NTNU
Kunnskap for en bedre verden

Institutt for informatikk og e-læring, NTNU

Cache

Geir Ove Rosvold
6. januar 2016

1. Cache

Resymé: Leksjonen beskriver cache-mekanismen. Tre forskjellige mapping-funksjoner diskuteres. Forskjellige utskiftingsalgoritmer beskrives også. Strategier for skriving via cache beskrives, og en del fundamentale problemer med cache diskuteres.

Innhold

1.1. Hva er cache?

I en tidligere leksjon så vi at moderne CPUer har et problem fordi CPU er så kjapp at primærminnet basert på DRAM ikke greier å levere instruksjoner og data med tilstrekkelig hastighet. For å bøte på dette brukes en mekanisme som vi skal beskrive i denne leksjonen, nemlig Cache.

1.1.1. Innledning

Ting en bruker ofte er det kjekt å ha liggende like ved seg, og slik er det også i datamaskinen. Vi kan oversette cache med hurtigbuffer. Cache er en mekanisme som forsøker å holde den informasjonen som brukes ofte inne i et kjappere minne enn primærminnet.

På kjøkkenet mitt har jeg vekt, hurtigmikser og sleiv liggende innerst i skapet. Når jeg skal bake tar jeg dette fram. Mens jeg baker lar jeg det ligge på kjøkkenbenken. På den måten slipper jeg å bruke tid på å grave det fram fra skapet for hver gang jeg skal bruke det. Når jeg er ferdig med å bake trenger jeg plassen på kjøkkenbenken til andre ting, så da må jeg legge bakeutstyret tilbake i skapet.

Jeg bruker altså kjøkkenbenken min som hurtigbuffer.

Legg merke til at:

1) Når jeg først er inne i skapet og graver tar det ikke særlig lenger tid å hente alt bakeutstyret. Istedenfor å hente bare en og en ting, henter jeg flere ting i samme slengen.
2) Kjøkkenbenken min virker som et hurtigbuffer fordi “aksesstiden” til kjøkkenbenken er kortere enn “aksesstiden” til skuffene.

Cache er altså et minne som har kortere aksesstid enn primærminnet - så kort aksesstid at det holder tritt med CPUen

Dette fører til en liten endring i virkemåten til datamaskinen. Tidligere sa vi at prosessoren hentet inn en instruksjon og utførte den - deretter hentet den neste instruksjon og utførte den.

Med cache hentes det isteden flere instruksjoner fra primærminnet. Alle disse legges i cachen. Neste gang det skal utføres en instruksjon sjekker prosessoren først om instruksjonen allerede ligger i cachen. Hvis den gjør det, ja så får CPUen tilgang til den straks. Dermed slipper prosessoren å vente på det trege primærminnet. Dersom neste instruksjon ikke ligger i cachen, må CPU vente på at den hentes fra primærminnet. I praksis er det faktisk slik at CPU nesten alltid finner neste instruksjon i cachen. Derfor er cache en meget effektivt virkemiddel til å øke ytelsen til en prosessor.

For bakeutstyret i rammen ovenfor var det enkelt å se hvorfor det lønte seg å hente alt bakeutstyret fra skapet med en gang. Når vi første begynner å bake, skal vi jo gjøre oss ferdig - vi begynner for eksempel ikke å lage middag midt oppi all bakingen. Så lenge bakingen foregår, er det bakeutstyret vi har bruk for. Spørsmålet er: hvorfor er det tilsvarende på en datamaskin? Hvorfor er det sannsynlig at neste instruksjon ligger like i nærheten av de foregående? For å forklare det må vi se på noe som kalles «prinsippet om lokalitet».

1.2. Prinsippet om lokalitet

Prinsippet om lokalitet er et svært viktig prinsipp i datateknikk.

Prinsippet sier at dersom en minnelokasjon er benyttet en gang, er det svært sannsynlig at den - eller en lokasjon like ved siden av - snart vil bli benyttet en gang til. I en cache utnyttes dette på den måten at cachen tar vare på en kopi av de sist aksesserte minnelokasjonene og deres nabolokasjoner. Det er nemlig sannsynlig at det snart vil bli bruk for dem igjen.

Prinsippet om lokalitet medfører altså at minnereferansene ikke er spredt tilfeldig rundt i minnet, men har en tendens til å "klumpe seg sammen". Det betyr at det til en hver tid er bare en liten del av minnet som blir brukt intenst. I løpet av denne stunden ligger resten av minnet nesten urørt. Over tid blir hele minnet brukt - men over relativt lange tidsrom (i forhold til CPU-hastigheten) aksesseres en liten del av minnet gjentatte ganger.

Noen grunner til dette er:

- Som regel ligger instruksjonene sekvensielt. Det betyr at neste instruksjon ligger i minnelokasjonen etter den forrige.
- Et program inneholder som regel mange løkker som gjentas flere ganger. Et enkelt eksempel er en while-løkke. Instruksjonene i en slik løkke gjentas mange ganger, noe som betyr at de samme minne-lokasjonene stadig refereres på nytt.
- Eksempler på instruksjoner der prinsippet **ikke** gjelder, er blant annet hopp-instruksjoner og funksjonskall (kall av subrutiner/prosedyrer). Begge deler er eksempler på at programutføringen flyttes til et nytt sted i minnet. Slike instruksjoner utgjør imidlertid bare en liten del av instruksjonene i et typisk program, og utføres derfor sjelden.
- Det er sjelden slik at en funksjon straks kaller en ny funksjon som straks kaller en ny, som straks kaller en ny, osv … Det er altså sjelden at funksjoner nøstes særlig dypt. Da gjelder igjen at de samme instruksjoner har en tendens til å bli brukt på nytt, relativt kort tid etter at de ble brukt forrige gang.
- Mange programkonstruksjoner er iterative. Det vil si at et lite antall instruksjoner gjentas mange gang.
- De aller fleste program benytter datastrukturer som er sammenhengende. Eksempler er arrays og records. Prinsippet om lokalitet gjelder altså ikke bare for instruksjoner, men også for data.

1.2.1. Romlig og temporal lokalitet

Vi skiller gjerne mellom *romlig lokalitet* (engelsk: spatial locality) og *temporal* (eller tidsmessig) *lokalitet* (temporal locality på engelsk).

Romlig lokalitet betyr at minnereferansene ligger nært hverandre. Sekvensiell utføring er et eksempel på slik romlig lokalitet.

Temporal lokalitet betyr at minnereferansene gjerne brukes gjentatte ganger. Løkker er et eksempel på at temporal lokalitet finner sted.

1.2.2. Cache-størrelse

Cache kan altså sees på som en arbeidskopi av det som i øyeblikket er den mest brukte delen av primærminnet. Hvor stor må denne arbeidskopien være? I praksis viser det seg at selv en svært liten cache har god effekt.

Vi kan først ser på noen gamle maskiner. På 80-tallet hadde NORD 10 maskinen fra Norsk Data en cache på 2 KB. Dette utgjorde ¼ promille av primærminnet. På en Intel 486-prosessor var cachen 8 KB, eller rundt en promille av primærminnet.

Moderne PCer har gjerne primærminne på 4-16 GB, og en total cache-størrelse som måles i omtrent like mange MB (altså 4-16 MB). Hva er forskjellen på GB og MB? Jo, siden 1 GB er 1000 ganger så stort som 1 MB, så utgjør cachen fortsatt rundt en promille av størrelsen til primærminnet.

Dette viser hvor sterkt prinsippet om lokalitet er! Over relativt lange tidsrom (i forhold til prosessorens klokkefrekvens) brukes bare omkring en promille av primærminnet. Når vi kopierer denne promillen til cache øker ytelsen til prosessoren betydelig.

Men: Det vi trenger er en automatisk mekanisme som bestemmer hvilke deler av minne som til enhver tid skal ligge i cachen.

1.3. Cachens virkemåte

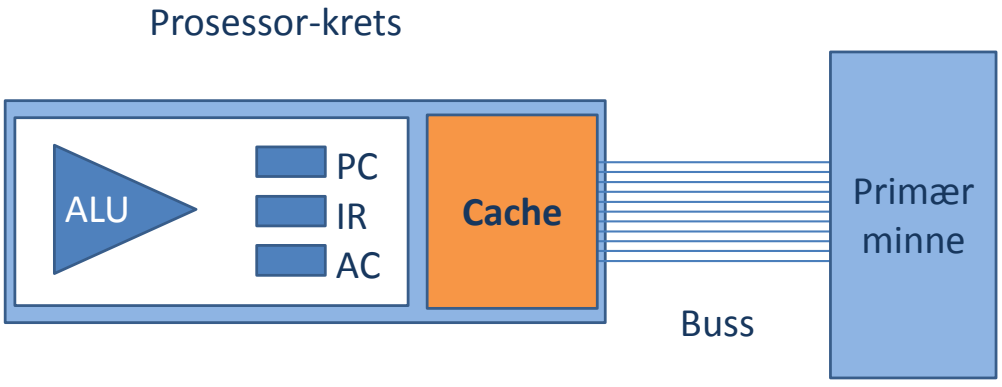
Virkemåten til cache er kort forklart at når prosessoren skal lese en lokasjon fra primærminnet, så sjekker den først om denne lokasjonen allerede er kopiert til cachen. Hvis den **er** det så vil prosessoren lese derfra - og den sparer mye tid siden cache-minnet er mye hurtigere enn primærminnet.

Dersom lokasjonen **ikke** er kopiert til cache vil prosessoren lese fra primærminnet. Da leses imidlertid ikke bare denne ene lokasjonen. Når prosessoren først må ut i primærminnet, så leses det like godt inn en større *blokk* av data bestående av et fast antall lokasjoner - deriblant den som prosessoren vil aksessere. Denne blokken kopieres inn i cachen. Siden prinsippet om lokalitet gjelder, er det stor sannsynlighet for at prosessorens neste lagerreferansen vil ligge i den samme blokken.

Cachen inneholder altså et antall *blokker* med fast størrelse.

1.3.1. Cachens plassering

Hvor er cachen fysisk plassert? Det er viktig at prosessoren har hurtig tilgang til cachen. Derfor ligger cachen i umiddelbar nærhet av prosessoren. Ja, nå til dags er den en integrert del av prosessorkretsen slik Figur 1 viser.

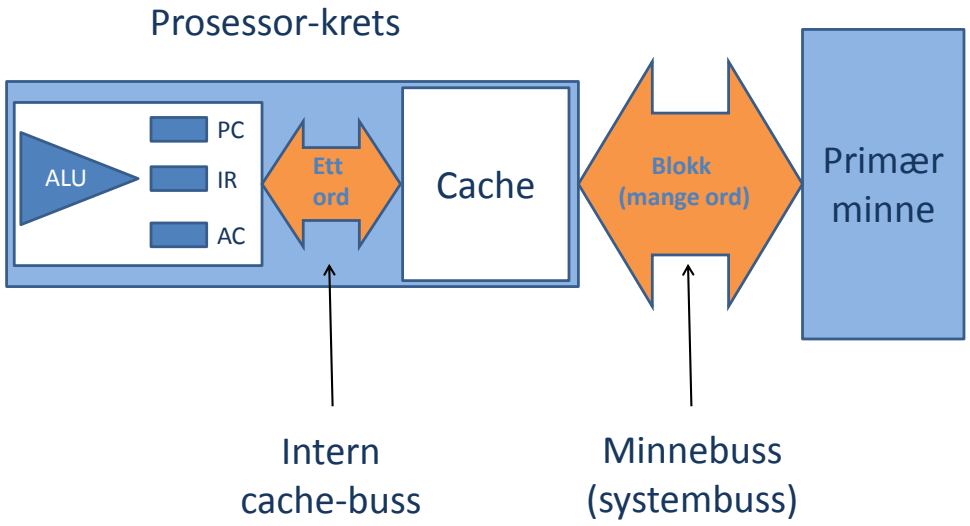


Figur 1 Cache må plasseres nært CPU slik at data mellom cache og CPU kan overføres hurtigst mulig. På moderne prosessorer er cachen innebygget i prosessor-kretsen.

Som nevnt tidligere overføres det blokker mellom primærminnet og cachen. En slik blokk inneholder mange ord. På en PC brukes for eksempel en blokkstørrelse på 64 bytes. Siden

prosessorens registre fortsatt inneholder ett ord, så overføres det selvfølgelig enkelt-ord mellom cache og registre. Dette er fremstilt i Figur 2.

Cachen er bygget opp av statisk RAM (SRAM). Som det ble nevnt i forrige leksjon har SRAM kortere aksesstid enn DRAM. Typiske aksesstider kan være 40-70 ns for primærminne (DRAM) og 1-20 ns for cachens minne (SRAM).



Figur 2 Mellom CPU og cache overføres det ord. Mellom minne og cache overføres det blokker som inneholder mange ord.

1.4. Treffrate

Hensikten med cache er at prosessoren skal slippe å vente på det langsomme primærminnet. Hvis vi er så heldig at alt som CPU skal lese befinner seg i cachen, så vil CPU jobbe uten forsinkelser fordi cachen holder tritt med CPU. Hvis vi derimot aldri finner det vi trenger i cachen, så må prosessoren hele tiden ut i primæminnet for å hente instruksjoner og data.

I praksis vil vi alltid ha en situasjon mellom disse ytterpunktene. En stund finner vi det som vi trenger i cachen, men så etterspør CPU noe som ikke ligger kopiert til cachen. Da må vi ut i primærminne og hente en blokk derfra.

Over litt tid vil vil derfor alltid ha noen aksesser der vi vi finner det vi ønsker i cachen (dette kaller vi treff), og noen aksesser der vi må ut i primærminnet (og det kaller vi bom).

Når prosessoren finner det den ønsker i cachen, så er aksesstiden lik cachens aksesstid. Men når vi må ut i primærminnet, så må rett blokk først hentes fra primærminnet og deretter må vi hente ønsket informasjon i denne blokka.

Den andelen av aksessene som finner data i cachen kalles *treffrate* (eller *hitrate* på engelsk). Treffraten varierer, men ligger ofte mellom 80%-98%. Når et nytt program startes er den lik null - fordi prosessoren ikke har aksessert programmets instruksjoner tidligere - men den øker raskt.

1.5. Effektiv aksesstid ved bruk av cache

Enkelte ganger er vi heldige og finner det vi leter etter i cachen, mens andre ganger er vi uheldige og bommer. Nå skal vi finne en gjennomsnittlig aksesstid ved en gitt treffrate. Da må vi vite aksesstiden ved treff, aksesstiden ved bom, og vi må kjenne treffraten.

Aksesstid ved treff: Når vi får et treff vil aksesstiden være lik aksesstiden til cachen. Vi kaller cachens aksesstid for T_c .

Aksesstid ved bom: Når vi får en bom vil vi først måtte lese data fra primærminnet og inn i cachen. Deretter leser vi data fra cachen. Vi kaller aksesstiden til primærminnet for T_p. Da blir aksesstiden ved bom lik $(T_p + T_c)$.

Treffraten: Vi kaller treffraten for H. Da vil H være er et tall mellom 0 og 1. Hvis treffraten er 0 har vi bare bom. Med en treffrate på 1 har vi bare treff. I alle andre tilfeller er H et sted mellom 0 og 1. La oss se på et eksempel: Hvis vi treffer i 70% av tilfellene vil treffraten, H, være 0,7. Legg merke til at da vil vi bomme i 30% av tilfellene. Vi bommer altså med andelen (1-H), som er 0,3 i dette tilfellet.

Den gjennomsnittlig aksesstid over mange aksesser kaller vi *effektiv aksesstid*, eller T_e. Nå kan vi sette opp et matematisk utrykk for denne. Vi antar en treffrate på H. Det betyr at *H-delen* av aksessene vil være treff. De øvrige aksesser, altså *(1-H) delen*, gir bom. Den delen av aksessene som gir treff vil ha en aksesstid på T_c. Den delen som gir bom, vil ha aksesstiden $(T_p + T_c)$. Den effektive (eller gjennomsnittlige) aksesstiden blir:

T_e = (Andel treff) x (aksesstid ved treff) + (Andel bom) x (aksesstid ved bom)

= H x Tc + (1-H) x (Tp + Tc)

= H Tc + (1-H) (Tp + Tc)

= H Tc + Tp + Tc – H Tp – H Tc

= Tc + Tp – H Tp

= Tc + (1-H)Tp (1)

Her er

T_e - er effektiv aksesstid.
H - er treffraten.
T_c - er cachens aksesstid.
T_p - er aksesstid til primærminnet.

Fra ligning (1) ovenfor ser vi følgende. Hvis vi har en høy treffrate vil H nesten være lik 1. Da vil (1-H) bli nesten null, og følgelig blir $(1-H)T_p$ også nesten null. Det betyr at T_e nesten bare er avhengig av cachens aksesstid, T_c. Ved høy treffrate er altså den effektive aksesstiden nesten lik cachens aksesstid.

Hvis treffraten er lav, altså at H nesten er null, vil (1-H) nærme seg 1. Det betyr at leddet $(1-H)T_p$ begynner å gjøre seg svært gjeldende. Fordi T_p er mye større enn T_c vil dette bety at T_e nesten bare avhenger av T_p. Ved lav treffrate vil den effektive aksesstiden altså bli tilnærmet lik aksesstiden til primærminnet.

Eksempel på beregning av effektiv aksesstid ved bruk av cache:

Anta at vi har et primærminne med aksesstid 60 ns. Mellom CPU og primærminnet har vi en

cache med aksesstid på 10 ns. Treffraten er 95%, det vil si at H=0,95. Finn effektiv aksesstid.

Vi bruker formelen vår:

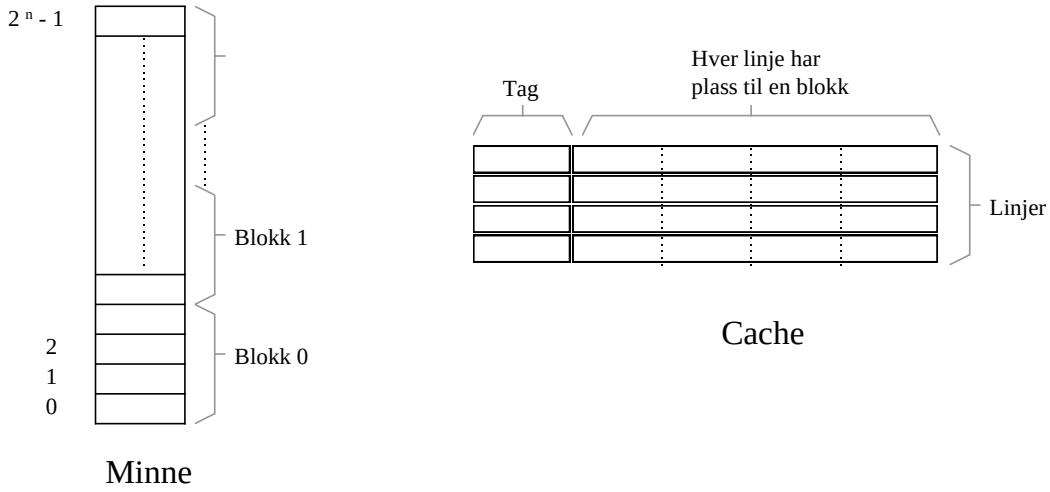
$T_e = T_c + (1-H)T = 10ns + (1-0,95)*60ns = 10ns + 0,05*60ns = 10ns + 3ns = 13ns$

Den aksesstiden som CPU "opplever" er altså 13 ns. Det vil si en effektiv aksesstid som er omtrent fem ganger så hurtig som primærminnet.

1.6. Mer om cachens virkemåte

Vi skal nå se i mer detalj på hvordan cachen virker. Det er tre ting som er viktig å huske når man leser resten av denne leksjonen:

1. Tidligere har vi sett på minnet som en samling av adresserbare ord, som regel tenker vi på minnet som bygget opp av bytes. Vi har sagt at når vi leser fra minnet, så henter vi ett og ett ord. Når vi bruker cache henter vi imidlertid aldri enkelt-ord eller enkelt-bytes. Isteden henter vi en hel blokk. Derfor går vi over til å se på minnet som en samling av slike blokker. Vi nummererer blokkene fra 0 og oppover. Når CPU etterspør ett bestemt ord, henter cache inn hele den blokken som inneholder dette ordet.
2. Cachen er mye mindre enn primærminnet, men har likevel plass til mer enn bare en blokk. Vi sier at cachen består av såkalte *linjer* hvor hver *linje* har plass til en blokk.
3. Siden cachen er så svært mye mindre enn primærminnet så vil mange blokker fra minnet måtte bytte på å bruke samme *linje* i cache. CPU må notere seg *hvilken* blokk som i øyeblikket er kopiert til hver enkelt *linje* i cachen. Hver cache-linje må derfor utstyres med en identifikator som forteller hvilken blokk av minnet som ligger der. Denne identifikasjonen lagres sammen med blokken og kalles en *tag* («merkelapp»). Se Figur 3



Figur 3 Minnet er organisert som adresserbare ord, men det kan også sees på som en rekke med blokker der hver blokk består av flere ord. Cache består av et antall "linjer", hvor hver linje kan lagre nøyaktig èn blokk. Når vi leser fra minnet leser vi en hel blokk - isteden for et ord - og legger blokken i en linje. "Tagen" til linjen forteller hvilken blokk som ligger der. En moderne PC har en blokkstørrelse på 64 bytes.

1.7. Blokkstørrelse

En cache kan enten deles opp i et lite antall store blokker, eller i et stort antall små blokker Blokkstørrelsen er alltid en avveining mellom flere forhold, og den optimale blokkstørrelsen vil variere med hva slags oppgaver prosessoren brukes til å løse.

Programmer med stor grad av romlig lokalitet vil jobbe lenge innenfor en svært avgrenset del av minnet. Da bør blokkstørrelsen være stor for å holde blokka i cachen så lenge som mulig.

Når man først får en bom så straffes man imidlertid hardere ved store blokker enn ved små blokker. Dette er fordi en ny stor blokk vil ta lengre tid å overføre fra primærminnet via bussen og over til cachen. (Med en gitt bussbredde vil en stor blokk kreve flere buss-sykluser enn en liten blokk.)

Et moment som taler for å ha små blokker er at dersom man har lav romlig lokalitet (man hopper litt frem og tilbake i minnet), men stor temporal lokalitet (man bruker likevel disse spredte lokasjonene om og om igjen), så vil en cache med mange små blokker ha en større sannsynlighet for å inneholde de rette blokkene.

Kompromisset som moderne cache-design har landet på er gjerne en blokkstørrelse på mellom 32 og 64 bytes.

Eksempel på blokkstørrelse

På en eldre PC brukes en blokkstørrelse på 32 bytes. Da sees altså minnet på som en samling blokker hvor hver blokk er 32 bytes. Blokk **nr 0** består av adressene 0..31, **blokk nr 1** består av adressene 32..63, **blokk nr 2** av adressene 64..95 og så videre.

Dersom CPU etterspør adresse nr 35 i primærminnet, henter altså cachen inn blokk nr 1 siden adresse 35 befinner seg i denne blokken. Cachen vil plukke ut en av linjene i cachen, og legge blokk 1 i denne linjen.

Nyere PCer bruker en blokkstørrelse på 64 bytes.

1.8. Mapping-funksjon

Når en blokk kopieres til cachen, må det bestemmes i hvilken linje i cachen blokka skal ligge. Dette bestemmes med en mappingfunksjon.

Valg av mapping-funksjon er en avveining mellom enkelhet på den ene siden og et ønske om å ikke unødig kaste blokker ut av cachen på den andre siden. Vi skal se på tre ulike mapping-funksjoner.

1.8.1. Full-assosiativ cache

Her kan en blokk legges i en hvilken som helst linje. Når CPU skal hente data, må den sjekke hver eneste tag i cachen. Dette krever komplisert elektronikk.

Det er viktig å huske at hvis det skal være noen gevinst i å bruke cache så har maskinen bare noen få nanosekund til å finne ut om data ligger i cachen. Selv med den hurtigste elektronikk som finnes i dag er det alt for tidkrevende å sjekke linjene etter tur. Samtlige tag-verdier må derfor sammenlignes i parallell mot tagen til de data CPU vil hente. Dette krever like mange sammenligningskretser som det er linjer i cachen - noe som betyr at full-assosiative cacher er

komplisert å implementere. Dette er grunnene til at full-assosiative cacher ikke brukes i praksis.

1.8.2. Ikke-assosiativ cache (direkte mapping)

For å slippe å ha sammenligningslogikk for hver eneste linje kan vi isteden la hver blokk i minnet ha en *fast* plass i cache. Dette kalles *ikke-assosiativ cache*, eller *direkte-mappet cache*. Da kan en blokk bare legges på en bestemt linje i cachen, men det er mange blokker som "tilhører" hver linje.

Når CPU skal sjekke om en bestemt blokk ligger i cachen, trenger den bare å sjekke en eneste linje. CPU vet på forhånd hvilken linje den skal sjekke – den vet ikke om den aktuelle blokken faktisk ligger i linjen, men den vet hvilken linje den skal sjekke.

Eksempel på ikke-assosiativ cache (direkte mapping)

Dersom vi har 4 linjer i cachen, vil
blokk nr 0, 4, 8, 12, …. konkurrere om å ligge i linje nr 0
blokk nr 1, 5, 9, 13, …. konkurrere om å ligge i linje nr 1
blokk nr 2, 6, 10, 14, …. konkurrere om å ligge i linje nr 2
blokk nr 3, 7, 11, 15, …. konkurrere om å ligge i linje nr 3

Dersom CPU etterspør ett ord i blokk nr 14, trenger ikke cachen å sjekke alle linjene. Den vet at dersom blokk nr 14 befinner seg i cachen, så ligger den i linje 2. Derfor sjekker den taggen kun til denne ene linjen.

I praksis har en cache mye mer enn fire linjer, men eksemplet illustrerer virkemåten.

Trashing

Ulempen er at to ulike ord som brukes samtidig kan komme til å havne opp i samme linje. For eksempel kan det hende at rett etter at en instruksjon er hentet inn må den vike plassen igjen for en data-verdi som havner i samme linje. Hvis dette skjer ofte kan verdien av caching reduseres betydelig.

Denne situasjonen hvor to blokker gjensidig tvinger hverandre ut av cache hele tiden kalles *trashing*.

1.8.3. Sett-assosiativ cache

For å unngå at blokker gjensidig tvinger hverandre ut av cache hele tiden, kan vi isteden organisere linjene i *sett*: Blokkene som sloss om samme linje får flere linjer å dele på, så det blir plass til flere av dem. Vi slår sammen for eksempel 2 eller 4 linjer til *et sett*, der flere «konkurrenter» kan ligge side om side.

En cache med 2 linjer pr sett kalles «2-veis», og kan ha to konkurrerende blokker inne samtidig; har vi 4 linjer pr sett kan fire konkurrenter være inne samtidig, og vi kaller det 4-veis cache.

I praksis er det nesten alltid sett-assosiative cacher som brukes. Sett-assosiative cacher gir en rimelig avveining mellom enkelhet og fare for trashing, og vil som regel gi den høyeste treffraten. I de aller fleste tilfeller brukes 2-veis, 4-veis, 8-veis eller 16-veis cacher.

1.9. Utskiftingsalgoritmer

For fullt assosiativ og sett-assosiativ cache må vi velge mellom flere mulige linjer der data fra minnet kan kopieres til cache. (For direkte-mappet cache har man intet valg.) Dette valget må gjøres i hardware, på noen få nanosekunder, om det ikke skal forsinke CPU. Metodene må derfor være enkle.

- **LRU** : Least Recently Used. Bytter ut den blokka det er lengst siden ble aksessert. Hver cache-linje har en teller som økes for hver minne-aksess der denne blokka ikke ble etterspurt, og nullstilles hvis blokka ble referert. (For sett-assosiativ cache endres telleren bare for referanser til samme sett.) Linjen med høyest telle-verdi er den som velges som offer først.
- **FIFO** : First In First Out. Bytter ut den blokka som har vært lengst i cachen. Som LRU, men telleren økes bare når en ny blokk hentes fra minnet; den nye blokkas teller settes til 0. (For sett-assosiativ cache bare når det hentes inn til samme sett.)
- **LFU** : Least Frequently Used. Bytter ut den blokka som har vært aksessert færrest antall ganger. Telleren økes for referanser til dette ordet, ellers ikke. Den linjen som har lavest telleverdi velges som offer først. Med jamne mellomrom telles alle tellerne ned med samme verdi for å «glemme» referanser som skjedde for en stund siden.
- **Random** : Tilfeldig utskifting. En tilfeldig av blokkene velges som offer.

Fordi valget må gjøres på noen få nanosekunder, brukes ofte vesentlig enklere utgaver av disse strategiene enn hva vi kan akseptere i andre sammenhenger. En form for tilfeldig utskifting er svært vanlig.

1.10. Skriving til minnet

Så langt har vi konsentrert oss om lesing fra minnet. Men av og til skal jo CPU skrive til minnet også. Hvordan skal dette gjøres når vi har cache?

Problem: CPU vil endre verdien i en lokasjon i minnet. Hva skal skje hvis en kopi av lokasjonene ligger i cachen?

Her brukes to forskjellige løsninger:

Write through:
Skriving skjer samtidig både på cachen og i primærminnet. Fordelen med denne løsningen er at primærminnet alltid er oppdatert. Ulempen er at bruken av cache ikke reduserer belastningen på bussen ved skriving til minnet; dette kan sinke både CPU (fordi den må vente på at skrivingen fullføres før neste skriving godtas) og andre enheter som arbeider i parallell på samme buss.

Write back:
Da skrives det kun i cachen, primærminnet oppdateres først når innholdet av en cache-linje må vike plassen. Fordelen er at vi får redusert buss-trafikk, og dermed maksimal hastighet. Ulempe er at utskifting av innholdet i en linje tar lengre tid, og CPUen må stå å vente i mellomtiden. Dessuten får vi problem med I/O-utstyr (se neste punkt)

1.11. Cache-koherens

I et datasystem har man ofte flere enheter som kan endre innholdet i minnet. For eksempel kan IO-utstyr legge inn nytt innhold i deler av minnet med DMA (Direkte Minne-Aksess). Et

annet eksempel er fler-prosessor-systemer der flere prosessorer deler på ett og samme primærminne.

Siden en cache inneholder en kopi av deler av primærminnet, kan vi fort risikere at lokasjoner som er kopiert til en cache blir endret. Hva skal vi da gjøre. Da samsvarer ikke innholdet i cachen med primærminnet. Vi sier at cache og primærminnet ikke er konsistent.

Slike praktiske problem er årsaken til at cache-mekanismen på enkelte CPUer er omtrent like komplisert som resten av CPUen. Det at det er samsvar mellom primærminnet og cache kalles cache-coherence.

Problem med at I/O-kontrollere kan endre minnet (DMA)

Problem 1: Hva skjer hvis I/O-utstyr endrer primærminnet, og en kopi av blokken ligger i cache?

Alternativ 1: Hvis ikke programmet skal lese foreldede verdier må cache-innholdet markeres som ugyldig. Som regel er det ikke mulig å gjøre dette for hver enkelt linje. Isteden gjøres det for hele cache-lageret under ett.

Alternativ 2: Hvis datamaskinen er et flerprosess-system kan operativsystemet overlate CPU (og cache) til et annet program enn det som ba om I/O-operasjon. Da kan I/O-utstyret endre primærminnet før programmet som startet I/O-operasjonen slipper til igjen. På denne måten unngås det at problemet oppstår.

Alternativ 3: Alle I/O-operasjoner kan i prinsippet gå via cachen. Dette gjør system-design og hardware-modularisering så problematisk at det sjelden benyttes.

Problem 2: Anta at maskinen har write-back cache. Hva skjer når et program endrer data i cache, og så ber om at I/O-utstyr (som leser direkte fra primærminne) skriver disse dataene ut til disk. Da er de nye data enda ikke skrevet tilbake til primærminnet, de nye verdiene ligger bare i cache?

Alternativ 1: De nye verdiene må skrives ut til primærminne før operativsystemet tillater at I/O-operasjonen settes igang. Alle write-back-cacher har en kontroll-inngang som «tømmer» endrede verdier til primærminne («cache flush»).

Alternativ 2: I/O-operasjoner kan gå gjennom cachen - se ovenfor.

Problem 3: Flere prosessorer (i et multi-CPU-system) har hver sin kopi av de samme data, i hver sin cache. Hva skjer når en av prosessorene endrer innholdet?

På fler-prosessorsystemer finnes det et eget system for å sikre samsvar mellom cachene, eller *cache-coherence* som det kalles. Det går egne kontroll-linjer der cachene rapporterer til hverandre hvilke data som blir endret hos dem. Andre cache-kontrollere med kopier av samme data vil da merke sine kopier som ugyldig, og neste gang de refereres hentes det ny, oppdatert kopi fra primærminne.

Senere skal vi lære om fler-kjerne-prosessorer. Dette er prosessorkretser som inneholder flere prosessorer som jobber i parallell. En av fordelene med å samle kjernene i en krets i stedet for å ha flere separate prosessor-kretser er at mekanismen for å sikre cache-koherense er samlet i en prosessor-krets.

1.12. Hastighetsforbedring ved bruk av cache

Vi har nå sett at bruk av cache kompliserer CPUens arbeid i stor grad. Det kan være naturlig å spørre om det er bryet verdt.

Gevinsten med cache varierer både med datamaskinens øvrige oppbygging, hva slags type program som kjøres og hva slags oppgave datamaskinen brukes til. Hvis det brukes langsom RAM til primærminnet vil cache gi større gevinst enn når det brukes hurtig RAM. Mye venting på I/O - som skjermfunksjoner o.l - gir mindre forbedring. Helt tilfeldig data-aksess gir mindre forbedring. Og til sist: svært tette løkker og/eller små datasett gir større forbedring.

Forsøk og forskning viser at optimal cachestørrelse varierer fra programmeringsoppgave til programmeringsoppgave, men selv svært små cacher (noen få ti-talls KB) gir god effekt.

2. Moderne cache-design

Alle moderne CPUer har cache. For eksempel ble cache introdusert på Intel sine prosessorer i 1984. Det var på 486-prosessoren. Etter hvert som tiden har gått har cache-mekanismen blitt mer og mer avansert. Den har også blitt viktigere og viktigere for prosessorens ytelse.

Typisk for de nye prosessorer er at de har såkalt *fler-nivå cache*. Andre begreper vi skal se på er *splittet cache* kontra *enhetlig cache*. I tillegg skal vi innom *delt cache* (eller *felles cache*). Alle disse begrepene er sentrale i moderne cache, og vi skal bruke litt tid på dem.

2.1.1. Fler-nivå caching

Forskjellen på prosessorens ytelse og minnets aksesstid blir bare større og større, og behovet for cache bare øker. Derfor kan man tenkte seg å øke størrelsen på cachen.

Cache-størrelse er en avveining mellom flere forhold. Vi ønsker at cache-størrelsen skal være så stor at vi oppnår en høy treffrate. Da vil jo den effektive aksesstiden i all hovedsak bli bestemt av cache-minnets aksesstid. Dessverre er det samtidig slik at dess større en cache er, dess mer elektronikk trengs for å kontrollere den. Elektronikkens hastighet er avhengig av størrelsen slik at en stor cache vil ha en tendens til å være langsommere enn en liten, selv når de er bygget opp med samme teknologi.

En godt alternativ kan da være å bruke to cacher: en liten og svært hurtig cache for de aller mest brukte lokasjonene, og en større cache for lokasjoner som er litt mindre brukt. Den store cachen er langsommere enn den lille, men fortsatt mye hurtigere enn primærminnet. På denne måten blir ytelsen bedre enn om man brukte en eneste stor cache.

Dette kalles to-nivå caching, og har i mange år vært vanlig på generelle prosessorer. Den lille cachen kalles *nivå 1 cache* (på engelsk: *Level 1 cache* eller bare *L1–cache*), mens den store cachen kalles *nivå 2 cache* (*Level 2 cache* eller *L2–cache*).

Etter som prosessor-ytelsen øker mye hurtigere enn ytelsen til DRAM blir cache bare viktigere, og de siste årene har mange prosessorer kommet med tre cache-nivåer. *Nivå 3 cachen* (L3-cachen) er enda større enn L2-cachen. Den er også litt tregere. Men den er mye hurtigere enn primærminnet, og avlaster primærminnet i stor grad.

Med flernivå caching blir ligningen for å beregne den effektive aksesstiden mer komplisert fordi vi opererer med flere treffrater. Jeg har valgt å ikke diskutere formler for effektiv aksess-tid for flernivå caching i dette kurset.

2.1.2. Splittet cache kontra enhetlig cache

En tradisjonell cache brukes både til instruksjoner og data. Dette kalles en *enhetlig cache* (*unified cache* på engelsk), og har den fordelen at den automatisk avpasser antall instruksjoner kontra antall data i cachen.

Det vil si at dersom et program oftere henter instruksjoner enn data fra minnet, så tilpasser cachen seg dette ved at den automatisk inneholder flere instruksjoner. Det motsatte gjelder selvfølgelig også, hvis programmet henter mer data enn instruksjoner. Vi har altså en dynamisk fordeling av data og instruksjoner. I tillegg har en enhetlig cache den fordelen at det er enklere å designe og implementere èn cache enn to.

Motstykket til en slik enhetlig cache er det vi kaller en *splittet cache*. Splittet cache vil si at cachen består av to selvstendige del-cacher; en for instruksjoner og en for data. Dette kan være en stor fordel i enkelte tilfeller, nemlig dersom vi skulle få bruk for både en instruksjon og data samtidig. Men når skulle et slikt behov kunne oppstå? Jo, det skal vi se mer i detalj på senere i kurset, men saken er at moderne prosessorer jobber med flere instruksjoner samtidig. Dette gjelder spesielt såkalte *superskalare prosessorer*; det vil si prosessorer som inneholder mer enn en prosesserende enhet, og som utfører flere instruksjoner samtidig. På slike prosessorer hender det ofte at èn prosesserende enhet vil lese en ny instruksjon akkurat samtidig med at en annen prosesserende enhet vil lese eller skrive data. Dette er fullt mulig med en delt cache siden de to del-cachene har hver sine forbindelseslinjer til prosessoren, og dermed kan aksesseres samtidig.

2.1.3. Egen kontra felles (delt) cache

Moderne prosessorer har flere kjerner. Hva dette betyr skal vi se på senere i kurset. Men i kortversjon betyr det at flere prosessorer er bygget sammen i en og samme krets. Med en fire-kjernes prosessor vil operativsystemet oppfatte det som en maskin med fire prosessorer. Selv om det bare er en prosessorkrets.

På en slik fler-kjerne prosessor kan enkelte cacher være felles for alle kjerner, mens andre cacher bare brukes av en kjerne.

På nyere prosessorer ser vi ofte at hver kjerne har sin egen L1- og L2-cache. Og så er det en egen L3-cache som brukes av alle kjernene; en såkalt *felles cache* eller *delt cache*.

2.1.4. Sett-størrelse

Den som designer et cache-system må velge hvor mange linjer det skal være i hvert sett. Dette kalles sett-størrelse, og som det fremgår av kapittel 1.8.2 er den ofte 2, 4, 8 eller 16. Nå vet vi at det finnes flere nivå med cacher på en prosessor, og vi kan diskutere sett-størrelsen litt mer inngående.

En høy verdi på sett-størrelsen vil selvsagt gjøre cachen lite sårbar for trashing. Sannsynligheten for å trashing vil jo være lavere dess høyere sett-størrelsen er.

Problemet med stor sett-størrelse er imidlertid at cachen blir mer komplisert å lage, og det er vanskelig å få cachen særlig hurtig fordi den krever mer kontroll-elektronikk. (De som leser Vedlegg A til denne leksjonen vil f.eks se at det kreves flere bits i tag-en til cachen når sett-størrelsen øker.) Alt dette gjør at straffen for å bomme blir større ved store sett-størrelser.

Avveiningen blir derfor mellom straffen for å bomme kontra gevinsten ved å treffe.

Siden straffen for å bomme er lavere for cachene som befinner seg nær prosessoren vil L1-cachen ha en lav sett-størrelse, f.eks. to eller fire, mens sett-størrelsen øker for cacher lenger bort. L3-cachen er ofte en 16-veis sett-assosiativ cache

2.1.5. Utviklingstrender innen cache

Siden gapet mellom ytelsen til CPU og ytelsen til primærminnet bare fortsetter å øke, vil cache bli viktigere og viktigere. De trendene vi har sett de senere år er at cache-mengden øker og at det tas i bruk flere nivå av cache.

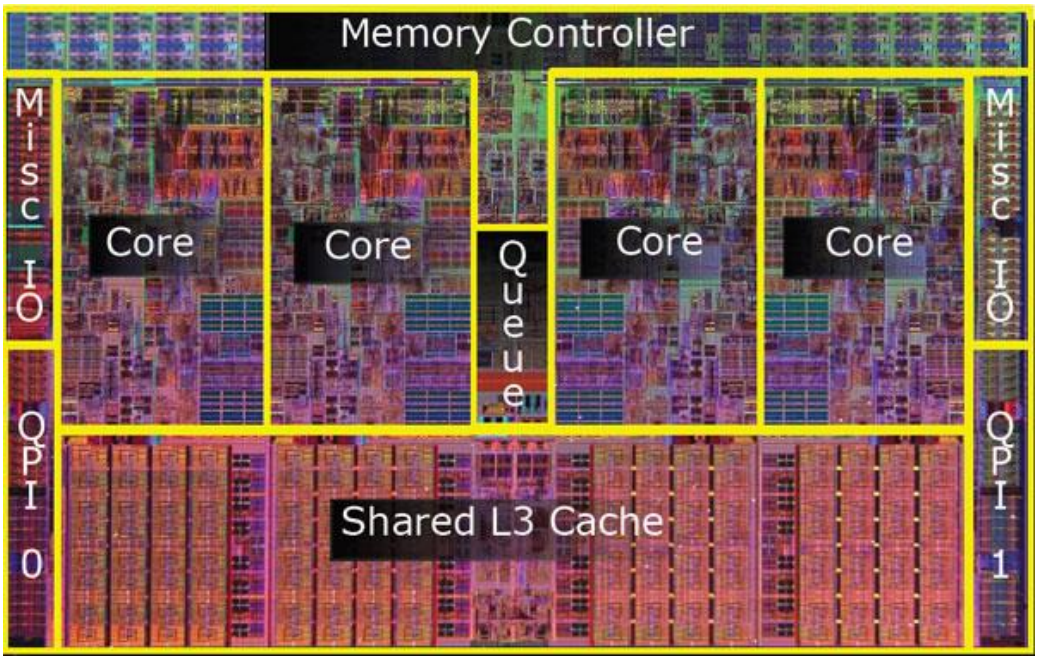
Større og større totalmengde cache. Fra en total mengde på noen KB cache på sent 80-tall, via noen hundretalls KB på 90-tallet, ser vi nå at totalmengde cache måles i 10-talls MB på moderne prosessorer.

Antall cache-nivå øker. På de fleste platformer er L1-cachen bare litt større enn den var på 90-tallet, 32 KB til instruksjons-cache og 32 KB til data-cache er typisk. L2-cachen er oftest fra en halv MB til noen få MB. Mens man ved større cache-behov heller baserer seg på et 3. cache-nivå: en L3-cache som kan være temmelig stor: fra 2 MB og oppover til flere 10-talls MB.

Totalmengde cache er svært avhengig av ytelsen til CPU. Lavpris-prosessorer har mye mindre cache enn dyre prosessorer med høy ytelse.

Eksempel på moderne cache, Intel i7 Sandy Bridge

Sandy Bridge er en nyere prosessorarkitektur fra Intel. Denne prosessoren har fire kjerner mer hver sin L1- og L2-cacher. I tillegg har den en stor felles L3-cache. Innmaten er vist i Figur 4.



Figur 4 Innmaten i Intel i7 (sandy bridge). Bildet viser en Sandy Bridge prosessor. Det er fire kjerner (og hver kjerne har sin egen L1 og L2 cache), en delt L3 cache (8 MB) og innebygget minne-kontroller. Legg merke til hvor stor del av prosessoren som brukes til cache. Bildet er hentet fra http://www.sharkyextreme.com/hardware/cpu/article.php/3782516/Intel-Core-i7965-XE--Core-i7920-Review.htm

Et eksempel på cache-konfigurasjon på en i7-prosessor kan være:

Blokk-størrelse: 64 Byte

L1-cache. Hver kjerne har:
Instruksjons-cache: 32 KB, 8-veis sett-assosiativ cache.
Data-cache: 32 KB, 8-veis sett-assosiativ.

L2-cache. Hver kjerne har:
Enhetlig cache: 256 KB, 8-veis sett-assosiativ cache.

L3-cache. Kjernene deler på en:
Delt enhetlig cache: 8 MB, 16-veis sett-assosiativ cache.

2.2. Sammenkobling av delene

I de to siste leksjonene har vi sett på primærminne og på cache. Disse samarbeider om å fore CPU med instruksjoner og data. I neste leksjon skal vi se nærmere på hvordan dette foregår. Denne delen av datateknikk kalles gjerne Systemarkitektur.

Vedlegg A. Mer om mapping-funksjoner.

I leksjonen har vi sett at når en blokk kopieres til cachen, så brukes en mappingfunksjon til å bestemme hvor i cachen blokka skal ligge.

Nå skal vi se nærmere på de tre viktigste mappingfunksjonene, nemlig full-assosiativ cache, ikke-assosiativ cache (direkte mapping) og sett-assosiativ cache.

Det vi skal se på er hvordan CPU greier holde oversikt over hvilke blokker som til en hver tid ligger i de ulike cache-linjer. Vi må huske at informasjonen om hvilken blokk som ligger i en linje finnes i tagen til linjen.

Full-assosiativ cache

Her kan en blokk legges i en hvilken som helst linje. Ulempen med metoden er at når CPU skal hente data, så vet den ikke på forhånd hvilken linje den kan forvente å finne blokka.. Derfor må cache sjekke tag-en til hver eneste linje. I praksis tar dette for langt tid. Vi tar en titt på dette tilfellet likevel.

Tag ved full-assosiativ cache: et enkelt eksempel

Vi skal se hvordan adressene til den blokken som ligger i cache kan brukes til å bygge opp tagen. Vi begynner med et enkelt eksempel for å se prinsippet - og deretter ser vi på et mer realistisk eksempel.

I Figur 5 ser vi et minne som består av 16 lokasjoner. Vi antar at cache bruker en blokk-størrelse på 2 lokasjoner. Da kan minnet også sees på som 8 blokker. Cachen har to linjer som hver har plass til en blokk. Vi ser at blokk 0 er kopiert til linje 0 i cachen.

Hvordan skal tagen se ut? Jo, se på Figur 6 og legg merke til at de tre mest signifikante[1] bitene i adressene utgjør blokknummeret. Derfor kan vi bruke disse til tagen. I dette tilfellet benyttes altså de tre mest signifikante bitene til tag. Den minst signifikante biten angir bare bytenummer innenfor blokken.

Tag ved full-assosiativ cache: et realistisk eksempel

La oss nå se på et mer realistisk eksempel. Vi antar en maskin med følgende data:

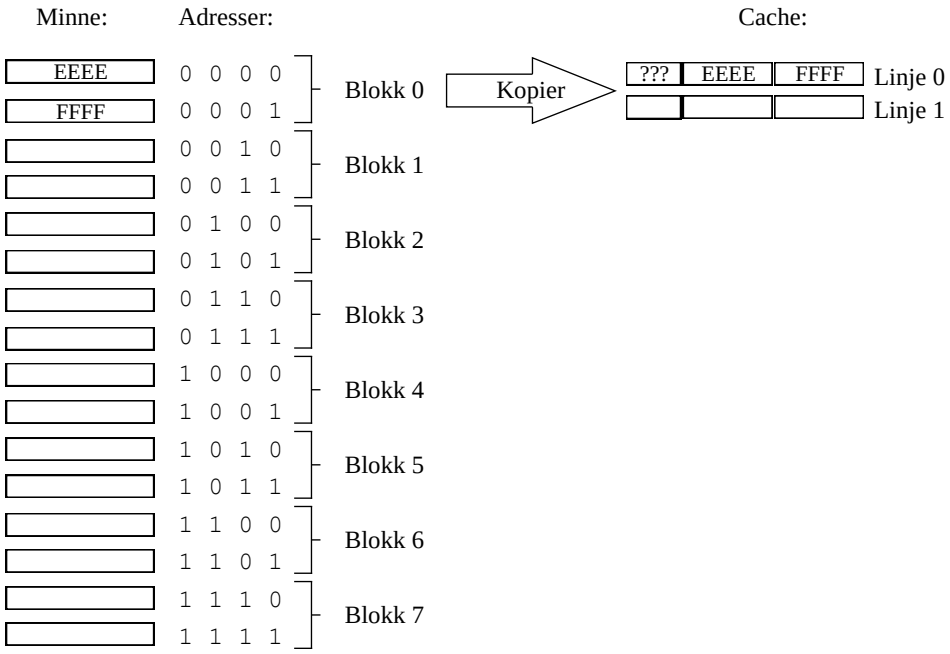
Primærminne : 4 MB (krever 22 bits adresser)
Cache størrelse : 4 KB
Blokkstørrelse : 4 B

Da vil antall linjer i cachen være: C = 4 KB/ 4 B = 1 K = 1024

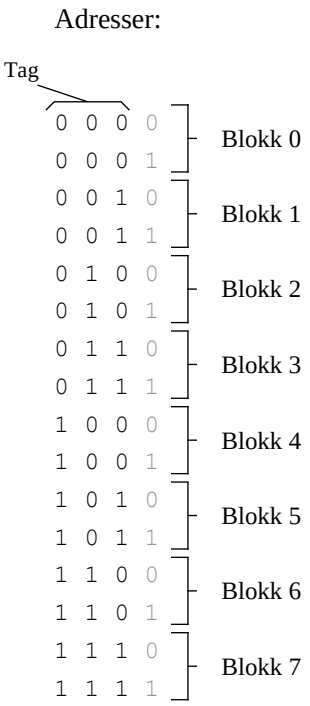
Isteden for å se på minnet som 4M lokasjoner à en byte, kan vi se på det som 1 M blokker som hver er 4 bytes. I en gitt linje kan det ligge et vilkårlig av disse 1M blokkene. For å identifisere hvilken blokk som er inne i en linje må vi bruke de 20 mest signifikante bitene i adressen som tag. De to laveste bitene, ...xx00, ...xx01, ...xx10 og ...xx11, skiller bare mellom

[1] Den mest signifikante biten (MSB - Most Significant Bit) er biten helt til venstre i et bitmønster. Den minst signifikante biten (LSB - Least Significant Bit) er biten helt til høyre. Mest signifikant betyr at en endring av bit-verdien (fra 0 til 1 eller motsatt) vil gi størst forandring i bitmønsterets verdi. Minst signifikant betyr at en endring av bit-verdien vil gi minst endring i bitmønsterets verdi.

blokkens 4 bytes, som alle hentes inn samlet. De er derfor ikke nødvendige for å skille mellom ulike blokkene i cachen.



Figur 5 En forenklet fremstilling av minne og cache. På figuren er blokk 0 kopiert til linje 0 i cachen, og spørsmålet er: hvordan skal Tag se ut for at vi skal kunne identifisere hvilken blokk som ligger der? Tagen er merket med tre spørsmålstegn.



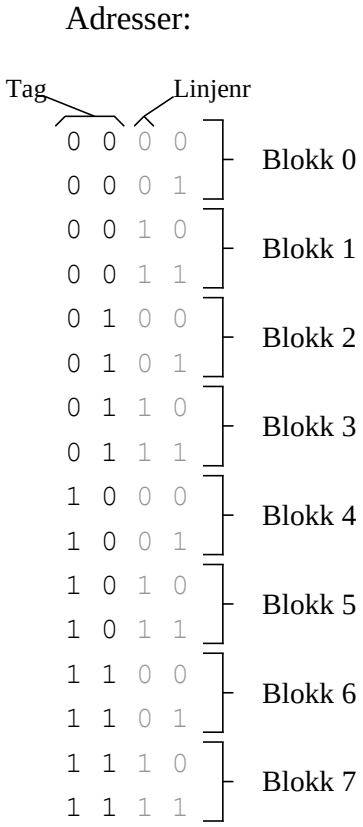
Figur 6 De tre mest signifikante bitene utgjør blokknummeret, og kan brukes til tag. Den minst signifikante biten angir bare bytenummer innenfor blokken.

Ikke-assosiativ cache (direkte mapping)

For å slippe å ha sammenligningslogikk for hver eneste linje (1024 sammenligningskretser i eksempelet ovenfor) kan vi isteden la hver blokk i minnet ha en *fast* plass i cache. Dette kalles *ikke-assosiativ cache*, eller *direkte-mappet cache*. Da kan en blokk bare legges på en bestemt linje i cachen, men det er mange blokker som må dele på hver linje. Nå blir det mye enklere for cachen. Den vet på forhånd hvilken linje den skal sjekke for å finne ut om blokka som etterspørres ligger i cache.

Tag ved ikke-assosiativ cache: et enkelt eksempel

Vi ser igjen på det forenklede tilfellet med bare 16 minnelokasjoner, og to cache-linjer. Vi bestemmer oss for at blokkene 0, 2, 4 og 6 (like nummer) skal "tilhøre" linje 0 i cachen, mens blokkene 1, 3, 5 og 7 (odde nummer) skal tilhøre linje 1. Som vi ser av Figur 7 vil bit nummer 1 nå angi hvilken linje blokken skal ligge i, mens bit nummer 0 fortsatt angir bytenummer innenfor blokken. De øvrige bitene brukes til tag. Når CPU skal sjekke om en blokk ligger i cachen, vet den om den skal sjekke en like eller odde blokk - og kan gå direkte til riktig linje. Derfor er det tilstrekkelig med bare to bits til tag i dette tilfellet.



Figur 7 Tag for ikke-assosiativ cache. Vi ser at de to mest signifikante bitene brukes til tag. Bit nummer 1 er linjenummer der blokken skal legges, og bit nummer 0 (den minst signifikante) angir fortsatt bare bytenummer innenfor blokken.

Eksempel på tag:

Hva blir tag hvis blokk 6 ligger i cachen? Jo, vi ser fra Figur 7 at da blir tag lik 11_2., som er de to mest signifikante bitene i adressene til blokk 6.

Enn hvis blokk 3 ligger i cachen? Da ser vi på samme måte at tag blir 01_2.

Men hvordan sjekkes det om en blokk ligger i cache?

Når CPU etterspør en minnelokasjon må elektronikken altså først sjekke om denne minnelokasjonen allerede er kopiert til cachen. La oss se på et eksempel med vår enkle maskin med 16 ord i minnet og to-linjers cache:

Anta at blokk 6 og 3 ligger i cachen, slik som i rammen rett ovenfor. Da er tag lik henholdsvis 11_2 og 01_2. Blokk 6 ligger i linje 0 siden 6 er et partall. Blokk 3 ligger i linje 1 siden 3 er et odde tall.

Eksempel på treff:
Anta at CPU vil lese lokasjon 7 fra minnet. Cachen vet at dersom lokasjon 7 (med adresse 0111_2) finnes i cachen, så ligger den i linje 1 (siden bit nr 1 i adressen er lik 1). Derfor vil elektronikken sjekke linje 1. Tagen til linje 1 er lik de to mest signifikante bitene i adressen som CPU etterspør. Vi har et treff, og cachen returnerer den rette lokasjonen fra cache-linjen.

Eksempel på bom:
Anta at CPU etterspør lokasjon 14 (=1110_2) fra minnet. Cachen vet at dersom lokasjon 14 finnes i cachen, så ligger den i linje 1 (siden bit nr 1 i adressen er lik 1). Derfor vil elektronikken sjekke linje 1, den finner at tagen til linje 1 ikke er lik de to mest signifikante biten i adressen som CPU etterspør. Vi har en bom, og elektronikken må helt ut i minnet for å hente en ny blokk inn til linje 1.

Tag ved ikke-assosiativ cache: et realistisk eksempel

Vi fortsetter med det mer realistiske eksempelet. Vi hadde et minne på 4 MB. Det vil si at hver adresse er 22 bits. Vi antar 4 bytes størrelse på cache-blokkene. De to laveste adressebitene identifiserer fortsatt hver enkelt byte i blokka. Nå lar vi de *neste* bitene bestemme hvor en blokk skal legges når den kopieres inn i cachen. Som vi husker hadde vi 1024 linjer (C = 1024):

Blokk 0 (byte 0...0000_{16} til 0...0003_{16} i minnet) har plass i linje 0
Blokk 1 (byte 0...0004_{16} til 0...0008_{16}) har plass i linje 1
:
:
:
Blokk 1023 (byte 0...$0FFC_{16}$ til 0...$0FFF_{16}$) har plass i linje 1023.

Blokk 1024 (byte 0...1000_{16} til 0...1003_{16}) har plass i linje 0, og deler altså plass med blokk 0
Blokk 1025 deler linje 1 med blokk 1 osv.

Siden linje 0 bare brukes av blokk 0, 1024, 2048, ..., og linje 1 bare brukes av blokk 1, 1025, 2049... osv. er det ikke nødvendig å ta med adressebits 2 til 11 i tag-feltet. Tag-feltet er bare på de 10 mest signifikante bits, det vil si bit 12 til 21.

Eksempel på bruk av ikke-assosiativ cache

La oss se på et eksempel hvor CPU vil lese byten på den 22-bits adressen (binært): 0000101011100000110101. Da deles adressen opp slik:

0000101011 1000001101 01
tag-verdi Linjenr byte-nummer innen blokk (brukes ikke i tag)

Ordet hører altså hjemme i linje 525 (1000001101 binært). Når akkurat denne blokken er inne i cache, vil tag-feltet på denne linjen ha verdien 43 (0000101011 binært).

Fordelen med en direkte-mapping-cache er foruten plassbesparelsen ved kortere tag-felt at det er tilstrekkelig med en eneste sammenlignings-krets for å finne ut om det riktige ordet ligger i cache. Det er heller ikke nødvendig å velge noen utskiftingsalgoritme, siden det er absolutt gitt hvilken linje som skal brukes.

Sett-assosiativ cache

For å unngå trashing har vi sett at det er lurt å bruke en sett-assosiativ cache; blokkene som tilhører samme linje får flere linjer å dele på, så det blir plass til begge.

Som i en direkte-mappet cache brukes en del av minne-adressen for å bestemme hvor i cache data skal lagres, men nå bestemmer adressen bare hvilket *sett* data skal legges i og ikke hvilken linje i settet. Som alltid gjør vi det slik at de laveste adresse-bitene bare bestemmer byte-nummer innen blokken og kan ignoreres, men nå lar vi de neste bitene velge sett: Hvis vi har 256 sett vil blokk 0 caches til sett 0, blokk 1 til sett 1 ... blokk 255 til sett 255, blokk 256 til sett 0, blokk 257 til sett 1...

Vi har færre sett enn vi hadde linjer i en direkte-mappet cache. Derfor må vi bruke flere bits i tag-feltet - en bit mer ved en 2-veis cache, to bits mer ved 4-veis.

Realistisk eksempel på bruk av sett-assosiativ cache

Vi antar som før et minne med 1M blokker, og en cache med 1024 linjer. Men nå er cachen organisert som 256 sett á 4 linjer. Vi bruker de 8 bitene fra 2 til 9 i adressen til å velge cache-sett. La oss se på et eksempel hvor CPU vil lese byten på 22 bits adressen (binært): 0000101011100000110101. Da deles adressen opp slik:

000010101110 00001101 01
tag-verdi sett-nummer byte-nummer innen blokk (brukes ikke i tag)

Ordet hører altså hjemme i sett 13 (1101 binært), dvs. linje 52, 53, 54 og 55. Hvis ordet er inne i cache vil tag-feltet på en av disse fire ha verdien 175 (000010101110 binært).

Når CPU adresserer data undersøker den tag-feltet til alle linjer i settet for å se hvor ordet ligger (eller om det må hentes inn). Da er det nok med så mange sammenligningskretser som antall blokker i et sett. Når en blokk hentes inn må det velges en av de 2, 4,.... linjene i settet, og det er nødvendig å implementere en utskiftings-algoritme som velger et «offer».

En sett-assosiativ cache gir en rimelig god avveining mellom kompleksitet og ytelse. Kravene til elektronikk er moderate, og faren for kollisjoner relativt liten. De aller fleste nyere cache-design er 2-veis, 4-veis eller iblant 8-veis sett-assosiative.

Som en kontroll på at du forstår dette kan du jo prøve å løse følgende oppgave:

Ta utgangspunkt i følgende adresse:

01001100100100100 11011

Anta en sett-assosiativ cache slik den ble beskrevet i forrige ramme. Vis at:

1. Denne adressen tilhører sett nr $26_{16} = 38_{10}$.
2. At sett 38_{10} består av cache-linjene 152, 153, 154 og 155.
3. At tag til en av disse linjene må være $4C9_{16}$ dersom adressen ovenfor ligger i cache.