# Bachelor thesis: cache simulator voor Dr Racket.

Samenvatting

**Wat is een Cache?**

een stuk geheugen geplaats tussen cpu en main memory om pc sneller te maken. Dit door veel gebruikte data dichter bij de cpu te plaatsen. Cache hebben een snellere dataflow dan main memory en zijn duurder per byte. Mensen willen oneindige snelle pc’s met oneindig veel geheugen aan lage prijs. Dus zet verschillende caches onder elkaar, elke cache heeft meer geheugen en is trager dan diegene boven hem. Door slim stukken data te wisselen kan het hele proces werken met de snelheid van de bovenste cache maar met de geheugen capaciteit van de onderste cache.

**Welke data verplaatsen naar bovenste (snelste cache)**?

rule of locallity: spatial locality and temporal locality.

Spatial locality data in de buurt van de data waar we nu aan het lezen zijn heeft een grotere kans om in de toekomst gelezen te worden.

Temporal locality: data die we nu lezen heeft een grotere kans om in de toekomst opnieuw gelezen worden. (loop, counter, itteratie.)

Cache hit: data gezocht in cache en gevonden.

Cache mis: data gezocht in cache en niet gevonden.

**Waar een blok in de cache plaatsen?**

1) fully associative: vrije keuze, om een blok te vinden sequentieel zoeken, btree bijhouden of parallel zoeken.

2) set associative: gelimiteerde keuze, de cache wordt verdeeld in sets, een blok wordt aan de hand van zijn blok nummer opgeslagen in één van deze sets, binnen een set zoeken zoals bij fully associative.

3) direct mapped: geen keuze, elk blok heeft exact één mogelijke plaats binnen geheugen. Om blok terug te vinden gebruik formulle:

(blok address) modullo (maximum aantal blokken in cache)

**Waar zoek ik naar een blok dat mogelijk in de cache zit?**

Pas op verschillende blokken zullen op dezelfde plaats komen te liggen, dit wilt zeggen dat een blok niet altijd in geheugen te vinden is. De blokken in cache moeten dus zelf bijhouden van welk adres te komen. Om een blok te vinden kijk dan waar hij zou kunnen staan en vergelijk gezochte adres met gevonden adres. Normaal worden alle mogelijk plaatsen van een blok parallel vergeleken.

**B8, beneden???**

**Waar een blok in de cache plaatsen?**

Direct mapped, geen keuze blok die op zelfde plaats staat als gezocht blok.

Fully/set associative: 3 mogelijkheden: random, LRU en FIFO.

Random: om allocatie uniform te verdelen, bij defuggen beter om pseudo random te nemen (pseude random kan herhaalt worden.)

Least resently used: Temporal locality zegt dat de minst recent gebruikte blok de kleinste kans heeft om in de toekomst opnieuw gebruikt te worden, doe dus dit blok weg.

First in First out: LRU is soms lastig te gebruiken (hardware mattig), dit benaderd LRU maar kan bereikt worden met simpele queue.

Een andere manier om LRU te benaderen is per set hou een byte-vector bij. Als een blok binnen de set gebruikt wordt zet overeenkomstige bit aan. Als dit de laatste bit is die aan moet gaan binnen de vector negeer dan alle bits. (voor negatie: alle bits buiten één true, na negatie: alle bits buiten één false.) Als een blok verwijderd moet worden neem er een waar de bit nog op false staat, als er meerdere mogelijk zijn kies random.

**Wat gebeurt er tijdens een write?**

Er zijn twee grote stijlen om uit te kiezen: write-through of write-back.

Write-through: data wordt geupdate in cache en onderliggende data structuur.

Write-back: data wordt geüpdate in cache en een bit wordt bijgehouden om aan te geven dat de data is lagere datastructuur moet geüpdate worden als de data verwijderd wordt uit de cache. Lagere bandbreedte nodig, verschilden writes op zelfde blok worden maar één keer naar beneden gepropageerd ook minder energie gebruik.

**Write buffer:**

Een buffer waar write operaties in worden opgeslagen. Dit maakt write en mogelijk ook read sneller (write back. )

Bv read 5

Zonder buffer met buffer

Blok 5 niet gevonden in lvl1Cache, Blok 15 staat in de plaats ==

read 5 in lvl2Cache ==

blok 5 gevonden, zet blok 5 op plaats van blok 15 ==

blok 15 is dirty ==

write 15 lvl2Cache zet 15 in buffer lvl2

write 5 lvl1cache geef blok 5 terug

geef blok 5 terug.

**Wat te doen bij een cache miss na een write?**

Write allocate: zelfde als bij read mis.

No-write allocate: ipv de blok naar boven te halen in de cache dan te updaten volgens strategie, update de data daar waar je hem hebt gevonden en stop.

Normaal: write-back+ write allocate / write-through + no-write allocate.

**Cache performance meten**

Gebruik Average memory acces time as maat van performance voor caches.

Formulles voor performence review b22

**Cache optimizatie**

Optimaliseren op drie vlakken:

1) Verlaag miss rate

2) Verlaag miss penalty

3) Verlaag tijd nodig voor cache hit

1) Verlaag miss rate

Misses kunnen onderverdeeld worden in drie groepen:

Compulsory: de eerste opvraging naar een blok is altijd een mis, de blok kan nog niet in de cache zitten.

Capacity: een opvraging naar een blok nadat deze terug is verwijderd

Conflinct: als te veel blokken naar dezelfde set mappen, kan een blok verwijderd worden en later opnieuw opgevraagd.

Capacity kan enkel opgelost worden door de grootte van de cache uit te breiden. Een geheugen hiërarchie wordt trash genoemd als de cache veel te klein is. Dit zal er voor zorgen dat een groot deel van de tijd de cpu bezig is data van cache niveau te laten switchen. De computer zal dus maar even snel zijn als het laagste niveau in de hiërarchie.

**Langere blokken**

Grotere blokken gebruiken verlaagt de miss rate en verhoogt de miss penalty. Dit is een duidelijke trade-off, blokken groter maken als de inpakt op miss penalty groter is dan op miss rate verlaagd memory access time dus niet overdrijven. Hoge latency en bandbreedte promoten lange blokken aangezien de cache veel meer byte krijgt per mis voor een kleine stijging in miss penalty.

Compulsory misses verlagen doordat grotere blokken meer gebruik kunnen maken van spacial locality.

**Grotere caches**

Verlaagt capacity misses maar verhoogt hit time en is duur.

**Hogere associativiteit**

Hoge associativiteit geeft een lagere miss rate.

Direct mappedly/one-way (lage associtiviteit) ⬄ fully associative (hoge associativiteit)

Problem is dat hogere associativiteit ook een langere hit time oplevert.

**Multilevel caches**

Door meerdere caches onder elkaar te plaatsen krijg je veel geheugen aan hoge snelheid.

Multilevel caches verlagen de hit penalty.

Local miss rate: het aantal misses op dit niveau van de geheugen hiërarchie

Global miss rate: het aantal misses gedeelt door het totaal aantal gegenereerde opzoekingen in de cpu.

Multilevel inclusion: alle data van lvlXcache is present in alle caches onder X.

Multilevel exclusion: alle data van lvlXcache is enkel te vinden in main memory en lvlXcache. Data wordt gewisseld tussen caches en niet gekopieerd.

**Prioriteit geven aan read boven write.**

Probleem read na write met gebruik van write buffer. Er is geen garantie dat alle writes al gebeurt zijn voor we beginnen met volgende read (daarom kan de cache sneller zijn.) Dus tijdens een read miss: wacht tot de buffer leeg is of check of de data toevallig in de write buffer zit. De tweede methode wordt praktisch altijd gebruikt omdat deze het snelste resultaat geeft.

Bij een read miss die een dirty blok moet terug schrijven, schrijft dirty naar buffer, lees juist blok, schrijf dirty naar memory.

**Verlaag hit tijd.**

Normaal zijn caches hardware, software vraag virtueel adres en cache moet dan data ophalen uit ffysiek adres. Om caches snel te maken gaan we virtuele adressen gebruiken in de cache. Dit kan op twee niveau’s:

1) om date terug te vinden moeten we binnen de cache indexeren gaan we indexeren op virtuele of reële adressen

2) na op de juiste index te hebben gelezen adressen vergelijken gebruiken we hiervoor virtuele of reële adressen

Virtuele cache gebruiken virtuele adressen, de meeste hedendaagse caches zijn virtueel.

Reële caches gebruiken reële adressen, dit zijn de traditionele caches.

Volledig virtuele caches hebben geen nood aan adres vertaling na een index te hebben gevonden en voor de tag te vergelijken. Toch zijn niet alle cache volledig virtueel, dit om redenen van veiligheid en omdat per switch van programma de virtuele adressen verwijzen naar een ander fysiek adres.

Zie pagina b37 en volgenden als fucus veranderd.

**Kleine en simpele lvl1Caches**

Een hit gebeurt in drie stappen:

1) adresseer de tag geheugen gebruik makend van het index gedeelte van het adres

2) vergelijk de tag met de waarde van het adres

3) zet de multiplexor zodanig dat de juiste date gelezen word (enkel bij set associatieve caches)

Door de lvl1cache simple (lage associativiteit: direct mapped / 2-way) geeft een lager stroom verbruik. We vergelijken maar 1 of 2 tags respectievelijk.

Toch gebruikt men tegenwoordig hogere associativiteit in lvl1caches dit om drie redenen:

1) de meeste processors reserveren twee clock cycles voor een cache acces dus hit time verkleinen heeft geen prioriteit.

2) Om de TLB (translation look-aside buffer) niet te hoeven gebruiken mag de lvl1cache maximaal associativiteit \* page size. Dit omdat enkel de bits in de pagina gebruikt worden als index.

3) Bij multitreading is er een grotere kans op conflict misses dit maakt hogere associativiteit aantrekkelijker.

**Way prediction**

Voorspel de volgende blok die gaat geaccesseerd worden. Zet de tag van deze blok al op de bus. Als de access plaats vind controlleerd of de voorspelling juist was. Als de voorspelling fout was, zoek blok en pas voorspelling aan.

Verder kunnen we, als de voorspelling juist was, gebruik maken van deze tag om de plaats van de data te vinden en zo stroom besparen, way selection.

Dit maakt pipelining lastig.

**Pipelining**

Pipeline cache acces geeft een snellere klok cyclus tijd en hogere band breedte maar geeft tragere hits.

**Niet blokkerende caches**

Als een computer aan out-of-order execution en pipelining doet kan een cpu tijdens een cache miss in de datacache toch nog instructies blijven ophalen uit de instructie cache.

Een niet blokkerende cache verbreed dit concept tot in de data cache. Dit soort caches kan tijdens een cache miss dus toch nog data blijven opleveren. Hierdoor daalt de hit penalty, dit omdat de cache werkt tijdens een miss ipv de requests van de cpu te negeren. (hit onder miss)

Sommige cache zijn zelfs in staat dit in meerdere dimensies te doen (hit under multiple miss / miss under miss)

Dit maakt performance reviews lastig aangezien de miss penalty niet uniform is. De miss penalty nu is eigenlijk hoeveel cpu stalls er zijn.

**Meerdere banken in cache**

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

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 geacceseerd 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 lijden dat de verschillende banken geaccesseerd worden dus zal het programma het snelst mogelijk zijn. Verdeling noemt sequentiaal interleaving.

**Kritiek woord eerst en vroege herstarting**

Als we een blok opvragen vanwegen een miss kunnen we eerst het gezochte woord copieren direct naar de cpu sturen en dan pas de rest van de cache vullen.

We kunnen ook de blok in normale volgorde kopieren en dan vanzodra we het juiste woord hebben dit doorsturen naar de cpu en de executie herstarten. Vandaar vroege herstarting, de rest van de blok wordt ook gekopieerd parallel aan de rest van de executie.

Deze twee methoden zijn enkel nuttig de blokken groot zijn.

**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 vermeiden door te kijken of er in de write buffer al data zit die naar hetzelfde blok moet schrijven en dan dit woord toe te voegen of te overschrijven.

Zonder write merging met write merging

|  |  |  |  |
| --- | --- | --- | --- |
| Write(100) |  |  |  |
| Write(108) |  |  |  |
| Write(116) |  |  |  |
| Write(124) |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Write(100) | Write(108) | Write(116) | Write(124) |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

De lege vakken hier zouden enkel gebruikt worden als er meerdere woorden tegelijk worden aangepast.

Met deze methode worden writes sneller, en zal de write buffer niet zo snel vol geraken.

**Compiler optimalisatie voor cache vriendelijk gebruik**

Zorg ervoor dat de compiler cache vriendelijke code genereerd.