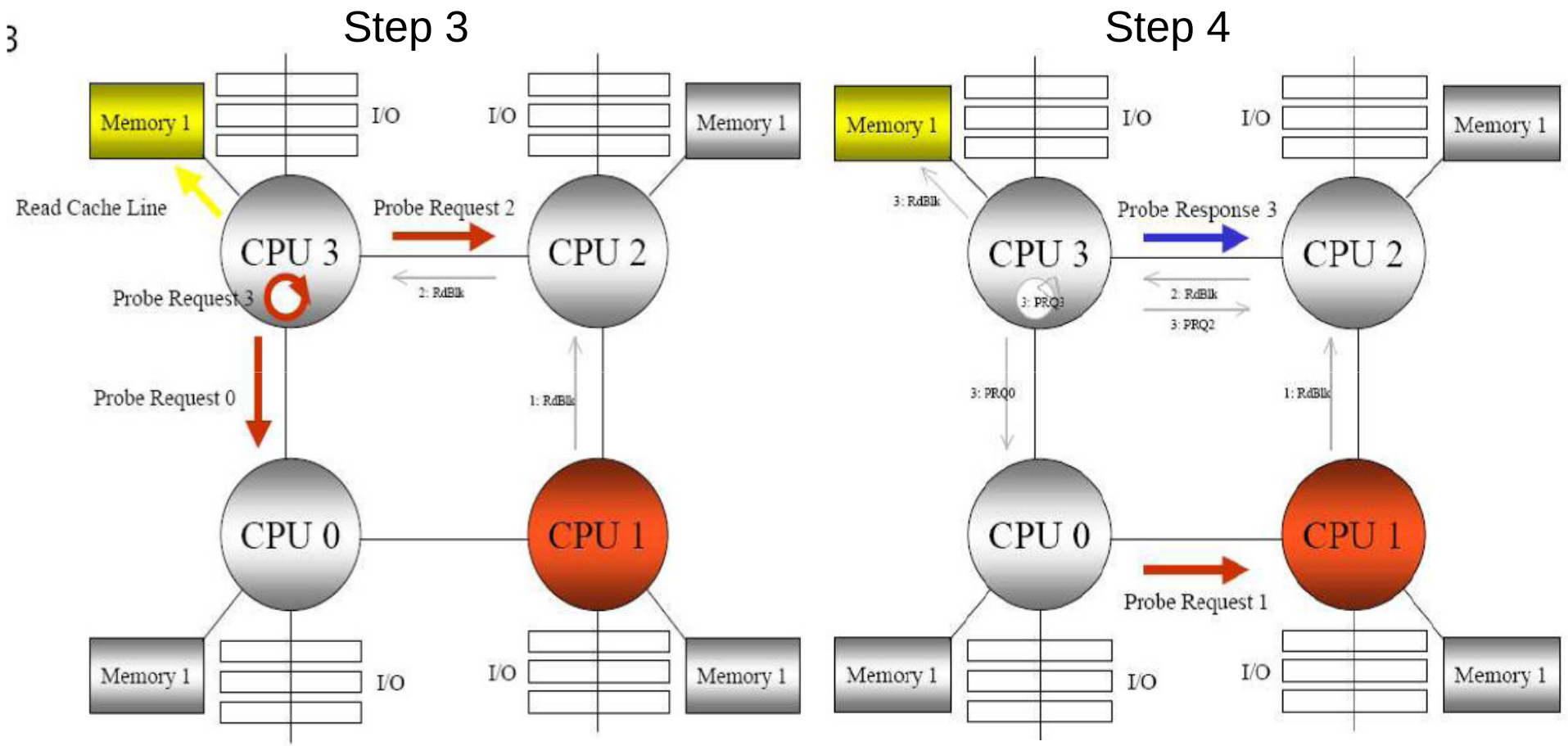


Zdroj: AMD

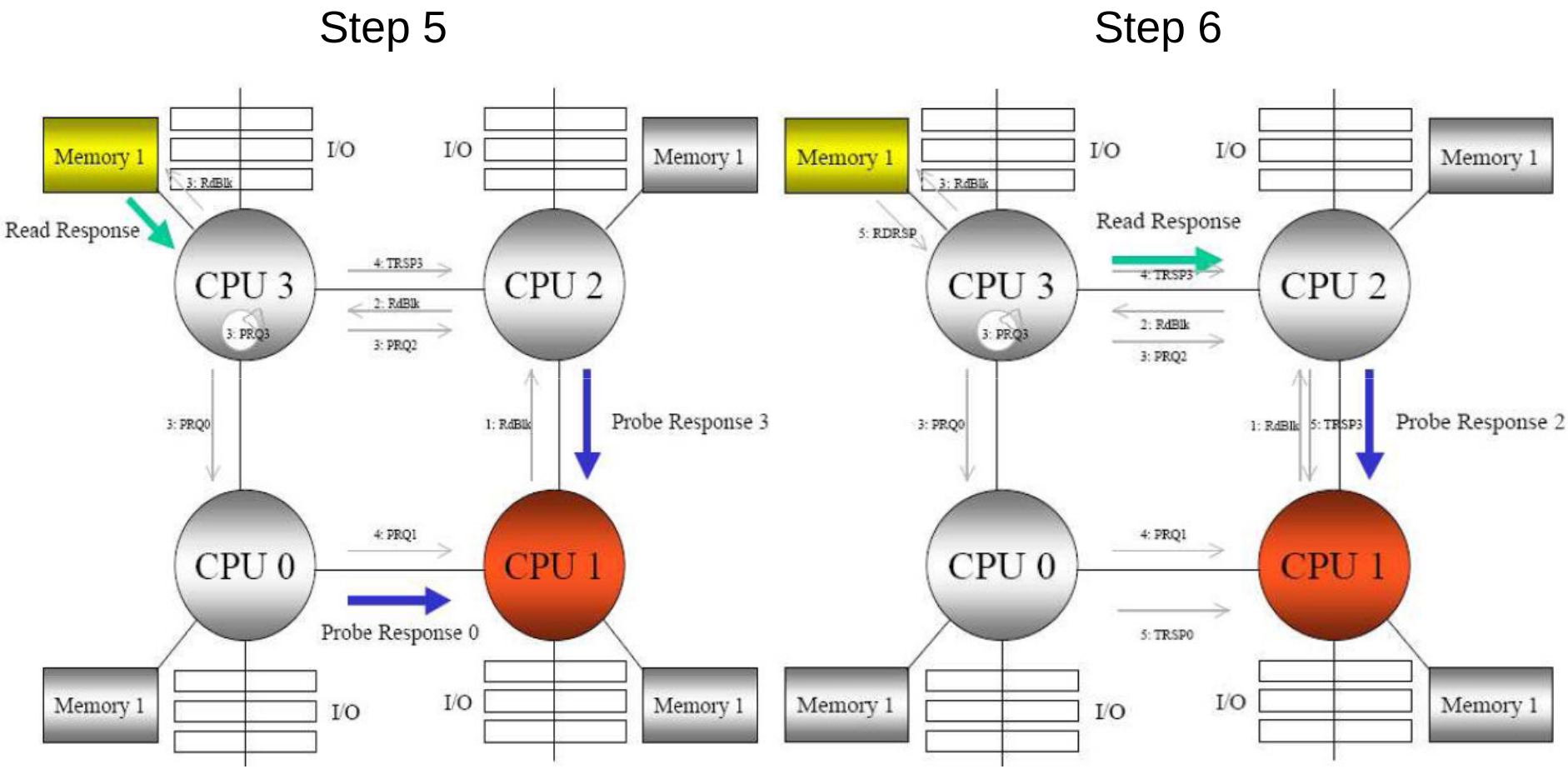
Příklad: CPU1 čte položku, která je domovská v CPU3



CPU1 čte položku, která je domovská v CPU3 - pokračování

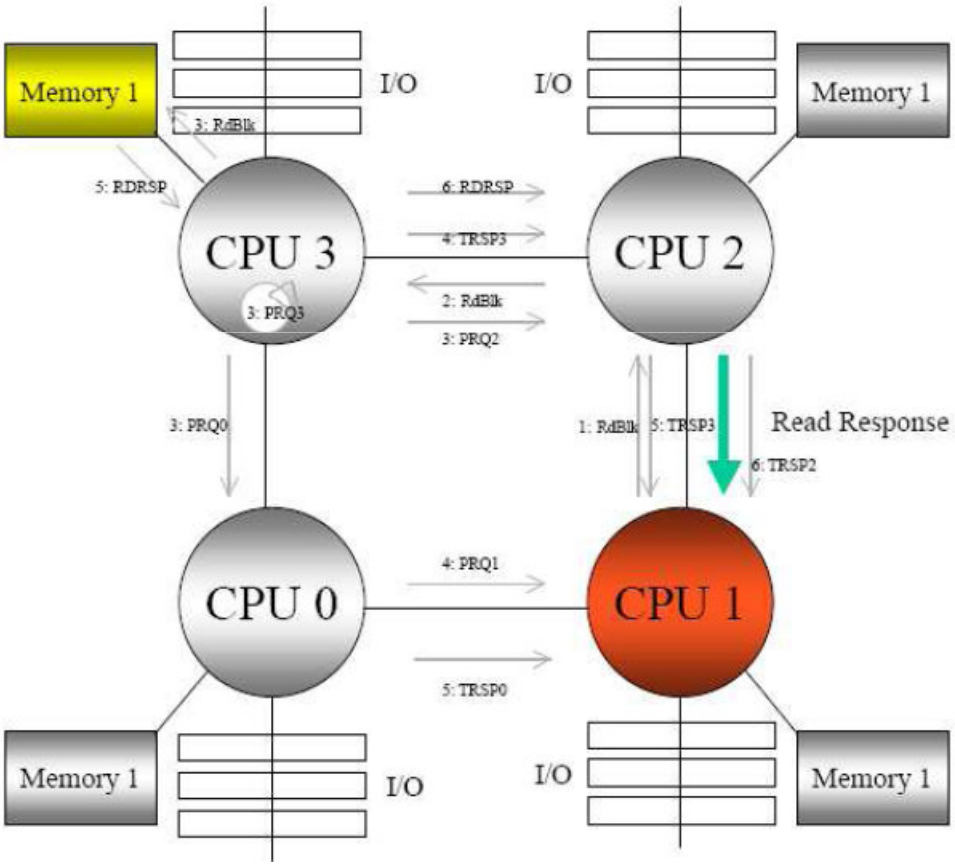


CPU1 čte položku, která je domovská v CPU3 - pokračování

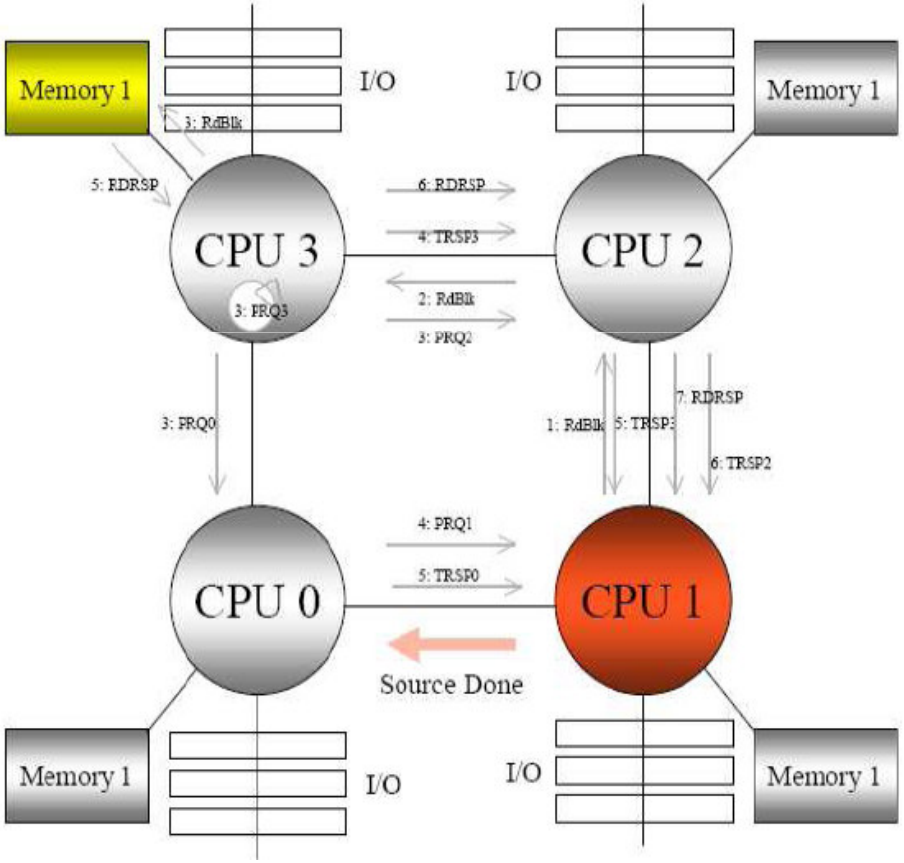


CPU1 čte položku, která je domovská v CPU3 - pokračování

Step 7

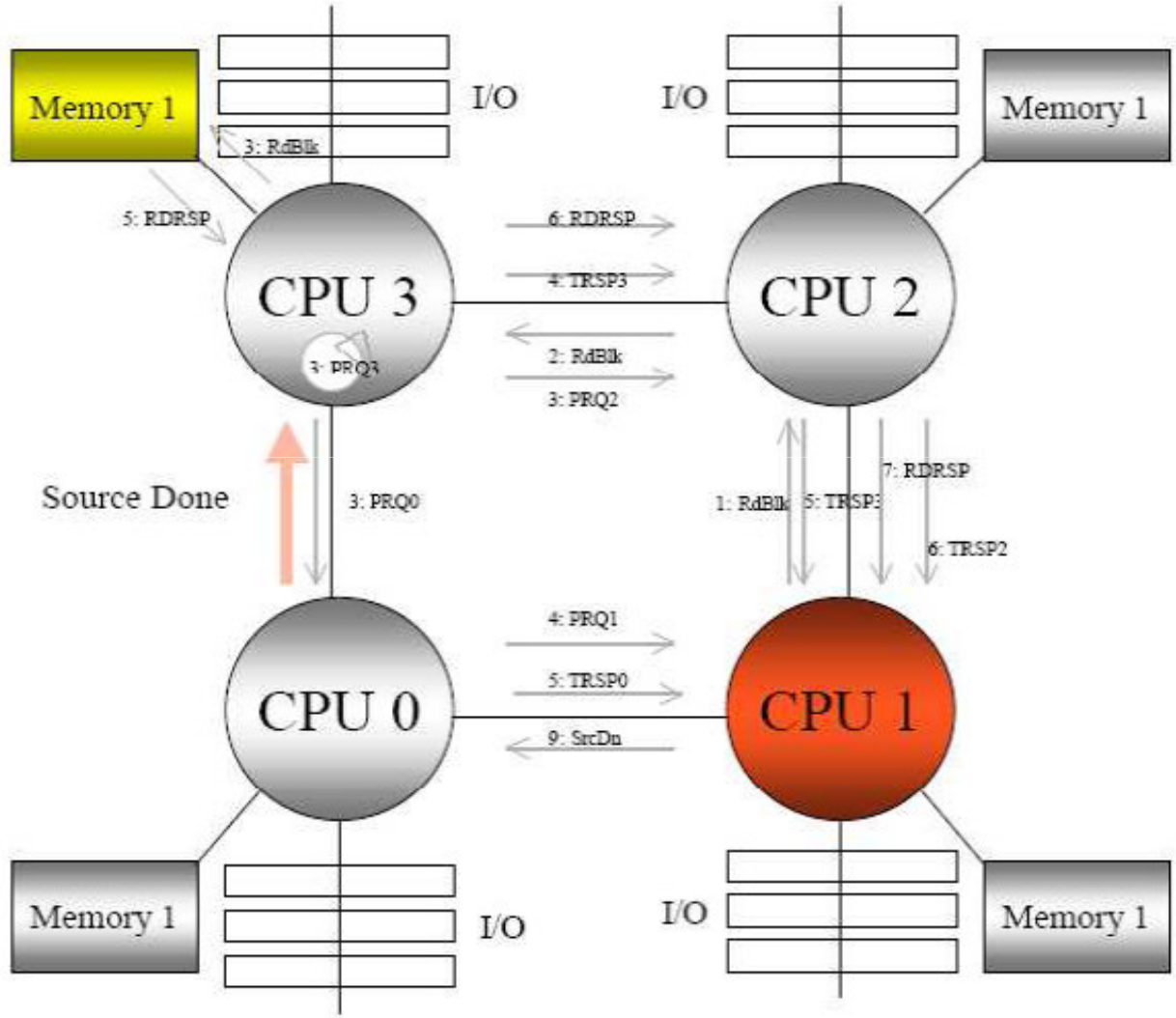


Step 8

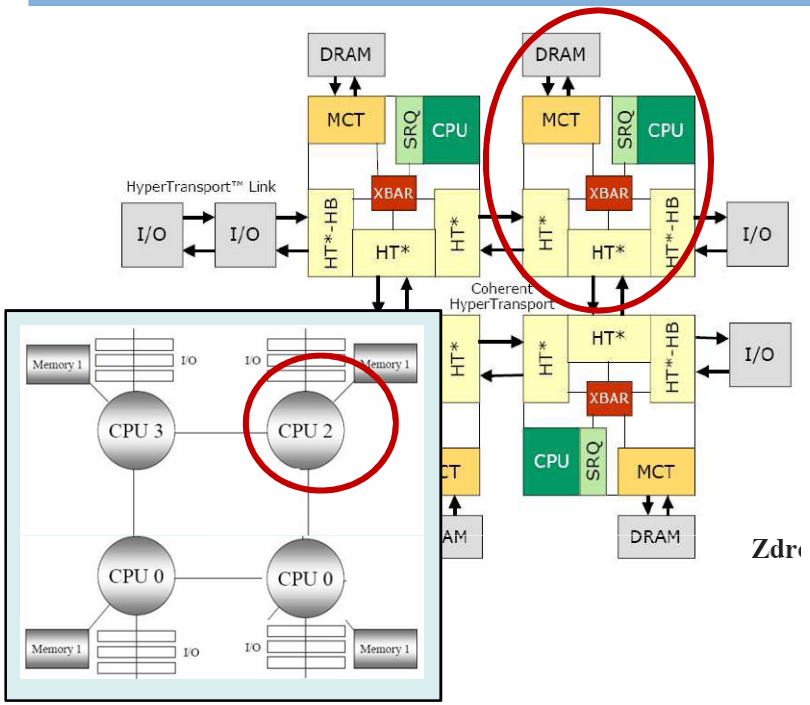


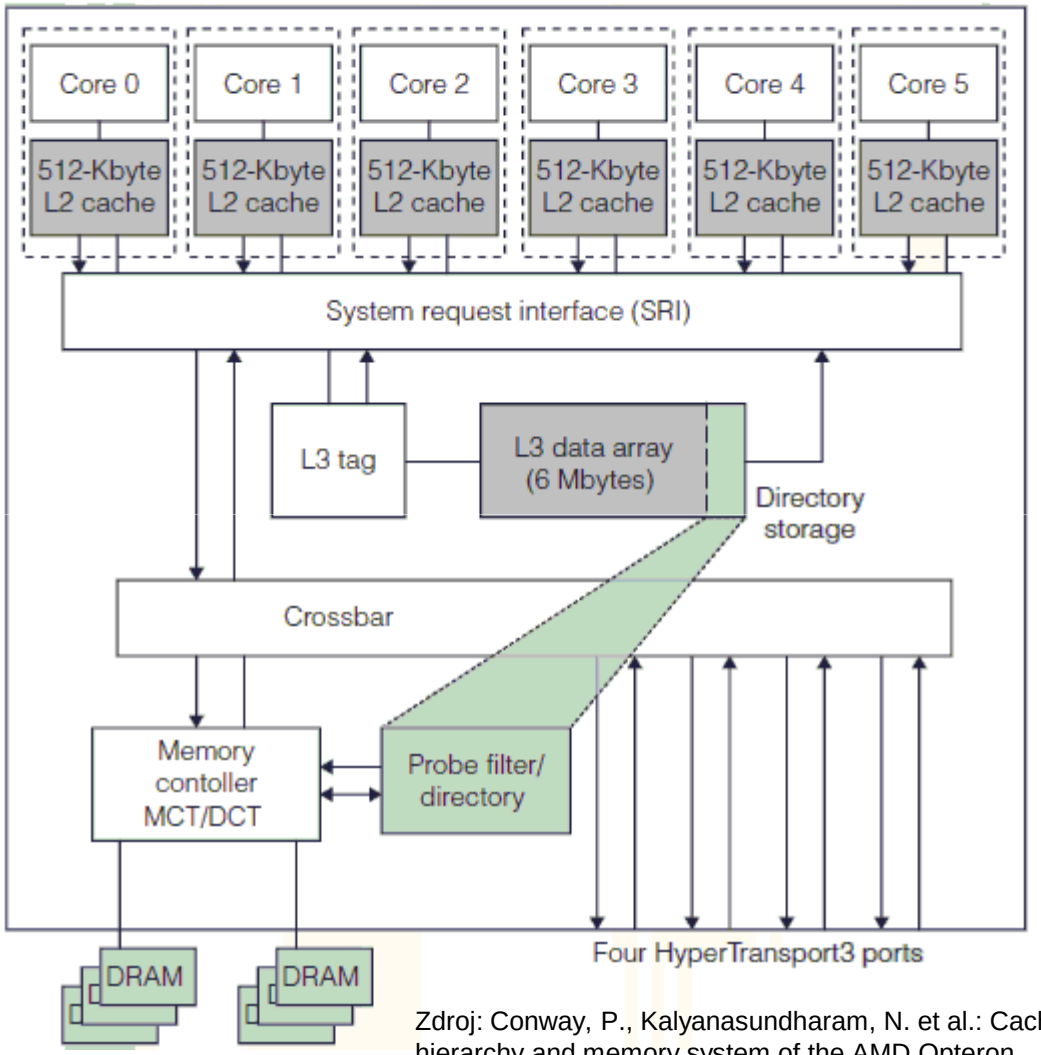
CPU1 čte položku, která je domovská v CPU3 - dokončení

Step 9



Takhle vypadá AMD Quad – Coherent HyperTransport





- Probe Filter (HT Assist) - funguje tak, že používá část L3 cache jako directory cache v níž monitoruje již nakešované řádky. Místo generování množství žádostí (cache probes), procesor prohledá tuto část L3 cache.

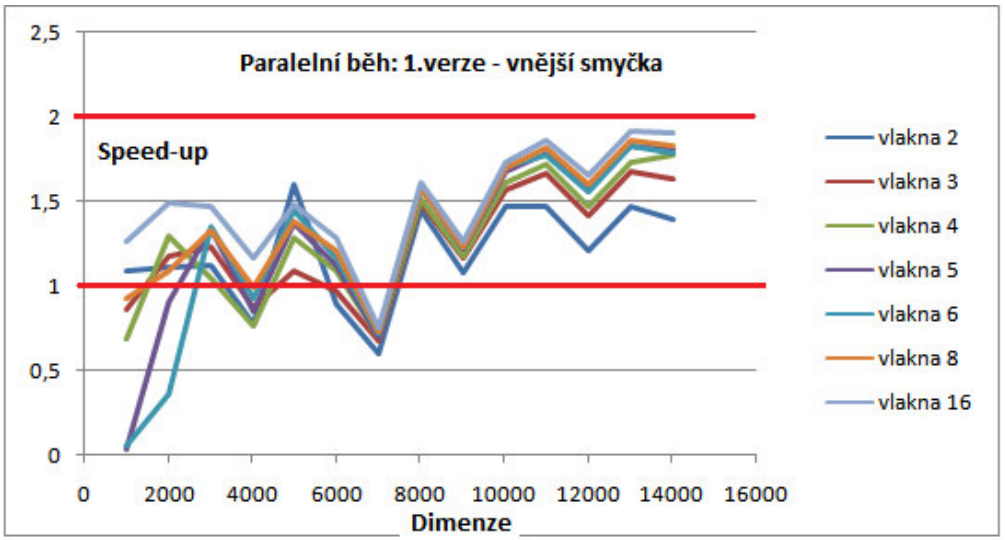
Zdroj: Conway, P., Kalyanasundharam, N. et al.: Cache hierarchy and memory system of the AMD Opteron processor, IEE Micro, March/April 2010.

Paměťová konzistence prakticky

```
void matrix_vector_mult_basic(double *A, double *x, double *y, int M, int N)
{
    /*
    Funkce pocita soucin matice a vektoru: y = A*x
    A - matice dimenze MxN ulozena po sloupcich
    x - vektor dimenze N
    y - vektor dimenze M
    */
    int ii, jj;
    for(ii=0;ii<M;ii++){
        y[ii] = 0.0;
        for(jj=0;jj<N;jj++)
            y[ii] += A[ii+jj*M]*x[jj];
    }
}
```

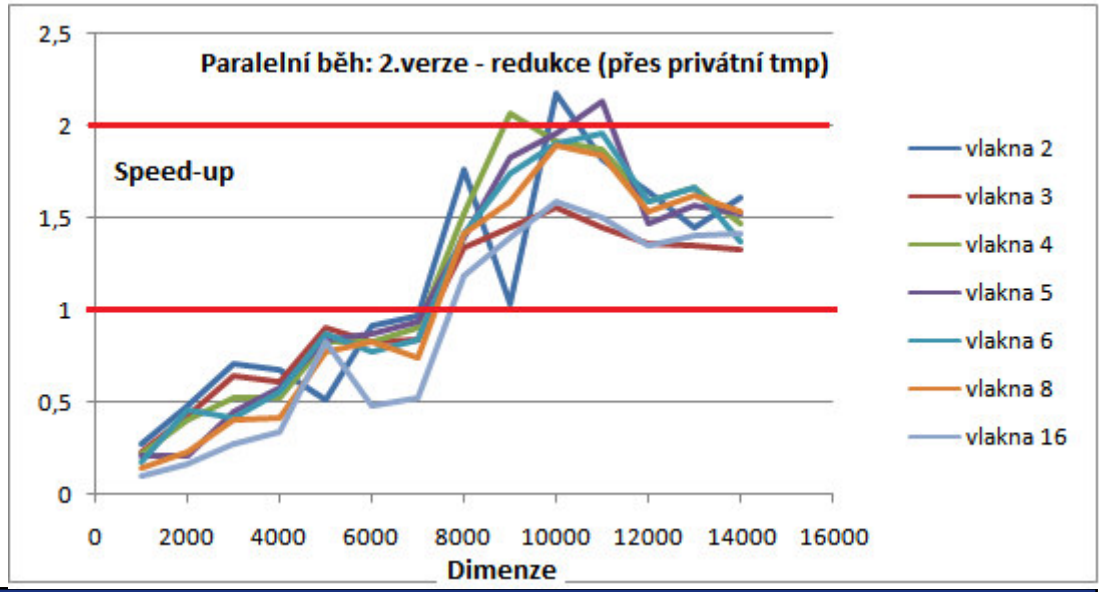
Paměťová konzistence prakticky

```
void matrix_vector_mult_parallel_first(double *A, double *x, double *y, int M, int N)
{
    int ii, jj;
    #pragma omp parallel for private(jj)
    for(ii=0;ii<M;ii++){
        y[ii] = 0.0;
        for(jj=0;jj<N;jj++)
            y[ii] += A[ii+jj*M]*x[jj];
    }
}
```



Paměťová konzistence prakticky

```
void matrix_vector_mult_parallel_sec(double *A, double *x, double *y, int M, int N)
{
    int ii, jj;
    double tmp;
    for(ii=0;ii<M;ii++){
        tmp = 0.0;
        #pragma omp parallel for reduction(+:tmp)
            for(jj=0;jj<N;jj++)
                tmp += A[ii+jj*M]*x[jj];
        y[ii] = tmp;
    }
}
```



Paměťová konzistence prakticky

```
void matrix_vector_mult_parallel_third(double *A, double *x, double *y, int M, int N)
{
    int ii, jj;
    double *tmp;
    if((tmp=(double*)malloc(M*sizeof(double)))==NULL)
        printf("Zde by se volalo klasicke nasobeni..\n");
    else{
        #pragma omp parallel for
            for(ii=0;ii<M;ii++)
                y[ii] = 0.0;

        #pragma omp parallel for private(ii,jj)
            for(jj=0;jj<N;jj++)
                for(ii=0;ii<M;ii++)
                    #pragma omp atomic
                        y[ii] += A[ii+jj*M]*x[jj];
        }
        free(tmp);
}
```

Paměťová konzistence prakticky