Kap. 5: Buss, hurtigbuffer og delt lager

- **Pensum (kursorisk i parantes)**
  - (5.2), 5.3.1-2, 5.4
- **Hurtigbuffer (repetisjon)**
  - Begreper og prinsipper
  - Splittet el. "unified" hurtigbuffer
  - "Write Through" (WT), eller "Write Back" (WB)
  - Fysiske el. virtuelle adr. i hurtigbufferet
  - Hovedlager-hurtigbuffer avbildning
  - Ytelse
- **Delt lager**
  - Lager-fletting
    - » Lav-ordens og høy-ordens lagerfletting
    - » Kombinert høy- og lav-ordens lagerfletting
    - » Avveining mellom båndbredde og feiltoleranse
  - Pipelined lager-aksess
- **Sekvensiell og svak konsistens**
  - "Lager-modeller"
  - Ordning av hendelser
  - Atomiske operasjoner
  - Sekvensiell og svak konsistens

Hurtigbuffer (Eng. cache) (repetisjon)

- **"Write Through" (WT)**
  - skriving i hurtigbuffer medfører skriving i hovedlager straks
- **"Write Back" (WB)**
  - venter med oppdatering av hovedlager før det er absolutt nødvendig
    - » hurtigbuffer-blokk hives ut
    - » ...og er modifisert ("dirty")
- **Hovedlager-hurtigbuffer avbildning**
  - direkte
  - fullt assosiativ
  - set-assosiativ
  - sektor avbildning
- **Ytelse**
  - blokkstørrelse
  - algoritmer for erstatning

Delt lager

Kap. 5.3.1. side 239

- **Lav-ordens lagerfletting**
  - Motivasjon; prosessor -- lager gapet (som øker)
  - Økt hastighet ved bruk av flere lagermoduler
  - Pipelined lager-aksess
- **Høy-ordens lagerfletting**
- **Kombinert høy- og lavordens lagerfletting**
  - Avveining mellom båndbredde og feiltoleranse

Båndbredde og feiltoleranse

m antall flettede lagermoduler

θ major cycle

t_1 gjennomsnittlig aksesstid for et ord i en vektor

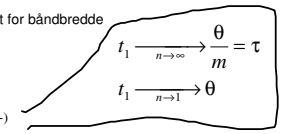
n vektorlengde

Hellermanns estimat for båndbredde

$B = m^{0.56} \approx \sqrt{m}$

Cragon

$t_1 = \frac{\theta}{m}(1 + \frac{m-1}{n})$



- **Kombinert høy- og lav-ordens lagerfletting**
  - Deler lageret inn i banker av moduler
  - Høyordens bit velger bank, mens lavordens bit velger modul innenfor valgt lagerbank
  - Etterhverandrefølgende ord er lagret innenfor (normalt) en lagerbank
  - Stort antall banker gjør at feil i en modul gjør en mindre del av lageret ubrukelig, dvs. økt feiltoleranse. Generelt lavere båndbredde, men lavere reduksjon ved feil.
  - Lite antall banker, gir stor grad av parallellitet i aksess til ord i en bank (fletting). Generelt større båndbredde, men større reduksjon ved feil.
- **Fig 5.17**

Kap. 5.4: Sekvensiell og svak konsistens

- **Kap. 5.4 i Hwang93 er vanskelig!**
  - Vanskelig i seg selv
  - Umodent (?), aktiv forskning pågår innen området
  - Stedvis mangelfullt forklart i boka
- **Målsetting**
  - Bli kjent med problemstillingen og en del grunnleggende prinsipper
  - Forstå intuitivt ulike sentrale modeller for lager-konsistens og forskjellen mellom disse
  - Bli kjent med den type definisjoner som ligger bak de ulike konsistens-modellene
  - **Full** forståelse av alle modellene vil **ikke** bli krevd så lenge bare Kap. 5.4. er pensum.
  - 45214 vil uansett ikke gjøre studentene "utlært" i dette temaet, men alt som foreleses kan være eksamensrelevant
- **Støttelitteratur:**
  - Michel Dubois, Christoph Scheurich og Fayé A. Briggs: *Synchronization, Coherence and Event Ordering in Multiprocessors*, , IEEE Computer, Februar 1988, side 9--21
  - Harold S. Stone: *High-Performance Computer Architecture,* Addison Wesley 1993, side 392--403 (+ side 367)

Ordning av hendelser, generelt

- **Strenge krav til ordning av hendelser i et system kan**
  - luke bort ulogisk oppførsel (nødvendig)
  - øke kontrollen med hva som skjer (diskutabelt)
    - » kan gi et enklere system, og kan dermed gi bedre ytelse
    - » vil ofte gi dårligere ytelse
- **Å løsne på kravene til oppførsel i et system kan ofte gi bedre ytelse**
- **Ønsker som må avveies mot hverandre;**
  - kontroll av hendelser
  - rom for ytelsesforbedring

Ordning av hendelser, (Banalt) eksempel

Eksempel-1:

- Program-A består av to instruksjoner A1(1) og A2(10) med varighet som angitt i parantes. Program-B består av to instruksjoner B1(10) og B2(1).
- Anta at A og B skal utføres på en multiprosessor med 2 prosessorer slik:

```
Prosessor-1:    Prosessor-2:
A1(1)           B1(10)
A2(10)          B2(1)
```

- a) Anta en *"streng ordning av hendelser"* i systemet som krever at begge prosessorene synkroniseres etter hver instruksjon. Dette kan være nødvendig f.eks. hvis et resultat av B1 påvirker A2, vi vil da bestemme at B1 skal utføres før A2. Beregningen tar da 20 tidsenheter. (+ synkroniseringsoverhead)
- b) Anta nå at program A og B er uavhengige og at vi eller systemet innser at vi derfor kan sløyfe en slik synkronisering. Beregningen kan da utføres på 11 tidsenheter. Dette er oppnådd fordi vi har tillatt en *"svakere ordning av hendelsene"*.
- Ved å tillate asynkron oppførsel istf. synkron oppførsel kan vi her i tilfelle b) få bedre ytelse.

- **Moral: En bør bare synkronisere der det er nødvendig. (Mere generelt; "ikke vær unødvendig streng")**

Ordning av hendelser, modell for lager konsistens

- **Se figur 5.19**
  - I et en-prosessor-system hvor lageret benyttes av bare en prosessor, vil program-rekkefølgen (PO) være den samme som rekkefølge av aksesser til lageret (MO).
  - I et fler-prosessor-system med felles lager vil ulike "flettinger" av de ulike program-rekkefølger PO1, PO2, ... gi ulike rekkefølger av aksesser til lageret.
    - » Hvis prosessorene samarbeider via delte data i felles lager vil ulike "flettinger" gi ulik oppførsel. Det vil som oftest være uakseptabelt
    - » Hvis programmene er komplett uavhengige er den aktuelle flettingen uinteressant; systemet er da ekvivalent med et antall prosessorer som opererer på hvert sitt lokale lager.
- **Lager modell**
  - Oppførselen til et felles lager system slik det observeres fra prosessorene kalles en *lager modell* (el. modell for lager konsistens)
  - I valg av lagermodell må en gjøre et kompromiss mellom en sterk modell som gir minimale restriksjoner for programvaren og en svakere modell som gir muligheter for mer effektiv realisering

Endringer av instruksjons-rekkefølge

- **På en-prosessor maskiner**
  - En vanlig prosessor kan utføre instruksjonene i et program i en annen rekkefølge enn hva som intuitivt forventes av programmereren, ved at kompilator el. prosessor endrer rekkefølge
  - Analyse (kompilator) og kontroll-logikk (prosessor) må sjekke at disse endringene ikke bryter data- og kontroll-avhengigheter mellom instruksjonene for at programmet skal utføres på en korrekt måte
  - Dette gjøres for å få mer effektiv kode ved f.eks. bedre utnyttelse av funksjonelle enheter
- **Korrekt utførelse på multiprosessor med delt lager er vanskeligere:**
  1) Rekkefølgen på instruksjoner i de ulike programmer er ikke spesifisert i et parallelt MIMD program. Hvis ingen eller få synkroniserings-punkter finnes i programmet vil antall mulige instruksjons-flettinger bli meget stort
  2) Hvis en tillater endret rekkefølge på instruksjoner innen hvert enkelt delprogram (for å bedre ytelsen som nevnt ovenfor) blir antall flettinger enda større
  3) Hvis en tillater ikke-atomisk aksess av data (f.eks. et cache-basert system hvor ikke alle kopier av en variabel oppdateres umiddelbart) kan ulike prosessorer *observere* ulike flettinger under en og samme kjøring (dvs. en fletting). Dette øker antall observerte instruksjons-rekkefølger

Ordning av hendelser, Eksempel 5.9

Fig 5.19 c)

Prosessor-1:	Prosessor-2:	Prosessor-3:
a. A := 1	c. B := 1	e. C := 1
b. Print B,C	d. Print A,C	f. Print A,B

- Antar samtidig og asynkron utførelse av de tre program
- A, B og C er initielt = 0
- Print leser begge variablene udelelig (forenkler)
- Utskrift fra de tre programmene konkateneres i rekkefølgen P1, P2, P3
- Resultatet av programmet er en 6-bits binær-vektor, med 64 mulige verdier
- Hvis hver prosessor utfører sitt program i riktig rekkefølge, er to mulige resultat
  - » 001011 (a,b,c,d,e,f) / 111111 (a,c,e,b,d,f)
- Hvis prosessorer tillates å utføre sine programmer i endret rekkefølge, er et mulig resultat
  - » 00000 (b,d,f,a,c,e)
- Ut av 720 (= 6!) mulige ordninger (flettinger) av de 6 instruksjonene er det 90 som beholder individuell program-rekkefølge
  - » ingen av disse kan resultere i 000000
- Kan hver enkelt av de 64 mulige resultat-vektorene bli produsert av minst en av de 720 mulige instruksjons-flettinger ?
- Dette avhenger av om vi har et atomisk eller et ikke-atomisk lagersystem

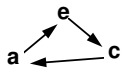
Atomisk eller ikke-atomisk lager-aksess

- **Atomisk lager**
  - En delt lager aksess er atomisk hvis oppdateringer i lageret "blir kjent i" alle prosessorer samtidig
  - Ulike realiseringer:
    - » 1) kreve at maksimalt en prosessor aksesserer en lager-modul av gangen, og ikke bruke private hurtigbuffere
    - » 2) opprettholde konsistens mellom hurtigbuffere ved å ikke avslutte skrive-operasjon mot eget hurtigbuffer før en vet at alle andre kopier i hurtigbuffere og kopi i hovedlageret er blitt oppdatert
    - » 3) som 2), men andre hurtigbufferkopier blir ugyldiggjort istedetfor oppdatert
- **Ikke-atomisk lager aksess**
  - kan f.eks. oppstå i et system med private hurtigbuffere der ugyldig-gjøringskommandoer ikke når fram samtidig i de ulike prosessorer
  - kan føre til at ulike prosessorer observerer ulike instruksjonsrekkefølger av en og samme program-kjøring

Atomisk lager-aksess, Eksempel 5.9 fortsatt

Prosessor-1:	Prosessor-2:	Prosessor-3:
a. A := 1	c. B := 1	e. C := 1
b. Print B,C	d. Print A,C	f. Print A,B

- **Tilfelle-A: anta atomisk lager-aksess:**
  - Resultatet 011001 kan ikke produseres, fordi
    - » 01xxxx betyr at P1 observerer at C er oppdatert men ikke B, dvs. (e) er utført før (c)
    - » xx10xx betyr tilsvarende at (a) er utført før (e)
    - » xxxx01 betyr tilsvarende at (c) er utført før (a)
    - » hvilket er umulig



Ikke-atomisk lager-aksess, Eksempel 5.9 fortsatt

Prosessor-1:	Prosessor-2:	Prosessor-3:
a. A := 1	c. B := 1	e. C := 1
b. Print B,C	d. Print A,C	f. Print A,B

- **Tilfelle-B: Anta ikke-atomisk lager-aksess, samt at**
  - instruksjons-rekkefølgen blir a,c,e,b,d,f
  - når P1 utfører (b) er P1's egen kopi av B ikke blitt oppdatert, mens P1's egen kopi av C er blitt oppdatert. Derfor skriver P1 ut 01.
  - Tilsvarende kan det skje at P2 skriver ut 10 og P3 skriver ut 01
  - Resultatet 011001 er mulig i et slikt system
    - » hvert enkelt program ble utført i riktig rekkefølge
    - » prosessorene observerer ulik rekkefølge, f.eks. tror P1 at (e) er utført før (c), som ikke stemmer
    - » samlet kan man si at prosessorene observerer situasjonen vi så på i tilfelle-A, og som er inkonsistent (umulig)
    - » enkeltvis ser prosessorene ingen feil
  - Om tilfelle-B er feil eller riktig avhenger av hvilke forventninger programmeren har av systemet
  - Ofte vil programmereren forvente sekvensiell konsistens, og tilfelle-B er da ikke-korrekt oppførsel

Sekvensiell konsistens (SC)

- **Uformell definisjon [Hwa93 side 252]:**
  - I en lager modell med sekvensiell konsistens vil lese, skrive og synkroniseringsoperasjoner mot lageret fra prosessorene "se ut som" (Eng. appear) at de blir utført serielt i en enkelt global (lager) rekkefølge som stemmer med den individuelle program rekkefølgen i hver enkelt prosessor
- **Leslie Lamport's definisjon (1979)**
  - "A multiprocessor system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program".
- **Sterk ordning av hendelser ≈ sekvensiell konsistens**

Betingelser for sekvensiell konsistens

- En nødvendig og tilstrekkelig betingelse for at et system med **atomisk lager-aksess** skal være sekvensielt konsistent er at aksesser til lageret blir utført i den rekkefølge programmereren mener, dvs. i program-rekkefølge for hvert individuelt program. [Dubois et.al.86]
- Som eksemplet over (Tilfelle-B) viste er opprettholdelse av individuell program-rekkefølge ikke tilstrekkelig for å garantere sekvensiell konsistens i et system med **ikke-atomisk lager aksess**.
- Det er bare i atomiske lager systemer at vi kan ha en **sterk ordning** av "hendelser i lagersystemet" slik at program-rekkefølgen kan garanteres å være konsistent med rekkefølgen av aksesser til det delte lageret
- **Ikke-atomiske lagersystemer kan være ønskelig** for å oppnå bedre ytelse, her må vi nøye oss med en svak ordning av hendelsene

Sterk ordning er lite skalerbart !(?)

- **Fig 5.20**
  - Anta at vi har private hurtigbuffere, eller på en annen måte flere distribuerte kopier av en variabel
- **Hvordan sikre sterk ordning?**
  - En "oppdatering må formidles til hele systemet" før vi kan "gå videre"
    - » dette vil ta tid i store distribuerte systemer, lite skalerbart
    - » dette kan håndteres effektivt for prosessorer knyttet til en felles buss, men det er en lite skalerbar arkitektur
- **Finnes det andre løsninger?**

Svak konsistens

- **Sekvensiell konsistens er enkelt men unødvendig strengt**
- **Store deler av et parallelt program kan utføres lokalt i en prosessor uten å påvirke andre deler av systemet**
  - slike deler bør skje for full hastighet uten unødvendig grad av kontroll (Jfr. eksempel-1)
- **Sterk ordning av aksess til synkroniseringsvariable og gjen-opprettelse av koherens ved synkroniseringspunkter er alt som kreves**
  - "løsere system" -> mer effektivt
  - men ikke lettere å realisere
- **Dette kalles svak ordning av hendelser**
- **Kunsten er:**

  ... disallow all interleavings that will result in incorrect outcomes while not being overly restrictive
- **Synkroniseringsvariable må kunne oppdages av maskinvaren**

Svak konsistens, eksempel

- **Buffring av skriveoperasjoner kan tillates for å gi bedre ytelse**
  - Men programmereren må være klar over "farene" ved dette, og legge inn synkroniseringskode der det er nødvendig for å sikre riktig samarbeid mellom prosessorer
- **Eksempel:**
  - "Dobbelt-buffer"
  - Anta at prosessor A skal sende data hurtig via delt lager til prosessor B.
    - » Vi benytter 2 buffere, hver på 100 ord, og systemet veksler mellom to faser
    - » I fase-1 skriver A til buffer-1 mens B leser fra buffer-2
    - » I fase 2 bytter buffrene roller
    - » I fasene kan begge prosessorene operere på full fart med bruk av privat hurtigbuffer og utsatt tilbakeskriving (Eng. Write Back)
    - » Vi trenger bare synkronisere ved bytte av faser
    - » Atomisk lager aksess (som fig 5.20) ville her ha gitt vesentlig redusert ytelse
  - ***Programmereren tar over ansvaret for at programmet blir riktig og oppnår bedre ytelse gjennom svak konsistens***

Svak ordning / Svak konsistens

- **Det finnes mange varianter av svak ordning**
  - Fig. 9.8
- **DSB-modellen (Dubois, Scheurich & Briggs) (noe omskrevet)**
  - 1) Synkronisering må gjøres helt ferdig
  - 2) Gjør ferdig alle lageraksesser før synkroniseringen starter
  - 3) Synkroniseringsaksesser er sekvensielt konsistente
- **Fig 5.21, TSO svak konsistens modell**
  - Total Store Order, definert for SPARC arkitektur
- **.... andre ?**

Oppsummering

- ***"The weak consistency model may offer better performance than the sequential consistency model at the expense of more complex hardware/software support and more programmer awareness of the imposed restrictions"***
- **aktiv forskning, sentrale temaer**
  - bør vi betrakte prosesser istf. prosessorer?
  - bør også I/U-lokasjoner regnes med i lagermodellen?
  - m.m.