Sander Lenaerts
11/02/2015
Rolnr: 500201
Begeleider: Mattias De Wael
Promotor: Jennifer Sartor

Vrije Universiteit Brussel
3de Bachelor Computerwetenschappen
Faculteit Wetenschappen en Bio-ingenieurswetenschappen

Cache simulaties

1. Abstract

Keuzes op vlak van cache implementatie worden gemaakt door hardware fabrikanten. Voor mijn thesis wil ik het mogelijk maken met deze implementatie te experimenteren. Ik wil aan studenten de mogelijkheid geven om eenzelfde programma te laten lopen op verschillende caches. Voor men zo een simulatie kan maken, moet men onderzoeken welke mogelijkheden er zijn en aan welke eigenschappen een simulatie moet voldoen voordat hij gebruikt kan worden om niet één cache maar alle mogelijke caches te simuleren.

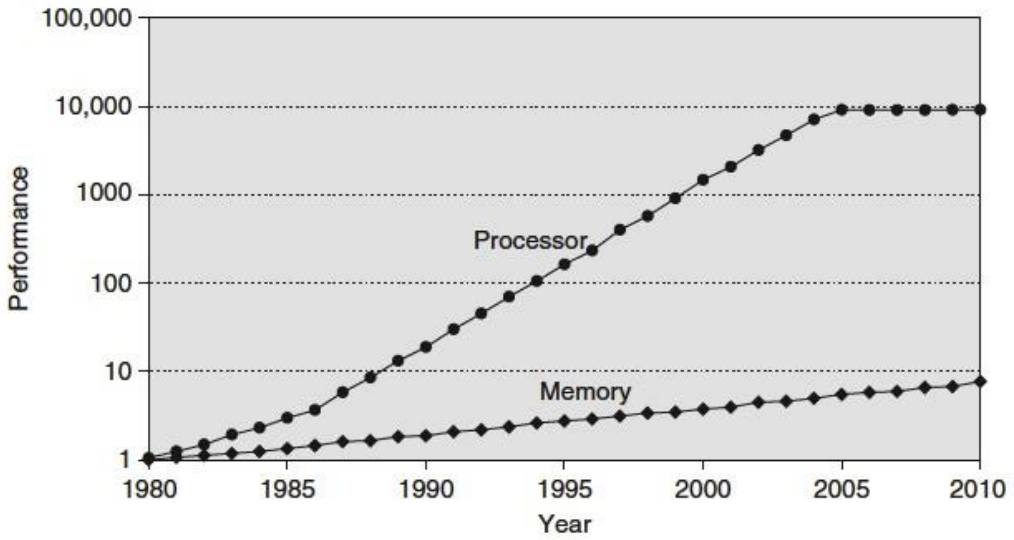
2. Introductie

Zowel CPU als memory access wordt elk jaar sneller, maar CPU snelheid evolueert veel sneller dan memory access snelheid. Dit zorgt voor een kloof en zou een limiet kunnen stellen aan de snelheid van computers. Caches gaan proberen deze kloof te overbruggen door data, die in de toekomst nodig zal zijn, al dichter bij de CPU te brengen en zo de access tijd te verminderen.

Als eerste gaan we in sectie drie en vier kijken hoe caches deze kloof overbruggen. Vervolgens gaan we in sectie vijf bespreken hoe de performantie van caches gemeten wordt om zo in sectie zes de cache optimalisaties beter te kunnen begrijpen. Hierna gaan we in sectie zeven multicore architecturen onder de loep nemen om als laatste te kunnen komen tot het algemeen verloop van cache operaties in sectie acht.

3. Het waarom van caches

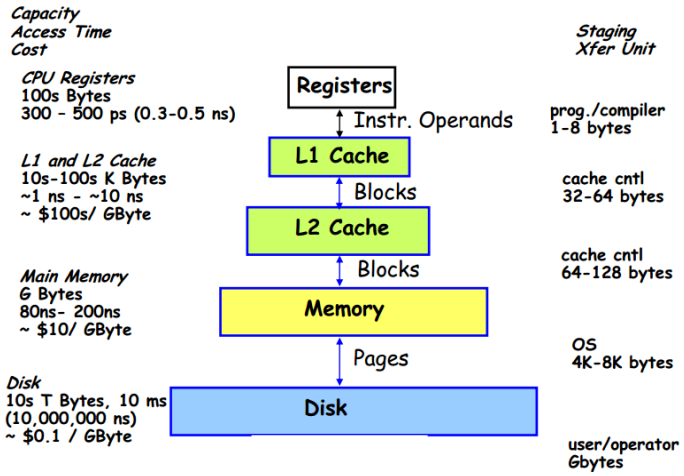
De wet van Moore stelt dat het aantal transistors in een CPU elke twee jaar verdubbelt. Simpel gezegd betekent dit dat processoren elke twee jaar ongeveer dubbel zo veel werk kunnen uitvoeren. De tijd die nodig is om data op te vragen en terug op te slaan in het geheugen van de computer daalt wel, maar dit aan een veel lager tempo. Tot de jaren 2005 stegen processor snelheden jaarlijks ongeveer met 55%, tot het heden stegen RAM snelheden maar met 10% per jaar. Er is dus sprake van een kloof tussen processor snelheid en geheugen access snelheid. Dit betekent dat we zonder extra zorg processor kracht verspillen, we kunnen immers onmogelijk genoeg data ophalen uit het main memory om de processor volledig te verzadigen.



Prestatie kloof tussen CPU en geheugen
bron: John L. Hennessy, Stanford University, and David A. Patterson
Memory Hierarchy Design - Part 1. Basics of Memory Hierarchies
<http://www.edn.com/design/systems-design/4397051/2/Memory-Hierarchy-Design-part-1>

Ten tweede is er nog het probleem van kost. Mensen willen computers die oneindig snel (processor), oneindig groot (geheugen) en oneindig goedkoop zijn. Hier is het probleem dat er enkel goedkoop groot geheugen of duur snel geheugen beschikbaar is op de markt.

Zoals gezegd gaan caches proberen deze problemen te overbruggen. Dit door een connectie te vormen tussen het trage (goedkope) main memory en de snelle (dure) CPU registers; en dit in de vorm van een piramide hiërarchie.



Bron: Dmitri Strukov Introduction to Computer Architecture - II 2013
< http://www.ece.ucsb.edu/~strukov/ece154bSpring2013/week1.pdf> slide 40

Elk niveau dichter bij de CPU wordt duurder, kleiner en sneller. Door een slimme implementatie van caches wordt het geheel snel, groot en goedkoop. Wat die slimme implementaties nu juist zijn, wordt later in deze paper besproken.

4. Het hoe van caches

Simpel gezegd moeten caches proberen data naar de processor te brengen nog voor de processor deze nodig heeft; ze gaan dus aan *latency hiding* doen. Caches zijn kleiner dan het geheugen van de computer. Dit betekent dat niet alle data uit het geheugen tegelijkertijd te vinden kan zijn in de cache. Als men data gaat opzoeken in een cache en deze data vindt, heeft er een zogenaamde *cache hit* plaatsgevonden. Als de data afwezig is, kan hij ook niet gevonden worden en zal er een *cache miss* plaatsvinden.

Het is niet mogelijk om met honderd procent zekerheid te voorspellen welke data in de toekomst nodig zal zijn. Als leidraad kan het principe van lokaliteit gebruikt worden. Dit principe stelt dat programma's niet gewoon willekeurig data en instructies ophalen uit het geheugen. Programma's gaan regelmatig blijven 'plakken' in een klein deel van de data. Het principe van lokaliteit gaat uit van een 90/10 regel: in 90% van de tijd zal de computer bezig zijn met 10% van de code. Uit dit principe van lokaliteit zijn twee regels af te leiden:

1. Rule of *spatial locality*:
   Data in de buurt van de data die we nu gebruiken, heeft een grotere kans heeft om in de toekomst opgevraagd te worden. (Bijvoorbeeld sequentieel over een lijst lopen.)
2. Rule of *temportal locality*:
   Data die we nu gebruiken, heeft een grotere kans om in de toekomst opnieuw gebruikt te worden. Bijvoorbeeld een counter / iterator.

Een voorbeeld dat geen van deze regels volgt is het opstarten van een nieuw programma.

Binnen computers worden om praktische redenen bits niet één per één rond gestuurd, ze worden gegroepeerd in zogenaamde *woorden*. Woorden zijn dan ook de kleinste eenheden van adresseerbare geheugen die gebruikt wordt binnen computers. Binnen caches gaan we niet werken met individuele woorden, dit zou te veel overhead genereren. In de plaats daarvan werken we met *blokken* zijnde groeperingen van *woorden*. Als één woord van een blok opgevraagd wordt, zegt spatial locality dat de andere woorden in de toekomst ook opgevraagd gaan worden.

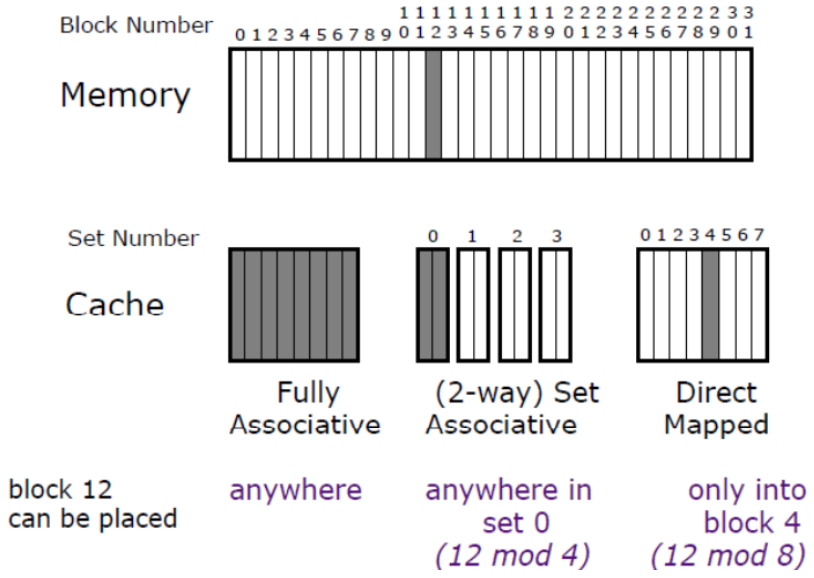
In de praktijk kan de werking van een specifieke cache beschreven worden op basis van vier karakteristieken. Elk van deze karakteristieken geeft antwoord op één van de centrale vragen rond caches:

- *Block placement*: waar zet ik een zeker blok in de cache?
- *Block identification*: waar zoek ik een zeker blok in de cache?
- *Block replacement*: welk blok doe ik weg na een cache miss?
- *Write strategy*: wat gebeurt er tijdens een write-operatie?

4.1 Plaatsing van blokken binnen de cache.

Er zijn drie grote manieren te onderscheiden om blokken te plaatsen binnen een cache.

1) fully associative:
   Een vrije keuze uit heel de cache. De plaatsing van het blok heeft geen relatie met het blok zelf. Om een blok te vinden moet er sequentieel gezocht worden door heel de cache, dit zal om performantieredenen altijd parallel gebeuren.
2) set associative:
   Een vrije keuze uit een gelimiteerde set. De cache wordt verdeeld in verschillende sets. Een blok wordt aan de hand van zijn bloknummer opgeslagen in één van deze sets. Om een blok te zoeken binnen zijn set, moet er sequentieel gezocht worden door zijn set; wederom zal dit parallel gebeuren.
3) direct mapped/non-associative:
   Elk blok heeft exact één plaats binnen het geheugen. Om de positie van een blok terug te vinden wordt de formule A module B gebruikt waarbij A het blok adres is en B het maximaal aantal blokken in de cache.



Bron: Dmitri Strukov Introduction to Computer Architecture - II 2013
< http://www.ece.ucsb.edu/~strukov/ece154bSpring2013/week2.pdf> slide 11

De graad van associativiteit heeft een grote invloed op performantie en stroomgebruik. In een direct mapped cache hoeft maar één plaats gecontroleerd te worden om te weten of een zekere blok te vinden is in de cache, toch is dit niet noodzakelijk de meest performante cache.

Door parallel door de set of hele cache (fully associative) te zoeken kan het controleren van één plaats even snel gemaakt worden als het controleren van alle plaatsen. Dit parallel controleren zorgt wel voor een veel hoger stroomgebruik en kost. De hardware nodig voor dit parallel controleren is immers veel uitgebreider.

In een direct mapped cache is maar één plaats om een blok te zetten, dit betekent dat als de processor gaat alterneren tussen blokken die naar dezelfde plaats mappen, er constant een cache miss plaats vindt. In een cache met hogere associativiteit zal de replacement strategy dit proberen vermijden (zie later). Binnen caches met een hogere associativiteit kan de extra vrijheid gebruikt worden om het meeste te halen uit het principe van lokaliteit.

X-way associative is een manier om uit te drukken dat er maximaal X blokken tegelijk in een set passen. Bijvoorbeeld:

- 2-way associative zijn sets van 2 blokken, dus een blok kan op één van twee mogelijke plaatsen kan terecht komen.
- 8-way associative zijn sets van 8 blokken, dus een blok kan op één van acht mogelijke plaatsen kan terecht komen.

4.2 Het opzoeken van een blok in de cache.

Caches zijn kleiner dan het main memory van computers. Naast de oorzaak voor cache misses, betekent dit ook dat verschillende blokken in dezelfde set terecht moeten komen (duiventil principe). Dit heeft twee belangrijke gevolgen:

1) We kunnen er niet van uitgaan dat een gezocht blok altijd in zijn set zal zitten.
2) Sets moeten zelf het adres onthouden van het blok dat ze bevatten.

Een blok zoeken in een set associative of direct mapped cache, betekent het set nummer berekenen en daarna het opslagen adres vergelijken met het gezochte adres. Een blok zoeken in een fully associative cache betekent alle opgeslagen adressen vergelijken met het gezochte adres, wat voor performantie redenen in parallel gebeurt.

4.3 Verwijderen van blokken uit de cache.

Op het moment dat het blok met blok nummer A niet gevonden is in de cache (ook cache miss genoemd), moet blok A opgezocht worden in onderliggende hiërarchie. Nadat blok A gevonden is, wordt dit opgeslagen in de cache en pas daarna doorgegeven aan het bovenliggende niveau. Dit betekent dat een blok gekozen moet worden om vervangen te worden door blok A. Ook hier zijn verschillende mogelijkheden. Herinner dat de associativiteit van een cache bepaalt op welke plaatsen een blok in de cache mag komen te staan. Als we een blok willen toevoegen in de cache, moeten we

één van deze plaatsen vrijmaken. De replacement strategy is dus afhankelijk van de associativiteit van de cache.

Bij een direct mapped cache is er geen keuze; alle blokken hebben immers maar één mogelijke plaats binnen het geheugen. Het blok dat op die plaats staat moet dan ook worden verwijderd.
In een Fully of set associative cache zijn er verschillende mogelijke plaatsen om een blok in de cache op te slaan. Er zijn dus ook meerdere blokken waaruit we er één moeten kiezen om plaats te maken voor het nieuwe blok. Er zijn vier mogelijkheden om de blokken te organiseren binnen de cache.

1. Random:
   Binnen de set wordt één blok random gekozen en verwijderd. Dit levert een uniforme allocatie tijd op, maar maakt geen optimaal gebruik van de regels van lokaliteit.
2. Least recently used:
   Temporal locality zegt dat het minst recent gebruikte blok de kleinste kans heeft om in de toekomst opnieuw gebruikt te worden. Dit is dus het blok dat we gaan verwijderen. Om deze techniek mogelijk te maken moet er voor elk blok de laatste allocatie tijd bijgehouden worden. Aangezien cache operaties kort op elkaar gebeuren (tot 1000 operaties per seconde) moet deze allocatie tijd vrij precies worden bijgehouden. Dit maakt deze methode op vlak van hardware heel duur en wordt daarom enkel gebruikt voor cache met een lage associativiteit zoals 2-way associative.
3. First in First out:
   Een first in first out queue benadert het gedrag van LRU en is hardware matig veel goedkoper. Dit is de eenvoudigste methode van allemaal en wordt gebruikt in cache met hoge associativiteit.
4. Not most resently used:
   Dit is een tweede manier om LRU te benaderen. In tegenstelling tot LRU en FIFO, die beiden redelijk wat geheugen kosten, wordt bij deze techniek enkel een bytevector bijgehouden. Er komt per blok in de set een bit bij. Deze bit gaat als vlag functioneren; als het overeenkomstig woord opgevraagd wordt, zal de vlag op true gezet worden. Als de laatste vlag op true gezet wordt, worden alle andere vlaggen terug op false gezet. Wanneer een cache miss plaatsvindt, gaan we random een blok kiezen uit alle blokken wiens vlag nog op false staat.

Temporal locality zegt ons dat een LRU cache het meest performante is, maar dit wordt regelmatig benaderd door goedkopere FIFO en NMRU caches.

4.4 Write operaties

De data in cache zijn kopieën van stukken data in het main memory. Dit brengt zoals altijd als we kopieën maken consistentie problemen met zich mee. Zolang enkel data gelezen zijn er geen problemen; als er daarentegen data aangepast wordt door middel van een write-operatie, moet op één of andere manier alle kopieën aangepast worden.

Hardware fabrikanten hebben twee grote stijlen om uit te kiezen: write-through of write-back.

1. Write-through:
   De data wordt aangepast in de cache en daarna in de hele onderliggende hiërarchie.
2. Write-back:
   De data wordt enkel aangepast in de cache en een vlag wordt bijgehouden om aan te geven dat de data aangepast is. Deze vlag wordt de *dirty bit* genoemd. Op het moment dat de *dirty data*

wordt verwijderd uit de cache, zal de nieuwe data worden geschreven naar onderliggende datastructuur. Hierdoor is een write-operatie afgerond als de bovenste cache de data heeft aangepast en de dirty flag heeft gezet en deze write-operaties zijn dus performanter.

Aangezien verschillende writes kunnen plaats vinden voordat data verwijderd wordt uit een cache en er enkel naar beneden wordt weggeschreven als de data verwijderd wordt, is er minder bandbreedte vereist.

Als er meerdere processoren zijn in eenzelfde machine zal elke processor zijn eigen cache krijgen. Hierdoor is er niet alleen een onderliggende hiërarchie maar ook een parallelle hiërarchie wat writes nog ingewikkelder maakt. Deze problematiek wordt in sectie zeven behandelt.

4.5 Cache misses tijdens een write operatie.

Een laatste manier om cache te onderscheiden is op vlak van write-allocatie. Hoe reageert de cache op een cache miss tijdens een write operatie. Hier zijn twee grote stromingen:

1. Write allocate:
   Laat de write operatie propageren doorheen de geheugen hiërarchie. Op elk niveau wordt er gekeken of het blok aanwezig is. Als het gezochte blok aanwezig is, wordt dit blok aangepast en het aangepaste blok wordt doorgegeven naar boven.
2. No-write allocate:
   De write operatie wordt doorgegeven naar beneden tot het gezochte blok gevonden is. Op dat niveau wordt de write uitgevoerd, maar de data wordt achteraf niet naar boven doorgegeven.

Alle combinaties van allocation policy and write policy zijn mogelijk, maar meestal gebruikt men write-back in combinatie met write allocate en write-through met no-write allocate.

Het voordeel van write-back is dat meerdere writes op éénzelfde blok binnen de cache maar één keer naar de onderliggende (tragere) architectuur geschreven wordt. Als de eerste write nu een cache miss veroorzaakt en de data wordt niet verplaatst naar de cache, zal de volgende write ook een cache miss veroorzaken. Om deze reden wordt write-back nagenoeg altijd met write-allocate gecombineerd.

In write-through caches wordt er voor een write zowiezo de hele structuur aangepast. Het kopiëren van de data naar de bovenste cache zal opeenvolgende writes niet sneller maken. Terwijl het kopiëren van data naar de bovenste cache de write zelf wel trager maakt; er moet immers gewacht worden tot het kopiëren voltooid is. Om deze reden wordt write-through praktisch enkel met no-write-allocate gecombineerd.

5. Cache performance meten

Voor we meer in detail gaan kijken naar manieren om de performantie van caches te optimaliseren, gaan we kijken naar hoe een cache zijn performantie geschat kan worden.

Een goede maat voor de performantie van cache is gemiddelde tijd om een geheugen locatie te laden (average memory access time).

$$average\ memory\ access\ time = hit\ time + miss\ rate * miss\ penalty$$

Hit time is de tijd die nodig is om een woord te zoeken als het aanwezig is binnen een cache.
Miss rate is de hoeveelheid opzoekingen in een cache die resulteren in een cache miss.
Miss penalty is de hoeveelheid tijd die nodig is om een woord op te zoeken in de hiërarchie, als deze niet te vinden is in de cache.

Voor out-of-order processoren dus processoren die kunnen blijven werken ondanks een cache miss is de miss rate berekenen een interessante opgave, men kan namelijk niet zomaar het aantal processor stalls nemen. Nochtans voor mijn simulatie (die in-order zal zijn) is dit niet de focus en wordt daarom ook genegeert.

6. Cache optimalisaties

Nu we een beter idee hebben van hoe we de performantie van caches kunnen berekenen, kunnen we enkele methoden bestuderen die gebruikt worden om de cache performantie te verbeteren. In het algemeen is cache performatie afhankelijk van drie parameters.

$$average\ memory\ access\ time = hit\ time + miss\ rate * miss\ penalty$$

Dit betekent dat als we cache willen optimaliseren, we moeten werken op drie vlakken:

1) Verlaag tijd nodig voor cache hit
2) Verlaag miss rate
3) Verlaag miss penalty

Al de optimalisaties hier besproken zijn hardware optimalisaties, dus optimalisaties gemaakt door de fabrikant, academisch interessant om mee te experimenteren maar een vast gegeven voor computerwetenschappers. In deze lijst wordt normaal ook compiler optimalisatie toegevoegd: men kan immers cache vriendelijke code ontwikkelen. Deze methode is voor computerwetenschappers eigenlijk interessanter omdat we zuiver met software de cache performance kunnen opdrijven. Toch heb ik gekozen deze methode niet te bespreken, juist omdat de cache niet wordt aangepast.

6.1 Miss rate verlagen

Alle klassieke cache optimalisaties werken op het verlagen van de miss rate. Voor we deze optimalizaties kunnen bestuderen, gaan we de cache misses opdelen in vier categorieën:

1. Compulsory misses:
   De eerste opvraging van een blok zal altijd een cache mis opleveren. Het blok kan immers nog niet in de cache zitten.
2. Capacity misses:
   Programma's die meer blokken nodig hebben dan tegelijk in een cache kunnen verblijven, veroorzaken capacity misses. Het zijn opvragingen naar blokken die recent verwijderd zijn. Als het programma blijft draaien, blijven deze zelfde misses terug keren.

3. Conflinct misses:
   Als er te veel blokken in eenzelfde set geplaatst worden kan een blok verwijderd worden en later opnieuw opgevraagd worden.
4. Coherency misses:
   In een multicore processor is er een cache per core, dit geeft consistency problemen, deze worden in de sectie zeven van de paper besproken.

Capacity misses zijn de lastigste om op te lossen, ze zijn immers enkel afhankelijk van de grootte van de cache. Als een cache veel te klein is om de data nodig voor een zeker programma, te bevatten, treedt er zogenoemd *thrashing* op. Thrashing betekent dat de computer constant bezig is data van cache niveau te laten wisselen en niet met het uitvoeren van echte taken. Als gevolg hiervan zal de computer werken aan de snelheid van het laagste (traagste) gedeelte van de memory hiërarchie.

6.1.1 Grotere caches

Door simpelweg de caches zelf groter te maken kunnen we het aantal capacity misses verlagen maar dit verhoogt de hit time omdat het sequentieel zoeken in cache langer duurt. Verder maakt deze methode de cache zelf duurder er moet immers meer hardware gemaakt worden.

6.1.2 Hogere associativiteit

Conflict misses zijn de makkelijkste om op te lossen. Hoe hoger de associativiteit van een cache, hoe meer blokken verspreid worden over de sets en dus hoe minder conflict misses. In fully associative caches zijn conflict misses volledig afwezig. Het nadeel van hogere associativiteit is dat de tijd nodig om een blok op te zoeken, groter wordt (in geval van normaal zoeken) of dat de hardware kost vergroot (in geval van parallel zoeken.) Er zijn immers meer woorden die vergeleken moeten worden voor we het juiste woord gevonden kunnen hebben.

Uit statistisch onderzoek zijn twee ruwe regels met betrekking tot associativiteit afgeleid, die kunnen helpen bij het ontwerpen van caches.

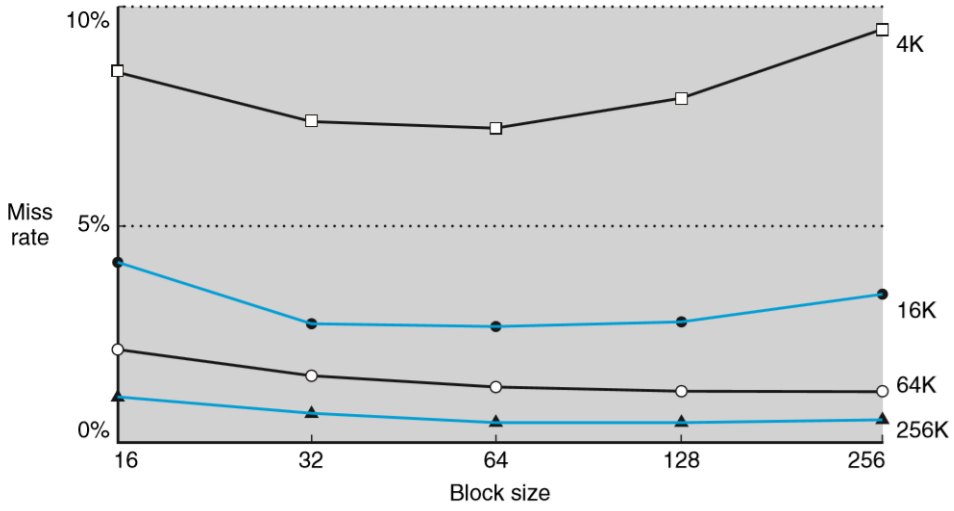
- Op vlak van miss rate zijn fully associative en 8-way associative perfect gelijk. In de praktijk worden fully associative caches vervangen door 8-way associative caches, deze zijn immers veel goedkoper en door de simpelere structuur zelfs iets performanter.
- Een direct mapped cache van grote N heeft dezelfde miss rate als een two-way associative cache met grote N/2. Hieruit kunnen we een trade off concluderen: men gaat direct mapped caches gebruiken als de cache dubbel zo groot maken minder kost dan de extra hardware die nodig is om twee cellen te controleren.

6.1.3 Grotere blokken

Door grotere blokken te gebruiken kunnen we het aantal compulsory misses laten dalen. Deze daling in misses is te wijten aan spatial locality van data. Een nadeel van grote blokken is dat ze ook een grote miss penalty veroorzaken; er moet immers meer data worden verplaatst voor een blok is gekopieerd naar een nieuw niveau in de hiërarchie. Zolang de impact op de miss penalty kleiner is dan de impact op de miss rate verlaagt de memory access time. Dit komt neer op een extremumvraagstuk.

Een tweede nadeel is dat grotere blokken een limiet stellen aan de hoeveelheid blokken die zich in de caches kunnen bevinden, wat nadelig is voor de capacity misses. Bijvoorbeeld in caches met een grootte van 4k kunnen maar 16 blokken met een grote van 265 byte. Dit kleine aantal blokken betekent dat er vaker blokken moeten verwijderen worden die in de toekomst terug zullen worden opgevraagd; vandaar de stijging van de curve voor 4k en 16k caches in onderstaande grafiek.

Computers met hoge latency en bandbreedte promoten lange blokken aangezien de cache veel meer bytes krijgt per cache miss voor een kleine stijging in miss penalty.



De miss rate in relatie met blok grootte voor verschillende caches.
Bron: John L. Hennessy en David A. Patterson
computer organization and design 4de revisie pagina 465

6.2 Miss penalty verlagen

Uit onderzoek rond de jaren 2000 bleek dat processor snelheid nog altijd sneller steeg dan RAM snelheid, dit betekende dat de miss penalty een steeds grotere impact kreeg op de performantie van de hele computer. Verder zijn miss rate en miss penalty even belangrijk voor de performantie van de cache. Om deze redenen is de focus van optimalisatie van cache verschoven van het verkleinen van de miss rate naar het verkleinen van de miss penalty.

6.2.1 Multilevel caches

De kloof tussen het main memory en de processor stelt cache ontwerpers voor een dilemma: maken ze de cache klein en snel zoals de processor of groot en traag zoals het main memory. In hedendaags computers worden beide gedaan, dit door een hele reeks van cache, elk kleiner, sneller en duurder dan de vorige te gebruiken. Elk niveau dichter bij de CPU heeft een kleinere hit tijd en miss penalty. Het is belangrijk dat de caches niet te klein worden; anders treed er thrashing op.

Deze multilevel caches geven aanleiding tot een nieuwe manier om computers te onderscheiden. Dit onderscheid wordt gemaakt op basis van inclusie (zijnde de aanwezigheid of afwezigheid van kopieën binnen de verschillende caches.) Herinner dat de data in een cache een kopie is van data op het main memory van de computer.

1. *Multilevel inclusion*:
   Alle data aanwezig in een cache op niveau X is ook aanwezig in alle caches onder niveau X.
2. *Multilevel exclusion*:
   Alle data aanwezig in een cache op niveau X is enkel aanwezig in die cache van niveau X en in het main memory. Bij een read en write wordt de data van niveau gewisseld en dus niet gekopieerd. Van één blok data zijn er dus maximaal twee kopieën één in het maim memory en één in een niveau van de cache. Deze methode heeft als voordeel dat er binnen de cache geen data duplicatie is.
   Verder is er minder geheugen nodig om dezelfde hoeveelheid data in cache te houden. Een nadeel van deze methode is dat de miss penalty verhoogt: we moeten immers dieper in de hiërarchie gaan kijken om een woord te zoeken.

De optimalisatie van caches dicht en ver van de processor, moet op een andere manier benaderd worden. Dicht bij de CPU is vooral de hit tijd belangrijk aangezien hier meer hits dan misses gaan voorkomen. In caches dicht bij het main memory gaan misses vaker plaats vinden, dit betekent dat de grootste optimalisatie bereikt kan worden, door te focussen op miss rate en miss penalty.

6.2.2 Prioriteit geven aan read boven write.

Tijdens een cache miss moet een blok worden gekozen om weg te schrijven naar de onderliggende architectuur. Men moet dus wachten tot deze write klaar is voordat men kan beginnen de data te overschrijven met de gezochte data en deze aan te bieden aan de processor. Door gebruik te maken van een write buffer op elk niveau van de cache hoeft men niet te wachten tot de write gedaan is voordat men de read kan hervatten. Een cache met write buffer is sneller omdat men bijvoorbeeld niet hoeft te wachten tot het doel blok gevonden is binnen de cache men schrijft immers de data naar de buffer.
Write buffer:
In de plaats van een write operatie direct uit te voeren, gaan we deze operatie opslaan in een buffer. Hierdoor kan de processor direct doorgaan met andere operatie zonder te moeten wachten op het eindigen van de write-operatie. Vanzelfsprekend hebben write-operaties dan minder clock ticks nodig. Verder wordt de miss penalty verlaagd in caches die gebruik maken van write-back.

Deze write buffer gaat onze performantie verbeteren maar houdt ook een zeker risico in. Wat als we een woord verwijderen uit de cache nog voordat een write op dat woord is uitgevoerd? Om dit probleem te vermijden zijn er twee mogelijkheden.

1) Voor een read miss wordt uitgevoerd moet de buffer leeg zijn.
2) Controleer de inhoud van de write buffer voor een mogelijk conflict.

In hedendaagse computers wordt vanwege performantie redenen nagenoeg enkel methode twee gebruikt.

6.2.3 Write buffers samenvoegen

Als we meerdere keren naar een zelfde blok schrijven zal de write buffer zich vullen met instructies naar hetzelfde blok. We kunnen dit vermijden door te kijken of er in de write buffer al data zit die naar hetzelfde blok moet schrijven en dan deze woorden toevoegen of overschrijven.

Met deze methode wordt de miss penalty kleiner doordat bij misses de write backs sneller gaan. Verder worden het aantal cycli waarbij de processor stil staat omdat de write buffer vol zit verminderd.

6.2.4 Snelle aanrijking

Als de processor een woord opvraagt dat niet in de cache zit, wordt er een volledig blok gekopieerd van onder in de hiërarchie tot in de L1 cache. Pas als het hele blok gekopieerd is, wordt het gezochte woord doorgegeven aan de processor. Bij snelle aanrijking wordt deze volgorde aangepast met als bedoeling het gezocht woord zo snel mogelijk aan de processor aan te bieden.

Er zijn twee varianten:

1. Critical word first:
   Vraag het gezochte woord eerst op, geef dit woord aan de processor en kopieer dan pas het blok naar de L1 cache.
2. Early restart:
   Begin het gezocht blok te kopiëren naar de L1 cache. Van zodra het gezochte woord gekopieerd wordt, stuurt het dan naar de processor.

Dit levert alleen voordeel op als de blokken groot zijn.

6.2.5 Prefetching

Caches gaan data naar de processor brengen nog voor de processor deze opvraagt. Prefetching is een techniek die dat idee gaat toepassen op niveau van de cache. Er wordt data uit de L(X-1) cache opgeslagen in de buurt van de LX cache nog voor de LX cache deze data opvraagt.

Per cache miss van de LX cache worden twee woorden uit de L(X-1) cache naar de LX cache gebracht. Het eerste woord wordt in de LX cache opgeslagen, het tweede woord wordt in een buffer gestockeerd.

De bolgende keer dat de LX cache een woord opvraagt is aan de L(X-1) cache, wordt er gekeken of dat woord toevallig niet in de buffer zit. Als het woord aanwezig is, hoeft het niet meer opgevraagd worden aan de L(X-1) cache.

Door prefectching te gebruiken kunnen tot 70% van alle misses opgevangen worden. Hierdoor wordt zowel de miss rate als de miss penalty verlaagd.

6.3 Verlaag hit tijd.

Een laatste manier om cache te optimaliseren, is door de tijd nodig om een cache hit te verwerken te verlagen. Deze optimalisaties zijn vooral belangrijk in de L1 cache omdat hier de meeste cache hits voorkomen.

6.3.1 Vermijd adres vertalingen

We weten dat de processoren werken met virtuele adressen die tijdelijk verwijzen naar reële adressen op het main memory. Éénzelfde virtueel adres wordt meerdere malen gebruikt door verschillen traits. De cache die zich bevindt tussen de twee spelers, moet dus ook rekening houden met adres vertalingen.

Binnen caches gebruiken we adressen voor twee redenen:

1) Om een blok data te zoeken binnen een set-associative cache, gaan we het blok adres omzetten naar een set index.
   Gaan we indexeren op virtuele of reële adressen?
2) Nadat we op de juiste index hebben gelezen gaan we het gevonden adres vergelijken met het gezochte adres.
   Gebruiken we hiervoor virtuele of reële adressen?

De bedoeling van cache is om snel te zijn, intuïtief lijkt de snelste methode die te zijn waar een minimum aan adressen vertaling plaatsvindt. Dus enkel helemaal vanboven of helemaal beneden in de hiërarchie aan adres vertaling te doen. Aangezien fysieke adressen groter zijn dan virtuele adressen wordt de voorkeur gegeven aan het gebruik van virtuele adressen in de cache. Er wordt dus enkel aan adres vertaling gedaan als er data uit het main memory gehaald moet worden.

Vroeger werden enkel fysieke caches gebruikt; dit zijn caches die enkel reële adressen gebruiken. Tegenwoordig gebruiken nagenoeg alle caches virtuele adressen en worden dan ook virtuele caches genoemd.
Volledig virtuele caches hebben geen nood aan adresvertaling, noch tijdens het opzoeken van de set, noch tijdens het vergelijken van de adressen. Om deze reden horen ze het snelst van alle caches te zijn.

Toch zijn er nog veel niet volledig virtuele caches; dit om twee belangrijke redenen.

1. Veiligheid:
   Page security wordt gecontroleerd als deel van de adresvertaling: als we niet aan adres vertaling doen hebben we dan ook geen pagina beveiliging. Dit kan opgelost worden door page data mee op te slaan in de cache.
2. Proces wisselen:
   Verschillende processen gebruiken dezelfde virtuele adressen om te verwijzen naar verschillende fysieke data. Dit kan op twee manieren opgelost worden:
   2.1. Per proces switch wordt de volledig cache ge-invalideert. Dit zou verschrikkelijk zijn voor de performantie van de cache en wordt niet gebruikt.
   2.2. Samen met elk virtueel adres wordt een proces identifier bijgehouden. Tijdens het opzoeken van een blok moet dan ook deze proces identifier gecontroleerd worden.

6.3.2 Kleine en simpele L1 caches

Het gebruik van lage associativiteit in L1 caches is het gevolg van enkele vaststellingen:

1. Om een blok te vinden binnen zijn set, moeten we de opgeslagen adressen vergelijken met het gezochte adres. In een cache met lage associativiteit (een cache met kleine sets) moeten minder adressen vergeleken worden, wat een kleine hit tijd oplevert.
2. Een hoge associativiteit binnen een cache verlaagt de miss rate.
3. Voor caches dicht bij de processors, is de hit time belangrijker is dan de miss rate.

Buiten een theoretische betere performantie geeft dit ook een lager energie gebruik.

Toch gebruikt men tegenwoordig hogere associativiteit in L1 caches en dit om drie redenen:

1) De meeste processors reserveren twee klok cycli voor een cache operatie, dus hit tijd verkleinen onder twee cycli heeft geen effect.
2) Stel de cache een grootte heeft van maximaal: $\text{associativiteit} * \text{page count}$.
   In dit geval kunnen enkel de bits in de pagina gebruikt worden als index van de set. Een grotere associativiteit betekende een grotere maximum grootte voor de cache.
3) Bij multithreading is er een grotere kans op conflict misses; dit maakt hogere associativiteit aantrekkelijker.

6.3.3 Meerdere banken in cache

In de plaats van de cache als één groot plat stuk geheugen te zien kunnen we de cache verdelen in verschillende losstaande banken die simultaan geconsulteerd kunnen worden. Belangrijk hier is de adressen zo te verdelen over de banken, dat de accesses uit zichzelf verdeeld worden over de verschillende banken.

In de verdeling hiernaast zal een sequentiele doorloping van de adressen er toe leiden dat de verschillende banken geaccesseerd worden, deze verdeling noemt men sequential interleaving.

Bank 1	Bank 2	Bank 3	Bank 4
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

6.3.4 Way prediction

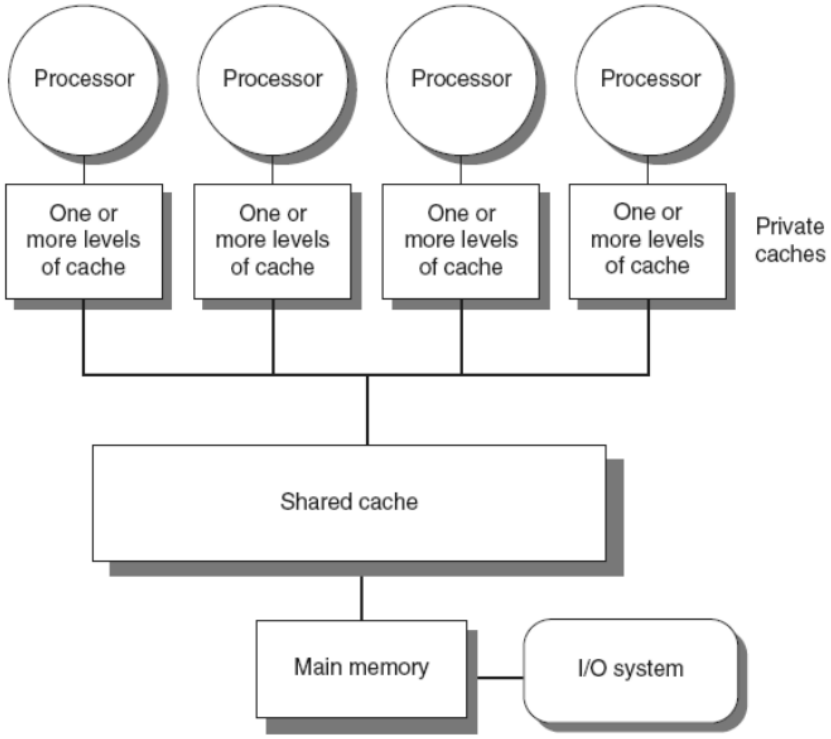
Bij way prediciton worden extra bits opgeslagen in de cache en met deze bits wordt er geprobeerd de volgende blok access te voorspellen. Op basis van deze voorspelling wordt de multiplexor (leeskop) van de cache gezet op het voorspelde blok. Tijdens de volgende cache access wordt enkel het adres dat onder de multiplexor staat, vergeleken en tegelijkertijd wordt de data al op de bus gezet. Als de voorspelling juist was, wordt er minder stroom verbruikt en is de operatie één klok cyclus kleiner. Als de voorspelling fout is, moet er één extra klok cyclus gewacht worden; dit omdat pas één cyclus later de adressen van de andere blokken worden vergeleken.
Simulaties hebben bewezen dat deze voorspellingen in ongeveer 85% van de gevallen correct zijn.

7. Multicore architectuur

Tot op dit moment zijn we altijd van een architectuur met één processor uit gegaan. In dit onderdeel gaan we kijken naar wat er verandert op vlak van caching als we meerdere processoren toevoegen in de architectuur.

Er wordt een onderscheid gemaakt tussen twee soorten multicore architecturen.

1. Symmetric multiprocessors (SMP):
   Deze processoren delen hun centraal geheugen. De naam komt van het feit dat alle processoren dezelfde rechten en memory access time hebben. Deze architectuur werkt alleen voor een kleine hoeveelheid processoren.



Bron: Dmitri Strukov Introduction to Computer Architecture - II 2013
< http://www.ece.ucsb.edu/~strukov/ece154bSpring2013/week10.pdf> slide 4

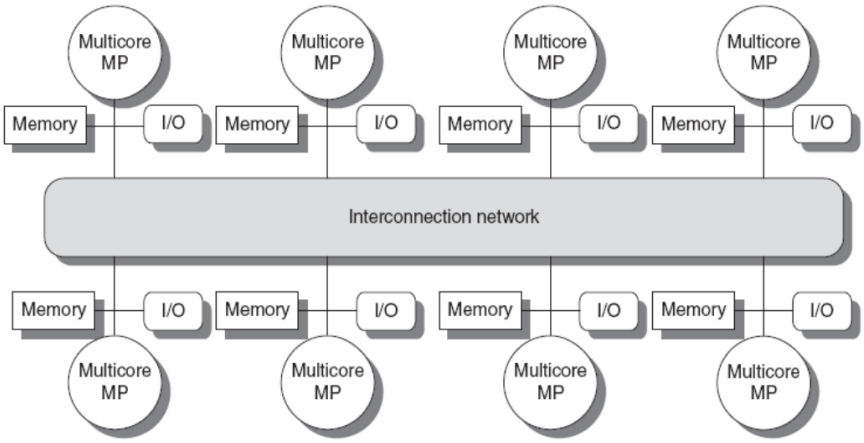
Binnen deze architectuur zijn er twee niveaus van caches.

1. De privé caches bevinden zich net onder de processors. Deze werken gelijkaardig aan de caches die we tot dit punt hebben besproken. Het enige verschil is dat ze ook rekening moeten houden met consistentie problemen.
2. De gedeelde cache bevindt zich tussen de privé caches en het main memory. Omdat de regels van lokaliteit nu gedeeld worden door de verschillende processoren, zal deze cache een abnormaal grote miss rate hebben. De grootste optimalisatie in deze cache is dan ook te halen uit het verlagen van de miss penalty, wat op zijn beurt weer een grote associativiteit promoot.

Deze methode schaalt niet goed: tijdens een cache miss gebruiken alle processoren dezelfde shared cache en het onderliggende main memory, de gedeelde cache is dus een bottle neck. De shared cache en shared medium moeten dus private miss rate * # of processors requests aankunnen. Dit betekent dat er een maximale hoeveelheid processoren te dragen is met deze techniek.

2. Distributed shared memory (DSM)

Deze computers danken hun naam aan het feit dat het centraal geheugen wordt gedistribueerd over de verschillende processoren. Deze methode schaalt veel beter, er is immers geen gedeelde cache die alle cache misses in privé caches moet opvangen. Het nadeel van deze methodes is dat memory latency afhankelijk is van de plaats van het gezochte blok in de hiërarchie. Data opzoeken in het memory juist onder de *multicore multiprocessor* gaat sneller dan data opzoeken in een andere MCMP zijn memory. Voor de consistentie in dit soort architectuur is snooping (zie later) niet mogelijk, er is gewoon geen medium dat zoveel bandbreedte heeft. Hier zal een gedistribueerde versie van het directory-based protocol gebruikt worden.



Bron: Dmitri Strukov Introduction to Computer Architecture - II 2013
< http://www.ece.ucsb.edu/~strukov/ece154bSpring2013/week10.pdf> slide 4

7.1 Coherence strategy

Het cachen van gedeelde data heeft hetzelfde voordeel als het cachen van privé data namelijk een snellere toegang tot de data. In tegenstelling tot privé data is er ook één groot nadeel aan verbonden: de kopieën in verschillende caches moeten consistent gehouden worden. Om deze consistentie te verzekeren zijn er twee mogelijke strategieën:

1. Snooping
   Bij snooping wordt er gebruik gemaakt van het gedeelde medium om de caches consistent te maken. Deze methode is populair voor processoren met maar één core.
2. Directory based:
   Er wordt een directory bijgehouden die de status van alle data onthoudt. In een SMP zal deze directory één centraal punt zijn, in een DSM zal deze directory uiteraard ook gedistribueerd zijn; we willen immers geen centraal punt creëren.

7.1.1 Snooping

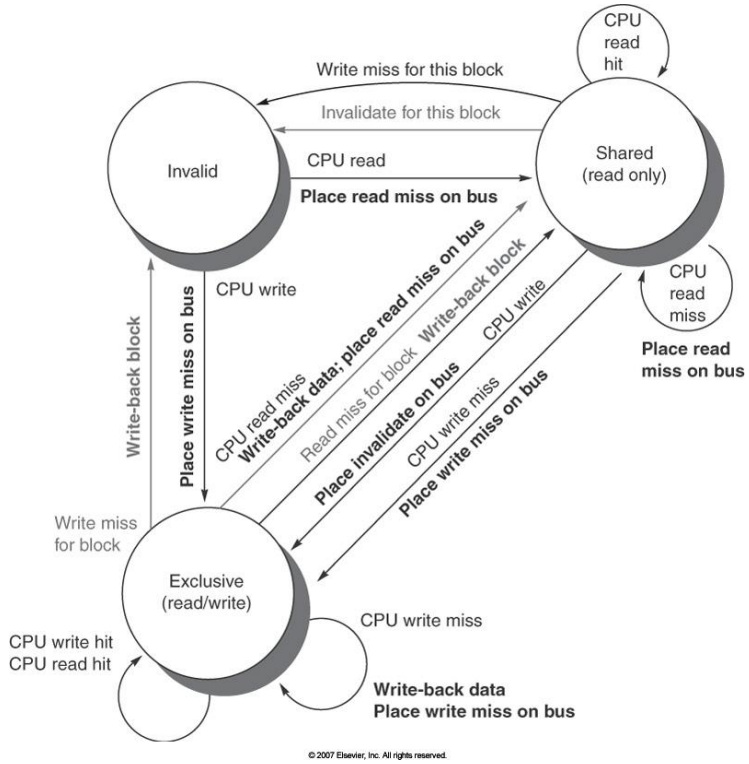
Er zijn twee basis protocollen die snooping implementeren; we kunnen eisen dat alle processoren hun eigen kopie aanpassen als één processor een write-operatie uitvoert of we kunnen eisen dat een processor unieke rechten heeft op de data voordat de processor deze data mag aan passen.

Bij de eerste manier moet een processor het doel adres en de nieuwe data op de bus zetten waarna elke prive cache en de gedeelde cache deze update moet overnemen. Dit betekent dat er veel bandbreedte en stroom wordt gebruikt; om deze reden wordt deze methode nagenoeg niet gebruikt.

Bij de tweede manier wordt enkel het doel adres op de bus gezet. Alle andere caches moeten dan hun kopie invalideren. Doordat er gebruik wordt gemaakt van de bus kan er maar één write tegelijkertijd gebeuren. Als twee processoren tegelijk proberen te schrijven zal één van de twee voorrang krijgen en als er twee maal naar hetzelfde adres geschreven wordt zal enkel de laatste cache zijn kopie overhouden.

Stel processor A heeft een write uitgevoerd en processor B wil nu datzelfde blok inlezen. Dit blok is uiteraard niet te vinden in de cache van processor B. De vraag is nu: waar vind processor B de geüpdate waarde van het gezocht blok?
Voor write-through caches is er geen probleem, de cache van processor A zal immers de waarde aanpassen binnen al zijn privé caches en daarna in de gedeelde cache.
Voor write-back caches is er wel een probleem, de meest recente waarde van het gezocht blok is immers niet noodzakelijk te vinden in de gedeelde cache. Om deze reden zal er een tweede keer gebruik worden gemaakt van de bus. Als een zeker blok niet te vinden is in een privé cache van processor B moet het adres worden opgezocht in de gedeelde cache en tegelijkertijd wordt het adres op de bus gezet. Cache A die een aangepaste versie van de data bezit, zal dan de aangepaste data op de bus zetten. Cache B zal dan werken met de aangepaste data van de bus en niet met de oude data van de gedeelde cache.

Uiteraard is het niet de taak van cache om data op de bus te zetten of te kijken of er data op de bus staat. In processoren wordt deze taak uitgevoerd door finite state controllers (een eindige toestandsmachine), één controller per processor. Deze controller gaat per blok in de cache een statemachine bijhouden. De drie mogelijke staten zijn ongeldig (als een andere processor wil schrijven), gedeeld en privé (voor en na je zelf wil schrijven.) De transities zijn het gevolg van acties van de processor of van data op de bus.



Bron: SIUE digital system design: computer architecture supplementary material
http://www.ee.siue.edu/~ece282/comparchsupp/Ch4-fig07.jpg

Ondanks het feit dat deze machine correct is, is het geen correcte weergave van de realiteit. In deze machine wordt er vanuit gegaan dat operaties atomair zijn. Operaties zoals een write miss die data op de bus zetten en een antwoord van de bus nodig hebben, zijn dat duidelijk niet.

Deze machine verklaart waarom snooping coherentie verzekert:

- Alle valid cache blokken zitten ofwel in de shared state ofwel in de exclusive state.
- Het enige moment waarop een write-operatie mag plaatsvinden, is als een blok zich in de exclusive staat bevindt.
- De enige manier om in de exclusieve staat te geraken, is via een invalidate of via een write-miss. Beiden zorgen ervoor dat alle andere caches hun kopie op invalid zetten.
- Als één van de andere cache het blok in exclusive heeft zitten, zal die cache een write-back uitvoeren voordat de data wordt ge-invalideert. Dit levert de nieuwste waarde op van de data.
- Als een read miss plaatsvindt op de bus terwijl een cache het gezocht blok in exclusive heeft staan, wordt dit blok naar de gedeelde staat verplaatst.

Ook deze methode schaalt niet goed: Bij elke cache miss moet er een bericht op het gedeeld medium worden gezet, dit medium kan niet dubbel gebruikt worden. Bovendien moet men wachten op een antwoord, een antwoord dat kan komen uit elk onderdeel in de hiërarchie, dus ook van het traagste onderdeel. De hele architectuur is dus maar even snel als de traagste component. Deze methode wordt dan ook enkel gebruikt voor architecturen met maximaal acht processoren.

7.1.2 Directory based

Bij een directory based multiprocessor architectuur wordt de status van blokken in de cache bijgehouden in een directory. In tegenstelling tot snooping wordt niet enkel onthouden of een blok gedeeld of aangepast is maar ook of waar dit blok te vinden is. Door deze informatie te gebruiken moet er enkel naar bepaalde controllers berichten worden gestuurd. Dit in tegenstelling tot een broadcast doen op het shared medium (zoals bij snooping gebeurt). Bijvoorbeeld: bij een write-operatie moeten de invalideerberichten enkel gestuurd worden naar die controllers die de data bevatten.

Voor een SMP architectuur zal de directory één centraal punt zijn. Dit maakt de hele zaak vrij simpel, er wordt gebruik gemaakt van een bitvector ter grote van het aantal blokken in de shared cache * het aantal processoren. In deze vector wordt er opgeslagen of de data aanwezig is in de zoveelste processor.

Voorbeeld in een architectuur met drie blokken en drie processoren. De eerste (0^{de}) bit geeft aan of data dirty is, de andere bits of data te vinden is in de n^{de} cache.

Data	1				2				3			
Aanwezigheid	1	1	0	0	0	0	0	0	0	1	1	0

Dit betekent dat:

blok 1 is dirty en te vinden in de cache van processor 1 en de shared cache
datablok 2 is clean en enkel te vinden in de shared cache
datablok 3 is clean en te vinden in de cache van processor 1 en 2 en in de shared cache.

Ook hier moet een processor wachten tot hij als enige een kopie heeft van de data voor hij deze mag aanpassen.

In de DSM gaan we uiteraard geen centraal punt introduceren, deze data moet dus gedistribueerd opgeslagen worden. Belangrijk hier is dat er in heel de architectuur maar één plaats is waar de deel status over een blok te vinden is. Elke processor moet bovendien weten waar hij deze status moet zoeken. Hoe dit allemaal zal gebeuren valt buiten de scope van mijn onderzoek.

8. Simulatie

Voor het ontwikkelen van een cache simulatie moeten twee operaties geïmplementeerd worden: een read-operatie en een write-operatie.

Een algemene read ziet er uit als:
1. Bereken de set waarin het gezochte blok zich kan bevinden
2. Kijk of het gezochte blok zich in de set bevindt
3. Is het blok gevonden?
    Ja: geef gezochte woord weer
    Neen:
    3.1. genereer read in onderliggende structuur
    3.2. genereer cache miss
    3.3. Bereken toekomstige locatie van data, gebruikmakend van de replacement strategie
    3.4. Verwijder data op toekomstige locatie, gebruikmakend van de write-strategy
    3.5. Schrijf data naar de toekomstige locatie en geef het gezochte woord terug.

Een algemene write-operatie ziet er uit als:
1. Bereken de set waarin het gezochte blok zich kan bevinden
2. Kijk of het gezochte blok zich in de set bevindt
3. Is het blok gevonden?
    Ja: pas gezochte blok aan, gebruikmakende van de write-strategy
    Neen:
    3.1. genereer write in onderliggende structuur
    3.2. genereer cache miss
    3.3. Doen we aan write-allocatie?
        Neen: operatie klaar
        Ja:
        3.3.1. Bereken toekomstige locatie van data, gebruikmakend van de replacement strategy
        3.3.2. Verwijder data op toekomstige locatie, gebruikmakend van de write-strategy
        3.3.3. Schrijf data naar de toekomstige locatie.

Buiten punt 2 en 3.1 zijn al deze punten ook variabele punten die gebruikt kunnen worden om de cache te classificeren.
Bijvoorbeeld

Punt 1	Direct mapped (define (address->block a) (modulo a block-count))	Fully associative (define (address->block a) (modulo (random) block-count))

9. Conclusie

Door het simuleren van alle operaties die een cache en omliggende hardware moet uitvoeren, krijg je geen realistische weergaven van performantie, onze simulatie zal immers veel trager zijn dan normaal. De meerwaarde van deze simulatie is te vinden in het academisch aspect: om te kijken wat het effect van verschillende cache implementaties op een programma is hoef je alleen maar andere input methoden te geven aan de simulaties.

10. Bachelor thesis

Voor mijn bachelor thesis ga ik twee dialecten van Scheme ontwikkelen. Samen gaan ze het mogelijk maken programma's te laten werken onder verschillende cache samenstellingen.

De eerste taal zal gebruikt worden om een cache samenstelling te definiëren. Ze zal van de gebruiker eisen dat bepaalde methoden (opgesomd in sectie acht) een implementatie krijgen. Deze methode zullen dan gebruikt worden om het gedrag van een cache te simuleren.

De tweede taal zal van de gebruiker eisen dat een geldige geheugen hiërarchie wordt meegegeven. Hierna zal elk programma geëvalueerd worden, gebruikmakend van deze hiërarchie. Verder kunnen gebruikers statistieken opvragen van de verschillende caches.

Het doel van dit alles is aan de gebruiker de mogelijkheid te geven om met verschillende cache samenstelling te kunnen experimenteren, zijnde die keuzes te maken die normaal door hardware ontwerpers gemaakt worden.
x

11. Referenties

Gekregen van assistent:

[1] P. R. Wilson, M. S. Lam and T. G. Moher, "Caching considerations for Generational Garbage Collection," New York, 1992.

[2] J. L. Hennessy and D. A. Patterson, Computer organisation and design: the hardware/software interface fourth edition, Morgan Kaufmann Publishers, 2011.

Eigen referenties:

[3] J. L. Hennessy and D. A. Patterson, Computer architecture a quantitative approach fifth edition., Morgan Kaufmann Publishers, 2011.

[4] B. Zorn, "The effect of garbage collection on cache performance," 1991.

[5] D. Strukov, "Introduction to Computer Architecture II," 2013.

[6] SIUE, "digital system design".