H O V E D O P P G A V E

OBJEKTBUFFER I OBJEKTORIENTERTE DATABASESYSTEM

Utført av Kåre Fosse

NTNU Trondheim
Norges teknisk-naturvitenskapelige universitet

28. januar 1998

FORORD

Denne rapporten dokumenterer resultatene fra gjennomføringen av hovedoppgaven i forbindelse med min sivilingeniørutdannelse ved *Norges teknisk-naturvitenskapelige universitet (NTNU), Fakultet for fysikk, informatikk og matematikk*. Hovedoppgaven ble utført i tidsrommet fra 22. september til 28. januar 1998.

Oppgaven gikk ut på å studere nye metoder for bufferhåndtering i databasesystem. Den har vært en kombinasjon av litteraturstudium, implementasjon og simulering. Faglærer for hovedoppgaven har vært professor Kjell Bratbergsengen ved *Institutt for datateknikk og informasjonsvitenskap.*

Hovedoppgaven ble gitt i forbindelse med doktorgrads arbeidet til veileder Kjetil Nørvåg om skriveoptimaliserte objektorienterte databasesystem. Jeg vil takke for den assistansen han har gitt meg, som har vært et viktig bidrag til oppgaven, samt at jeg har fått benyttet hans store litteratursamling om temaet. Jeg håper at arbeidet mitt er til nytte for han.

Trondheim, 28. januar 1998

Kåre Fosse

INNHOLD

FIGURLISTE

SAMMENDRAG

Denne oppgaven handler om bufferhåndtering i databasesystem. For å oppnå høy ytelse i et databasesystem, enten det er snakk om relasjonsdatabaser eller objektorienterte databaser, er den mest kritiske komponenten bufferhåndtereren. Den vanlige måten å implementere denne komponenten på er som et sidebuffer. Dette er en god metode så lenge sammenklyngingen, av de objektene som brukes mest, på disken er god. Dersom den ikke er det, blir ikke bufferminnet godt utnyttet fordi objekter som nesten ikke brukes vil da måtte buffres i hovedminnet. Motivasjonen for å bruke en objektbasert bufferhåndterer ligger nettopp i dette poenget. Da kan bufferminnet utnyttes bedre ved å buffre de mest brukte objektene enkeltvis. Buffring av deler av databasen i hovedminnet er spesielt gunstig ved høy lokalitet i referansene, og lokalitet er et krav dersom objektbuffer skal ha noen mening. I dette prosjektet har derfor sammenhengen mellom aksessmønstre, minneutnyttelse og ytelse i et databasesystem som bruker objektbuffer i steden for sidebuffer blitt studert.

For å kunne sammenlikne ytelsen har jeg implementert en buffersimulator som kan simulere både objektbuffer og sidebuffer. Hovedvekten er lagt på de nederste nivåene i et databasesystem. De omfatter fysisk I/O (Input/Output), bufferhåndtering og lagringsstrukturer. Analytisk studie av ytelse blir for komplisert. Å implementere et sidebuffer er relativt rett fram. Som sideutskiftingsalgoritme ble LRU (Least Recently Used) valgt, fordi den er regnet som en av de beste til å utnytte lokalitet i referansene. Dessuten viste eksperimenter at den ga en reduksjon i I/O på opptil 19 % i forhold til klokkealgoritmen. Når det gjelder objektbufferet, er ikke ting fullt så enkelt lenger. Her ble klokkealgoritmen foretrukket som objektutskiftingsalgoritme på grunn av at minnekostnadene ble for store med LRU. LRU må ha to pekere per objekt, mens klokkemetoden bare trenger ett bit. Den endelige versjonen av objektbufferet benytter seg av et underliggende sidebuffer for å få god kald ytelse. I tillegg ga dette en noe enklere implementasjon. Den relative størrelsen mellom objektbufferet og det underliggende sidebufferet endres dynamisk ettersom objektbufferet fylles med objekter. Denne adaptive virkemåten ga en sterk reduksjon i sidefeil når bufferet var kaldt.

Hovedoppgaven til objektbufferet dersom det skal være brukbart, er dynamisk minneallokering, men med den forskjellen i forhold til tradisjonell minnehåndtering, som for eksempel *malloc()* og *free()* fra standardbibliotekene i ANSI C, at objekter kan kastes ut når bufferet er fullt for å frigi minne. En minneallokator må holde orden på hvilke deler av minnet som er i bruk og hvilke deler som er ledige. De ledige minneblokkene lenkes sammen i ei ledigliste. Ved å splitte lediglista i flere lister basert på blokkstørrelsen og legge inn kontrollinformasjon i halen på ledige blokker, har det lykkes å utvikle en minneallokator som tildeler og frigir bufferplass på konstant tid. Splitting av lediglista pluss sammenslåing av ledige naboblokker medførte også lav fragmentering. Flere tester viste en minneutnyttelse på 91 %. Ytterligere optimalisering kan ligge i å minske den interne fragmenteringen, som nå er 14 byte per objekt, og utvikle bedre allokeringsmetoder for små objekter.

Ytelsen, gitt ved antall sidefeil, ble målt ved å implementere to testapplikasjoner, OO1 og OO2. OO1 er en standard innen benchmarker, og har en database med deler og koblinger. Målingene omfatter innsetting og henting av deler og følging av koblinger mellom delene både for varmt og kaldt buffer. OO2 benchmarken spesifiserte jeg selv. Den var mye enklere å implementere og ga bedre kontroll over sentrale eksperimentelle parametrer som sammenklynging, modifikasjon og lokalitet. Foruten disse er også

bufferstørrelsen en viktig parameter. Hvis sammenklyngingen er maksimal har objektbufferet opptil 50 % flere sidefeil, med OO2 som benchmark, i varm tilstand for visse bufferstørrelser. Dette snur til en reduksjon på opptil 50 % eller 90 % når sammenklyngingen er henholdsvis 50 % og 10 %. Objektbufferet får problemer hvis objektene oppdateres. Problemet er at objektets hjemmeside først må leses inn, noe som medfører en økning i sider lest i forhold til sidebufferet som jo allerede har sida inne. Et eksperiment med 25 % modifikasjon og 10 % sammenklynging viste en reduksjon i I/O på 50 % ved bruk av objektbuffer, som er et vesentlig dårligere resultat enn uten modifikasjon. Sidebufferet hadde stort sett alltid best kald ytelse.

RAPPORTOVERSIKT

Når det gjelder konvensjonene som er benyttet, settes funksjonsnavn i kursiv og etterfølges av parenteser, for eksempel *debug()*. Kursiv er også brukt for variabelnavn. Skrivemaskinsfont benyttes for klassedefinisjoner, pseudoalgoritmer og kodestubber. Hakeparenteser settes rundt en litteraturreferanse.

Rapporten er organisert i de følgende seks kapitlene og tre vedleggene:

- 1. OBJEKTORIENTERTE DATABASESYSTEM: I dette innledende kapittelet tar vi en kikk på objektdatabaser. Disse databasene håndterer komplekse objekter, lange transaksjoner, datatyper for lagring av bilder eller store tekstposter og applikasjons spesifikke operasjoner. Selv om objektbuffer kan tilpasses relasjonsdatabaser, er det først og fremst til objektdatabaser det egner seg best.
- 2. PROBLEMSPESIFIKASJON: Gode løsninger kommer først når problemene er skikkelig forstått. Derfor vil dette kapittelet beskrive den teoretiske bakgrunnen for problemet, og gi referanser til andre arbeider som er relevante. Noen av problemstillingene er utvikling av en rask minneallokator med lav fragmentering. Det er også viktig med gode utskiftingsalgoritmer.
- 3. DESIGN OG IMPLEMENTASJON AV SIMULATOREN: Dette kapittelet bør være lett tilgjengelig når kildekoden leses. Det inneholder en beskrivelse av den interne strukturen på simulatoren, og datastrukturene og algoritmene som er brukt i implementasjonen. Alle klassene og klassefunksjonene blir gjennomgått. Utvalgte algoritmer vises med C liknende pseudokode, og figurer er med på å klare opp i datastrukturer.
- 4. BENCHMARKENE: To benchmarker er implementert for å teste ytelsen til simulatoren, OO1 og OO2. Kapittelet gjennomgår hvordan de er implementert. Testprogrammene muliggjør omfattende testing med varierende forsøksbetingelser for sammenklynging, lokalitet og oppdateringer.
- 5. SIMULERINGSRESULTATER: Her blir det en gjennomgang av ytelsesdataene som ble samlet inn i forbindelse med benchmark kjøringene. Presentasjonsformen blir mye grafer som kommenteres og analyseres. Ytelsen måles i antall sidefeil.
- 6. KONKLUSJON: Tar for seg de erfaringene som er gjort og hvilke slutninger som kan trekkes, og skisserer noen muligheter for videre arbeid.
- VEDLEGG A: SKRIVEOPTIMALISERTE DATABASESYSTEM: Dr. ing. student Kjetil Nørvåg sitt arbeid dreier seg om dette, og simulatoren ble derfor utvidet til å kunne simulere skriveoptimalisering. Men det ble ikke brukt mye tid på denne utvidelsen. Det medførte at skriveoptimalisering ikke simuleres særlig realistisk, og dataene i dette kapittelet må tas med en klype salt.
- VEDLEGG B: OO2 MED 75 % MODIFIKASJON: Testkjøringen med 10 % sammenklyngig og 75 % modifikasjon er inkludert i rapporten for å tilfredsstille de spesielt interesserte.
- VEDLEGG C: KILDEKODE: For at det skal være mulig å arbeide videre med oppgaven, er hele koden til simulatoren inkludert i rapporten. Koden er skrevet i C++ som gir mulighet for en ryddig organsisering. Det at jeg i tillegg har lagt inn en god del med kommentarer og konsekvent følgt en

programmeringsstandard, har forhåpentligvis resultert i lettlest kode. For å få koden feilfri, har jeg gjennomført uformelle kodeinspeksjoner og tester, samt brukt feilfinningsfunksjonene *assert()* og *debug()*.

Denne rapporten er ment å leses i den rekkefølgen som kapitlene er presentert. Representative litteraturreferanser ligger til slutt i rapporten sammen med en stikkordliste.

1. OBJEKTORIENTERTE DATABASESYSTEM

I dette kapittelet skal vi kort se på objektorienterte databaser. Hva er forskjellen på relasjonsdatabaser og objektorienterte databaser? En grundigere gjennomgang av denne nye typen av databasesystem finnes i [Cattell94]. Vi ser også på spørsmål omkring implementasjon av et objektorientert databasesystem, fordi det er sentralt i forhold til oppgaven, med spesiell vekt på et konkret system, O_2. Det blir ingen omfattende gjennomgang, siden lite er blitt publisert om komplette implementasjoner.

1.1 Generelt

Objektorienterte databasesystem (ODBMS er en vanlig forkortelse på engelsk) er foreslått som svaret på de nye utfordringene innen utvikling av komplekse applikasjoner innenfor områder som CAD (Computer Aided Design) og bilde-, grafikk- og multimediadatabaser, hvor de klassiske relasjonsdatabasene har vist svakheter. I steden for bare et høynivåspråk som SQL (Structured Query Language) for datamanipulering, integrerer derimot et ODBMS databasefunksjoner med objektorienterte programmeringsspråk, slik som C++. Dette gjør det unødvendig å lære et separat DML (Data Manipulation Language), fjerner behovet for å kopiere og oversette data mellom database- og programmeringsspråk representasjoner, og støtter rask aksess til komplekse objekt. I relasjonsdatabaser blir ofte et komplekst objekt spredd over flere tabeller. Når dette så skal settes sammen igjen, kreves flere tabell foreninger med sammenlikninger av primærnøkler og fremmednøkler. ODBMS'et utvider språket med transaksjoner, samtidighetskontroll, recovery, assosiative spørringer og sikkerhet. Andre fordeler med objektdatabaser sammenliknet med relasjonsdatabaser er versjonering, lange transaksjoner, nøstete transaksjoner, arving og brukerdefinerte datatyper. ODMG-93, beskrevet i [Cattell96], definerer en standard for ODBMS'er, inkludert både datamodell og språk. Den skal forsikre portabilitet på tvers av plattformer og produkter, slik SQL har gjort for relasjonsdatabaser. ODBMS'er lagrer et objekts attributter, ikke metodene som endrer et objekts tilstand. Hver metode har en signatur som identifiserer navnene og typene til argumentene, så vel som navn og type til returverdien. Den kjørbare koden for metodene lastes separat. Persistente objekter, det vil si objekter som har en levetid som er lengre enn et kjørende program, oppnås ved å bruke et avansert filsystem (les for eksempel om den objektorienterte lagringskomponenten i databasesystemet EXODUS beskrevet i [CareyDeWittRichardsonShekita86], eller om Texas i artikkelen [SinghalKakkadWilson92]). For optimal ytelse må objektet og eventuelt andre refererte objekter være residente i hovedminnet. Boka [ElmasriNavathe94] inneholder et ganske bra kapittel om objektorienterte databaser.

1.2 Implementasjon

Hvert objekt får en unik identifikator (OID) av ODBMS'et ved oppretting. ID'en legges inn i en identifikatorindeks. En objektidentifikator kan være:

- Fysisk: Diskadressen, vanligvis et sidenummer og en offset eller plass på sida, hvor objektet er lagret er direkte gitt av ID'en. Dette er den mest effektive

representasjonen for raskt å komme til objektet, men vi får problemer dersom et objekt skal flyttes til et annet sted på disken eller ved skjemaendring.
- Logisk: ID'en, normalt et heltall på 32 bit eller mer, må oversettes for å finne diskadressen og gir derfor lavere ytelse.

Når en applikasjon refererer et objekt via dets OID, konverterer mange ODBMS'er den til en virtuell adresse. Denne konverteringen av OID til minneadresse kalles swizzling og er viktig for rask aksess til data i minnet. En annen viktig faktor er buffring av de mest brukte objektene i hovedminnet for å øke ytelsen. Buffring er av avgjørende betydning i det vanlige scenarioet hvor det er lokalitet i referansene eller hele arbeidssettet får plass i hovedminnet. Lokalitet vil si at en stor andel av forespørslene refererer en liten del av databasen. Den lille delen kalles arbeidssettet eller de varme objektene. Teknikker og problemstillinger forbundet med bufferhåndtering kommer vi tilbake til i neste kapittel.

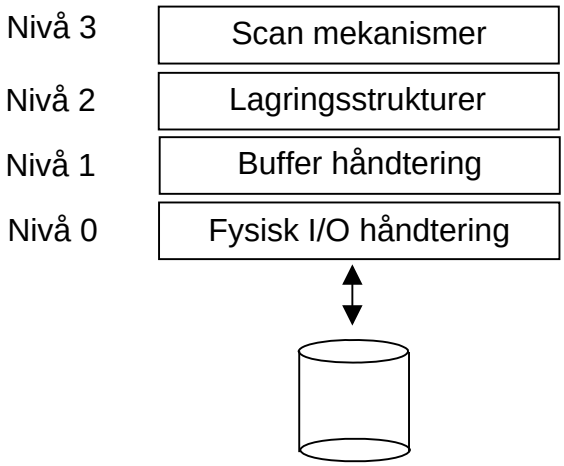
I tillegg til selve objektene må et ODBMS vedlikeholde kjøretidsinformasjon for de objektene som er i hovedminnet, for eksempel et bit som markerer om objektet er skittent. Objekter kan deles mellom flere brukere, og de må derfor utstyres med skrivelåser for å forhindre samtidige oppdateringer.

1.3 Et eksempel, O_2

O_2, se [BancilhonDelobelKanellakis92] (vær observant på at denne boka skildrer den pre-kommersielle versjonen av O_2) for omfattende dokumentasjon, er et kommersielt ODBMS som har vært i salg siden 1991, og som støtter lagring, henting og oppdatering av persistente objekter som kan deles av flere program. O_2Engine danner basisen og støtter flere programmeringsgrensesnitt: C, C++, O_2C - et objektorientert 4. generasjons språk, OQL - et objektorientert spørrespråk og O_2API - et lavnivå applikasjon programmeringsgrensesnitt. O_2Engine har en klient/tjener arkitektur, skiller mellom fysisk og logisk objekthåndtering og tilbyr samtidighets- og recoverytjenester, indekser, spørreoptimalisering og sammenklynging. Motoren består av de følgende tre lagene:

- **Schema Manager:** Overvåker oppretting, henting, oppdatering og sletting av klasser, metoder og globale navn. Den tar seg av arvemekanismene, sjekking av konsistensen til klassedeklarasjoner foruten klasseevolusjon.
- **Objekt Manager:** Håndterer objekter med identitet og leverer beskjeder til objekter. Den implementerer indekser og sammenklynging av relaterte objekter. Objektidentifikatorer er i O_2 implementert som fysiske diskadresser for å unngå overheaden med å oversette fra logisk til fysisk identifikator.
- **O_2Store:** Er en utvidelse av WiSS, et lagringssystem utviklet ved Universitetet i Wisconsin. Utvidelsene omfatter hovedsakelig samtidighetskontroll, recovery og støtte for flertråding. Som vist i Figur 1-1, består WiSS av fire forskjellige lag. Nivå 0 tar seg av aspektene rundt fysisk I/O, inkludert allokering av diskplass. Bufferhåndtereren danner nivå 1, og bruker les og skriv operasjonene til nivå 0 for å tilby buffret I/O til de høyere lagene. LRU brukes som utskiftingsalgoritme. Nivå 2 implementerer sekvensielle filer, B+-trær og lange dataobjekt. Det er også ansvarlig for å mappe referanser til poster til sider. Det øverste laget implementerer aksessmetoder for å scanne ei fil. O_2Store låser sider for samtidighetskontroll, og en write-ahead loggteknikk brukes for tilbakerulling og recovery.

Splittingen i en klientdel og en tjenerdel skjer på nivå 1 av WiSS. Tjenerkomponenten er en sidetjener. Den jobber bare med sider, kjenner ikke til objektstrukturene og har ansvaret for å hente sider effektivt når den blir instruert av klienten til å gjøre det.



Figur 1-1 WiSS arkitektur

Tjeneren buffrer en viss mengde sider for å redusere disk I/O, og aksesseres via et RPC (Remote Procedure Call) grensesnitt fra klientene. Mesteparten av funksjonaliteten ligger hos klienten. Klienten håndterer et lokalt sidebuffer.

1.4 Klient/tjener bufferarkitektur

Objektorienterte databasesystem implementeres vanligvis som en klient/tjener arkitektur over et nettverk. Eksakt hvordan funksjonaliteten fordeles mellom klienten og tjeneren varierer mellom de forskjellige implementasjonene. To grunnleggende framgangsmåter for splitting av bufferfunksjonaliteten er:

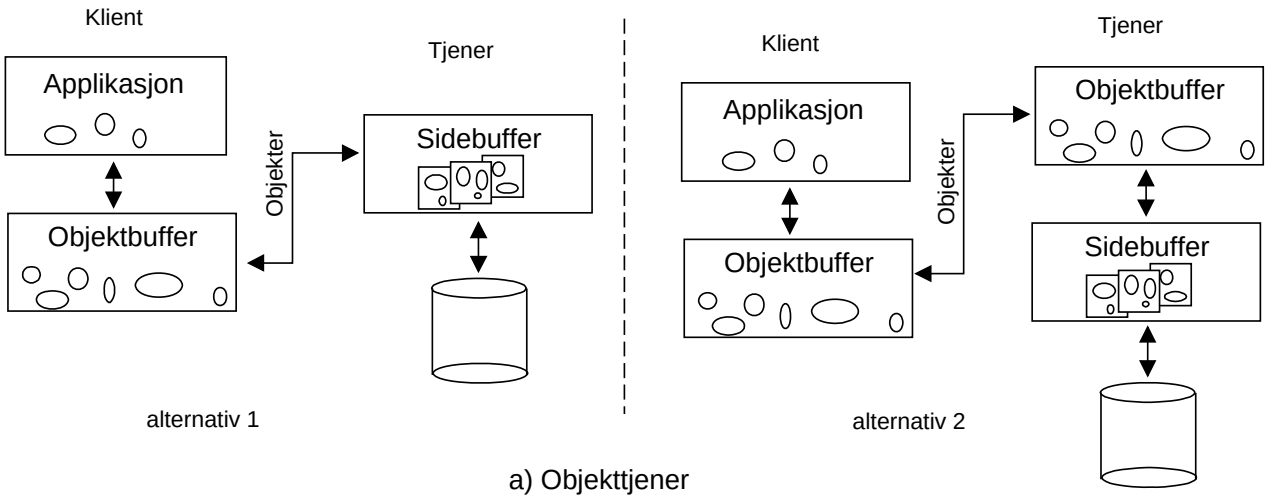
- Objekttjener, Figur 1-2 a). Klienten og tjeneren kommuniserer ved å flytte objekter mellom tjenerens database og klientens minne. Klienten vil da ha et objektbuffer, mens tjeneren har et sidebuffer pluss et eventuelt objektbuffer. Klienten anmoder objekter ved å spesifisere fysisk OID. Enheten for låsing blir i alternativ 1 et objekt.
- Sidetjener, Figur 1-2 b). Kommunikasjonen mellom tjeneren og klienten foregår med sider som enhet. Tjeneren er implementert med et sidebuffer, mens klienten har et sidebuffer og et eventuelt overliggende objektbuffer.

De fleste leverandørene bruker en sidetjener. I tilfeller hvor nettverks overheaden er høy og objektene er dårlig sammenklynget, vil en objekttjener være et bedre valg.

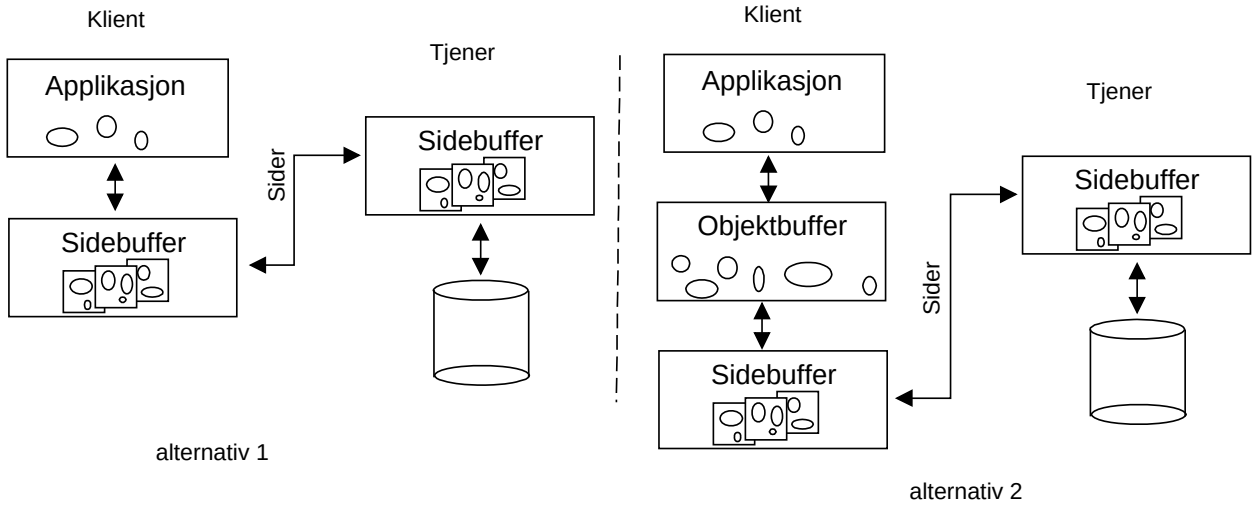
Formålet med dette prosjektet vil ikke være å avgjøre om det er best med en objekttjener eller en sidetjener, men:

- Dersom vi har en objekttjener, er da alternativ 2 bedre enn alternativ 1 som arkitektur?
- Dersom vi har en sidetjener, er da alternativ 2 bedre enn alternativ 1 som arkitektur?

Ytelsen til et ODBMS er påvirket av hvilke enheter som skal sendes mellom klienten og tjeneren, hvordan et objekt skal representeres på disken, hos tjeneren og hos klienten, hvilken transportprotokoll som benyttes, hvordan funksjonaliteten fordeles mellom klienten og tjeneren; pluss et utall andre faktorer. Men den viktigste faktoren er allikevel bufferhåndteringen, og den vil isolert sett studeres og ikke i sammenheng med de faktorene nevnt ovenfor. Dersom ikke denne begrensningen innføres, ville ting fort bli veldig komplekst.



a) Objekttjener



b) Sidetjener

Figur 1-2 Klient/tjener arkitektur

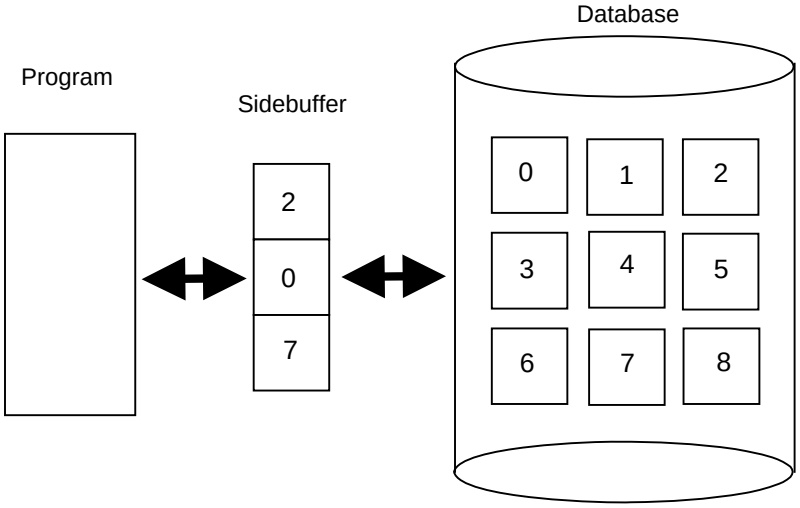
2. PROBLEMSPESIFIKASJON

Denne oppgaven går ut på å studere nye måter for bufferhåndtering. Vi vil i dette kapittelet derfor starte med å se hvordan dette tradisjonelt har blitt gjort med bruk av sidebuffer, og deretter beskrive et annet alternativ, nemlig objektbuffer. Mer informasjon kan finnes i [EffelsbergHaerder84] og [CopelandKhoshafianSmithValduriez86]. Problemstillingene med virtuelt minne i et operativsystem er langt på vei de samme. [SilberschatzGalvin94] er derfor ei relevant bok. Til slutt vil formålet med denne oppgaven konkretiseres.

2.1 Sidebuffer

Persistente objekter må overføres til et ikke-flyktig lagringsmedium for langtidslagring. I de fleste tilfeller er dette en magnetisk harddisk. De har i de siste årene befestet sin stilling som masselagringsmedium, med en dobling i kapasitet og halvering i pris hver 18. måned. Relaterte objekter organiseres i databaser (samme som fil) på disken. Hver database består av *k* sider, der hver side er på *m* byte (4096 eller 8192 byte er ganske vanlig), og rommer vanligvis mange objekter. Spesielle teknikker må brukes dersom objektene er lange, det vil si at de går over flere sider. All overføring av data mellom hovedminnet og disken skjer sidevis for økt ytelse. En typisk databaseapplikasjon trenger bare en liten del av databasen om gangen, men før et objekt kan brukes må et grunnleggende krav tilfredsstilles: Objektet må være i hovedminnet. Vi må derfor vite på hvilke side objektet er lagret, og overføre den sida til hovedminnet før objektene kan prosesseres.

Sidebufferet, en komponent innenfor et ODBMS, sin plass er vist i Figur 2-1. Hensikten er at sidebufferet skal inneholde de sidene som er mest brukt til enhver tid. Hvis ei side finnes i bufferet slipper man å overføre sida fra databasen og til hovedminnet, som er en svært kostbar operasjon, hvilket medfører en reduksjon i I/O. Fra figuren ser vi at databasen er på 9 sider, mens sidebufferet har 3 rammer (en ramme er like stor som ei side). Dersom programmet ikke benytter andre sider enn 2, 0 og 7, vil



Figur 2-1 Sidebufferet sin plass i et ODBMS

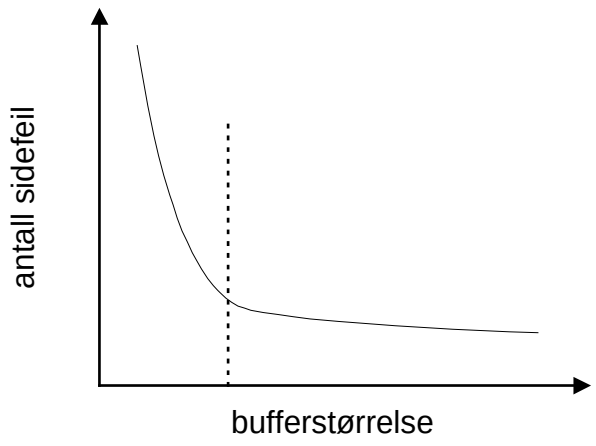
ytelsen bli svært høy. Etter at databasesida er overført til sidebufferet, kan objektene på sida fritt manipuleres.

Når det kommer en forespørsel til ei side som ikke ligger i bufferet, dette kalles en sidefeil, må ei som allerede ligger der kastes ut for å gjøre plass til den nye. Det er ikke nødvendig å skrive sida som må kastes ut til disken hvis ingen av objektene på sida er blitt endret i løpet av den tiden sida har ligget i bufferet. Dersom et objekt er endret, vil dets skittenflagg være satt. Valget av algoritme for utskifting av objekter er avgjørende for effektiviteten til bufferet. De sidene som nylig har vært brukt har som regel høy sannsynlighet for å bli brukt på ny i den nærmeste framtiden og motsatt. LRU (Least Recently Used) bygger på denne oppførselen i aksessmønstrene og er nok den mest populære av algoritmene. Den velger den sida som det er lengst siden har vært brukt til utkasting. Klokkealgoritmen, som er noe enklere å implementere (spesielt dersom hardware støtte er ønskelig), er en god tilnærming til LRU. Dessuten trenger den bare ett bit per side mens LRU må ha to pekere per side, men det er ikke så viktig siden størrelsen på ei side vanligvis er 4 kilobyte eller mer.

For at uskiftingsalgoritmene skal ha noen hensikt, i forhold til tilfeldig offervalg, må der være lokalitet i referansene til sider. Det vil si at sannsynligheten for å referere nylig refererte sider er høyere enn den gjennomsnittlige referansesannsynligheten. Arbeidssettet (eller varm-settet) til en databaseapplikasjon er de sidene som aktivt brukes i en spesifikk tidsperiode. Dersom bufferet er stort nok til å holde arbeidssettet, vil kjøringen være effektiv siden de fleste referansene blir funnet i bufferet. Ved lav lokalitet blir arbeidssettet stort, mens det blir lite ved høy lokalitet. Vi kan plotte antall sidefeil som en funksjon av bufferstørrelse, slik det er gjort i Figur 2-2, og vi vil da se at antall sidefeil øker raskt når bufferstørrelsen er mindre enn arbeidssettet.

2.2 Objektbuffer

Problemet til sidebufferet er når objektene som utgjør arbeidssettet, de varme objektene, ikke er sammenklynget. Maksimal sammenklynging har vi når alle sidene i arbeidssettet er fulle av varme objekter. Ved lav sammenklynging kan bufferminnet utnyttes mer effektivt ved å buffre objektene enkeltvis; vi får da et objektbuffer. Objektene leses allikevel fra disken sidevis, men de objektene som brukes kopieres over i objektbufferet. På den måten behøver vi ikke å buffre lite brukte objekter bare fordi de tilfeldigvis lå på den samme sida. Utnyttelsen av minnet blir derfor bedre slik at vi får redusert antall objektfeil (en objektfeil er når objektet ikke ligger i bufferet ved en aksess, og derfor må hentes fra disken). Målet er altså å ha flest mulig av de mest brukte objektene inne i hovedminnet. For å oppnå det må objektutskiftingsalgoritmen være god. På grunn av at overheaden per objekt med LRU blir 8 byte, er klokkealgoritmen bedre egnet til objektbufferet.



Figur 2-2 Sidefeil mot bufferstørrelse

Relativt lite har blitt publisert om denne alternative måten å håndtere et buffer på. Artikkelen [KemperKossmann92] omhandler dobbelbuffring, både et sidebuffer og et objektbuffer, der man forsøker gjennom intelligent samarbeid å buffre godt sammenklyngete sider og samtidig å trekke ut varme objekter fra ellers ubrukelige sider. Ulike varianter av: i) Når skal et objekt kopieres fra sidebufferet til objektbufferet?, og ii) Når skal et objekt kopieres fra objektbufferet til sidebufferet? ble studert. Forfatterne fant gjennom eksperimentering med OO7 benchmarken (jeg har dessverre ikke noe norsk ord for benchmark) og arkitektur b) i Figur 1-2 at dobbelbuffring var spesielt effektivt dersom bufferet var stort nok til å holde et stort antall av objektene i applikasjonens arbeidssett og dog for lite til å holde alle tilhørende sider. I dette tilfellet kunne dobbelbuffring redusere antall sidefeil med opptil 40 %.

2.2.1 Dynamisk minnehåndtering

Objektene som skal buffres er av variabel størrelse. Hovedoppgaven til objektbufferet blir effektiv minnehåndtering. Når blokker av variabel størrelse allokeres og deallokeres, medfører det at minnet blir fragmentert, som vist i Figur 2-3, i ledige og reserverte blokker. Denne situasjonen medfører fragmentering, som er minne som går til spille. Vi skiller mellom ekstern fragmentering som er de ledige blokkene, og intern fragmentering som er kontrollinformasjon som må legges inn i reserverte blokker.
I starten på hver blokk lagres størrelsen på blokka. De ledige blokkene lenkes sammen i ei ledigliste med en peker i hver ledig blokk. Når et objekt på f byte skal lastes inn, søker vi denne lista etter ei blokk på s byte, der $s \geq f$. Strategier for valg av blokk er:

- Ta den første blokka som er stor nok. Søkingen kan avsluttes straks vi finner ei stor nok blokk.
- Finn den blokka som er minst blant de som er store nok. Hele lista må søkes.
- Finn den største blokka blant de som er store nok. Hele lista må søkes.

Når blokka er funnet, deles den i ei reservert blokk på f byte og ei ledig på $s - f$ byte. Dersom vi ikke finner ei stor nok blokk til objektet, må vi kaste ut objekter til det oppstår ei. Valg av objekter som skal kastes ut må gjøres med en algoritme som tar hensyn til lokalitet i referansene, for eksempel LRU- eller klokkealgoritmen. Når ei blokk frigis, må vi sjekke om også naboblokkene er ledige, og da slå dem sammen for å gjøre fragmenteringen så lav som mulig.

Buddy systemet er en annen løsning på dynamisk minnehåndtering. Den sentrale ideen er at blokkene bare kommer i visse størrelser, for eksempel 1, 2, 4, 8, ..., eller 1, 2, 3, 5, ... Ulempen med buddy systemet er at blokkene har et begrenset antall størrelser, så plass går til spille ved å plassere et objekt i ei større blokk enn nødvendig. Fordelen er at allokering og deallokering av minne går raskt.

Mye forskning omkring dynamisk minnehåndtering er publisert. Knuth har tatt for seg dynamisk minnehåndtering i klassikeren [Knuth73] som inneholder flere gode tips. Ellers har vi boka [AhoHopcroftUllman83], samt artikkelen [WilsonJohnstoneNeelyBoles95] som er en omfattende gjennomgang og evaluering av konvensjonelle dynamiske allokatorer.

Figur 2-3 Ledige og reserverte blokker

2.3 Målsetting

Oppgaven går ut på å studere sammenhengen mellom aksessmønstre, minneutnyttelse og ytelse i et databasesystem som bruker objektbuffer. Objekter er av variabel størrelse, noe som gjør det mulig å utnytte minnet og disken mer effektivt ved lav sammenklynging av objektene. Hypotesen er at under visse betingelser vil det være bedre med et objektbasert buffer i steden for et sidebasert buffer. Men bruk av objektbuffer skaper også nye problemstillinger, som for eksempel minnefragmentering. For å prøve å gi svar på om hypotesen er korrekt, vil jeg i dette prosjektet utvikle et program som simulerer et objektbasert buffer. Simulatoren må også være i stand til å simulere et sidebuffer for å kunne sammenlikne ytelsen mellom buffertypene. Videre må det utvikles testapplikasjoner for å måle ytelsen. Ved så å analysere resultatene fra eksperimentene med buffersimulatoren, er målet er å komme fram til i hvilke situasjoner det er best å bruke sidebuffer, og i hvilke situasjoner det er best å bruke objektbuffer. Prosjektet vil også kunne gi verdifull erfaring omkring implementasjonsspørsmål for objektbufferet.

2.4 Krav til simulatoren og testapplikasjonen

Det er viktig at simulatoren er realistisk slik at ytelsesresultatene er relevante, men på grunn av tidsrammen til prosjektet vil forenklinger måtte innføres der det er mulig, og det kan naturlig nok påvirke realismen. Det vil ikke settes en stor mengde krav til programmene som skal utvikles, fordi det kan innskrenke friheten og kreativiteten. Noen generelle krav til arbeidet fra oppgavegiveren er:

- Det som vil ha høyest prioritet er å studere og implementere effektiv minnehåndtering med hensyn til tidsforbruk og fragmentering for objektbufferet. Valg av feil teknikker og metoder for å holde oversikt over ledig plass, finne ledig plass, frigi plass og utkasting av objektet kan få alvorlige konsekvenser for ytelsen. Et aktuelt spørsmål kan være om det er mulig å utnytte det faktum at objektene i en database vil være av et begrenset antall klasser. Hvor mye av bufferminnet som går vekk på grunn av fragmenteringen er et annet interessant spørsmål.
- Det er ikke ønskelig å innføre for mange parametre, siden det vil gjøre det vanskeligere å vite hva som faktisk påvirker ytelsen. For eksempel vil det være nok å simulere tid til lesing og skriving av sider i henhold til kostnadsfunksjoner.
- Man må kunne velge om simulatoren skal bruke sidebuffer eller objektbuffer internt, slik at samme testapplikasjon kan benyttes uavhengig av hvilken buffertype som benyttes.
- Ytelsen måles med en testapplikasjon, og den bør være en implementasjon av deler av en av de kjente benchmarkene, OO1 eller OO7.
- Betydningen av lokalitet i aksessene i forhold til tilfeldige aksesser må utforskes
- Betydningen av sammenklynging må studeres
- Betydningen av at aksessene også oppdaterer objektene i forhold til bare lesing
- Se på bruk av objektbuffer i skriveoptimaliserte databasesystem
- Kildekoden skal skrives i C++, ikke være kryptisk og med mye kommentarer. Videre arbeid vil da bli mye enklere.

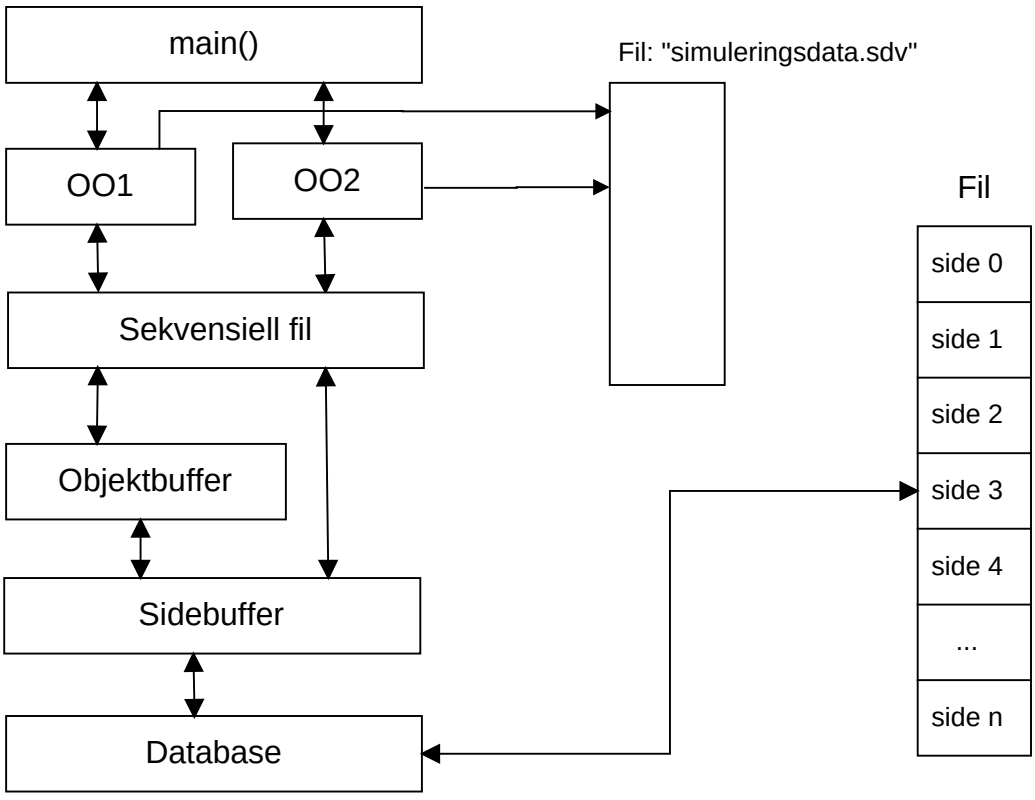
3. DESIGN OG IMPLEMENTASJON AV SIMULATOREN

Designet av simulatoren vil presenteres i dette kapittelet. Etter litt oversiktsinformasjon, blir klassene beskrevet bunn-opp. Alle klassefunksjoner presenteres, både private og offentlige, mens klassevariablene er utelatt. Dokumentasjonen, når det gjelder detaljeringsgrad, er forsøkt lagt et sted midt i mellom en brukermanual og kildekoden. For enkle funksjoner blir det bare forklart hva de gjør. Derimot vil det for de mer kompliserte funksjonene også beskrives hvordan de gjør det. Utvalgte algoritmer dokumenteres med pseudokode. I tillegg er det også laget noen figurer.

Kildekoden er skrevet i C++, og kan beskues i Vedlegg C. Ved noen få anledninger er funksjoner fra Windows API (Application Programmer Interface) brukt, samt enkelte funksjoner fra standardbibliotekene.

3.1 Oversikt

Simulatorens programvarestruktur er vist i Figur 3-1. Det har hele tiden vært en avveining mellom å gjøre ting så enkelt som mulig, samtidig som simulatoren må være realistisk slik at man kan trekke korrekte slutninger basert på analyse av ytelsesmålingene.



Figur 3-1 Oversikt

For å teste ytelsen, har jeg laget en implementasjon av en kjent og relativt enkel benchmark, OO1. Denne er spesielt rettet mot objektorienterte databaser. Jeg har også laget en enda enklere testapplikasjon for å få bedre kontroll over eksperimentelle betingelser som sammenklynging, oppdatering og lokalitet. Sammen vil disse testprogrammene gi et nyansert bilde av ytelsen. Benchmarkene blir gjennomgått i neste kapittel.

Et testprogram bruker operasjonene til den sekvensielle fila. En av disse er å sette inn et objekt. Siden en sekvensiell fil setter inn objektene på slutten av fila, sjekkes det først om der er plass på den siste sida. Ei ny side allokeres fra databasen dersom det ikke er plass. Objektet får nå en fysisk OID, som er sidenummer og offset. Det sendes en beskjed til sidebufferet om å laste inn aktuell side fra disken. Sidebufferet sjekker først om sida allerede er innlastet. Dersom det ikke er plass kastes den minst nylig brukte sida ut, og den nye lastes inn. Til selve overføringen av sider mellom hovedminnet og disken brukes databaseobjektet. Det skriver og leser sidene fra Windowsfila, samt holder orden på hvilke sider som er ledige og i bruk. Den sekvensielle fila kopierer deretter objektet fra sidebufferet og over i brukerminnet.

Hvis den sekvensielle fila benytter et objektbuffer og et objekt skal hentes fra databasen, vil fila sende en beskjed om å laste inn aktuelt objekt til objektbufferet, der objektet identifiseres med den fysiske OID'en. Objektbufferet sjekker om objektet allerede er innlastet. Dersom ikke, gjøres det plass i bufferet til det nye objektet. Da må kanskje noen av de andre objektene kastes ut først, der lite brukte objekter kastes ut før mye brukte objekter. Det underliggende sidebufferet kommanderes til å hente inn den sida som objektet ligger på. Deretter kopieres det over til objektbufferet.

3.2 Programvarekvalitet

I koden er funksjonene *debug(char flagg, char *format, ...)* og *assert(int uttrykk)* brukt en del for å gjøre det enklere å finne og fikse eventuelle feil, og derved også øke påliteligheten til programsystemet. Dersom *utrykk* er null vil *assert()* skrive en beskjed på formen:

```
Assertion failed: uttrykk, file: filnavn, line: nnn
```

Den kaller så *abort()* for å avslutte kjøringen. Funksjonen skal fange opp umulige betingelser i programmet. I noen tilfeller er den også, for enkelhets skyld, brukt i steden for feilhåndtering. Eksempel på bruk kan være:

```
assert(sidenr < database->AntallSider());
```

debug() skriver til skjermen på samme måte som *printf()* dersom *flagg* er satt, ellers skrives ingenting. Aktiv bruk av denne funksjonen gjør det mye enklere å spore feil. Et eksempel på bruk er:

```
debug('d', "Database: Leser side %d fra databasen %s\n",
                                                sidenr, navn);
```

3.3 Database

En database danner et abstrakt grensesnitt mot disken. Funksjonene til databaseklassen er vist i Figur 3-2. Overføring av data mellom hovedminnet og disken utføres i enheter av sider, der ei side er satt til å være 8 sektorer. Hver sektor er 512 byte på den disken som ble brukt, altså blir da sida 4096 byte. Sida kan gjerne være både mindre eller større, men ikke mindre enn en sektor og alltid et helt antall sektorer. Sidestørrelsen er en global variabel som kan endres.

```
class Database {
public:
     Database(char *databasenavn, int antallSiderIDatabase);
     ~Database();
     void LesSide(int sidenr, char *data);
     void SkrivSide(int sidenr, char *data);
     bool AllokerSide(int &sidenr);
     void DeallokerSide(int sidenr);
     int AntallSider();
     void SettAksesstid(unsigned int aksesstid);
     void HentStatistikk(int &antallSiderLest, int
          &antallSiderSkrevet, int &antallAllokerteSider);
     void NullstillStatistikk();
     void SkrivUtStatistikk();
     void VelgModus(DatabaseModus modus);
}
```

Figur 3-2 Database klassen

Tiden som det tar å lokalisere ei diskside (eller diskblokk), for så å overføre den mellom hovedminnet og disken, aksesstiden, er summen av:

- Posisjoneringstid: Dette er tiden det tar å mekanisk posisjonere lese/skrivehodet på korrekt spor, noe som tar mellom 10 til 60 millisekund.
- Rotasjonstid: Dette er tiden det tar før den ønskede sida har rotert inn i posisjon under lese/skrivehodet. Den er avhengig av diskens rotasjonshastighet. Ved 60 rotasjoner i sekundet, blir denne forsinkelsen 8.33 millisekund.
- Overføringstid: Dette er tiden som går til selve overføringen av sida. Den bestemmes av sidestørrelsen (antall byte som skal overføres), sporstørrelsen og rotasjonshastigheten.

Databasen bruker selv ei normal Windowsfil. Fila opprettes med *CreateFile()* (Win32 API) med flagget *FILE_FLAG_NO_BUFFERING* satt for da vil ikke operativsystemet bruke sitt eget buffer for filoperasjonene. Videre bruk av filer som er opprettet eller åpnet på denne måten krever at filaksesser må være til adresser (både i fila og i bufferet) og for et antall byte som begge er et integer multiplum av sektorstørrelsen. *LesSide()* leser ei side fra fila og inn i hovedminnet, mens *SkrivSide()* overfører sida fra hovedminnet til disken. Databasen kan kjøres i to modi, ekte eller simuler, og ønsket modus velges ved bruk av *VelgModus()*. I ekte modus blir filfunksjonene *SetFilePointer()* og *ReadFile()/WriteFile()* fra Win32 API brukt slik at dataene faktisk lagres på disken, mens i simuler modus blir ikke fila brukt. Aksesstiden i millisekund kan settes ved *SettAksesstid()*. Da vil lesing og skriving av ei side ta konstant tid, uavhengig av den faktiske tiden som brukes. Ytelsesmålingene blir da mindre upålitelige.

Antall sider som er lest og skrevet blir telt opp. Tellerverdiene hentes ved *HentStatistikk()*, og *NullstillStatistikk()* nullstiller dem. Databasen kan romme et fast antall sider som gis ved oppretting. En tabell brukes for å holde oversikt over hvilke sider som er i bruk. Tabellen vedlikeholdes av funksjonene *AllokerSide()* og *DeallokerSide()*.

3.4 Sidebuffer

Objektene ligger lagret på en magnetisk disk, fordi den gir rimelig permanent lagring av data. Logisk relaterte objekter samles i en database som er inndelt i sider av lik størrelse. Når et objekt skal leses, endres eller et nytt objekt skal settes inn, må den aktuelle databasesida først overføres fra disken til hovedminnet. Etter eventuelle endringer, skrives sida tilbake igjen til disken. For å øke ytelsen, ved å redusere

```
class Sidebuffer {
public:
      Sidebuffer(int antallRammerIBuffer);
      ~Sidebuffer();
      char *HentSide(int sidenr);
      void FrigiSide(int sidenr, int skitten = FALSK);
      void Flush();
      void NullstillStatistikk();
      void HentStatistikk(int &logiskeReferanser, int
                                          &fysiskeReferanser);
      void SkrivUtStatistikk();
      void ReduserAntallSegmenter();
private:
*     Ramme *FinnOffer();
*     void OppdaterLRUListe(Ramme *ramme);
*     void FjernFraLRUListe(Ramme *ramme);
}
* - Kun i versjon 1
```

Figur 3-3 Sidebuffer klassen

datatransporten mellom disken og hovedminnet, lagres de mest brukte sidene i hovedminnet. Sidebufferet danner grensesnittet mot databasen, og funksjonene er vist i Figur 3-3.

Konstruktøren allokerer et sammenhengende minneområde, stort nok til å romme et visst antall sider. Dette området inndeles logisk i rammer, der en ramme er like stor som ei side. En rammetabell brukes for å holde orden på hvilke sider som er innlastet og status til sidene. Strukturen til hvert element i den tabellen er vist i Figur 3-4. Variabelen *bufferadresse* er for så vidt unødvendig, fordi den kan beregnes utfra elementnummer, sidestørrelse og startadressen til bufferminnet.

HentSide(), se Figur 3-5, henter ei side fra databasen til bufferet. Dersom sida ikke allerede ligger i bufferet, og det også er fullt, må ei anna side kastes ut. Søking i bufferet for å undersøke om sida allerede ligger i hovedminnet, er implementert med en oversettingstabell; noe som gir konstant søketid. Den tabellen har like mange element som det er sider i databasen, og indekseres med sidenummeret. Hvert element er en rammepeker. Dersom side *i* er innlastet, vil tabellelement *i* peke på den rammen som sida er innlastet i, ellers er pekeren NULL (sida er ikke innlastet). Denne metoden er begrenset til svært små databaser. Vanligvis brukes en hashtabell.

Der er mange sideutskiftingsalgoritmer (se [EffelsbergHaerder84] eller [SilberschatzGalvin94] kapittel 9.5). Vi ønsker å bruke en algoritme med lite overhead og som samtidig har færrest mulig sidefeil. Jeg har implementert to av de mest populære utskiftingsalgoritmene. I versjon 0 brukes klokkealgoritmen til sideutskifting. Når ei side har blitt valgt som offer, sjekkes bruktflagget. Det indikerer om sida har vært referert siden sist sirkulering av offerpekeren. Bruktflagget resettes og offerpekeren flyttes til neste ramme dersom sida har vært brukt. Dette fortsetter helt til ei ubrukt side blir funnet. Sida skrives til disken dersom den er skitten. For å forhindre at sida blir kastet ut mens den benyttes av høyere lag, låses den. *FrigiSide()* låser opp sida, merker at den eventuelt er skitten og setter bruktflagget. *Flush()* går gjennom alle rammene og skriver skitne sider til disken. Hver forespørsel til ei side kalles en logisk referanse, mens hver aksess til ei side på disken kalles en fysisk referanse. Statistikkfunksjonene *HentStatisikkk()* og *NullstillStatisikk()* henholdsvis returnerer og nullstiller antall logiske og fysiske referanser. *SkrivUtStatisikk()* skriver de to statistikkvariablene til skjermen.

```
class Ramme {
public:
      int sidenr;
      bool skitten;
      bool laast;
*     bool brukt;
      char *bufferadresse;
**    Ramme *neste;
**    Ramme *forrige;
}
*   - Kun i versjon 0
**  - Kun i versjon 1
```

Figur 3-4 Ramme klassen

I versjon 1 er LRU brukt som sideutskiftingsalgoritme i bufferet. Den er regnet for å være bedre enn klokkealgoritmen og er meget utbredt. Hvert rammeelement må nå utvides med to pekere, *forrige* og *neste*. Vi lager altså ei dobbeltlenket liste av rammer. Rammene er sortert etter økende tid siden sist de ble brukt. Det siste elementet i lista har den største tiden siden sist den ble brukt, og derfor velges den som offer i *FinnOffer()* når ei side skal kastes ut. Dersom den sida skulle være låst, går man til den nest siste sida i LRU-lista, og så videre. Lista oppdateres hver gang en ramme frigis. Da kaller *OppdaterLRUListe()* først *FjernFraLRUListe()* som tar rammen ut av sin nåværende plass i lista, og deretter setter den rammen inn som det første elementet i LRU-lista. Ved å bruke et objekt, *hode,* av rammeklassen, der *neste* peker på det første elementet i lista og *forrige* peker på det siste elementet i lista (tips fra [Knuth73]), blir funksjonene for å fjerne et element fra lista og sette det inn først i lista svært enkle, med få spesialtilfeller:

- *Hode* initialiseres som:
  ```
  hode->neste = hode;
  hode->forrige = hode;
  ```
- Fjerne rammen fra lista:
  ```
  ramme->forrige->neste = ramme->neste;
  ramme->neste->forrige = ramme->forrige;
  ```
- Sette inn rammen som første element i LRU-lista:
  ```
  ramme->neste = hode->neste;
  hode->neste->forrige = ramme;
  hode->neste = ramme;
  ramme->forrige = hode;
  ```

ReduserAntallSegmenter() reduserer antall rammer i sidebufferet med *AntallSiderPerSegment,* som er satt til 27. Dette medfører at størrelsen på sidebufferet er dynamisk og kan endres under en databasesesjon. Funksjonen brukes av det adaptive objektbufferet (versjon 2), som ved kaldstart er lite med et stort underliggende sidebuffer. Etter hvert som objektbufferet fylles opp, så økes størrelsen samtidig som sidebufferet reduseres, noe som gir bedre kald ytelse. Funksjonen *ReduserAntallSegmenter()* kaster ut sidene som ligger i det siste segmentet, og hashtabellen og LRU-lista oppdateres. Dette er en primitiv metode, men er valgt for enkelthets skyld. En bedre metode, men som ikke er vesentlig for dette prosjektet, ville vært følgende:

- Ved oppretting av sidebufferet, allokeres et sammenhengende virtuelt adresserom med *VirtualAlloc()* (Win32 API) i steden for *malloc()*.
- LRU-lista benyttes til å finne de minst brukte sidene. De frigis så, side for side, med *VirtualFree()* når sidebufferet reduseres.

```
Char *Sidebuffer::HentSide(int sidenr)
{
   øk antall logiske referanser;
   if (sida er innlastet) {
      lås sida;
      return adressen til sida;
   }
   øk antall fysiske referanser;
   // sida er ikke i bufferet og vi må derfor finne en ramme
   while (offeret er låst eller har vært brukt ) {
      if (offeret er låst) {
         if (alle sidene er låst) {
            abort();
         }
         øk offerpekeren;
      }
      else if (offeret er brukt) {
         resett bruktflagget;
         øk offerpekeren;
      }
      else
         en ulåst ubrukt ramme er funnet, break;
   }
   if (sida er skitten) {
      database->Skriv sida til disken;
   }
   register i oversettingstabellen at sida er kastet ut;
   database->Les inn den nye sida;
   registrer i oversettingstabellen den nye sida som er innlastet;
   initialiser rammeflaggene;
   øk offerpekeren;
   return adressen til sida;
}
```

Figur 3-5 HentSide() algoritmen i versjon 0

3.5 Objektbuffer

3.5.1 Versjon 0

Objektbufferet buffrer objekter i steden for sider. Det forsøker å holde utvalgte objekter i hovedminnet slik at man ved gjentatt bruk skal slippe å lese dem inn fra disken igjen. Siden bare et begrenset antall objekter får plass i bufferet, er det svært viktig å holde på dem som brukes mest. I denne implementasjonen vil allikevel objektbufferet benytte seg av et underliggende sidebuffer. Dersom den totale bufferstørrelsen er 1200 rammer, vil 10 % (120 rammer) gå til sidebufferet, mens 1080 til objektbufferet (dette stemmer ikke for versjon 2). Etter at sida som objektet ligger på er lastet inn fra disken av sidebufferet, kopieres objektet over i objektbufferet. Klassen for objektbuffer er vist i Figur 3-6.

Fritt lager holdes som ei lenket liste av ledige blokker. Ei blokk er et sett med kontinuerlige minneceller. De første 8 byte av hver ledig blokk, blokkhodet, inneholder en peker til neste blokk pluss blokkstørrelsen, gitt ved:

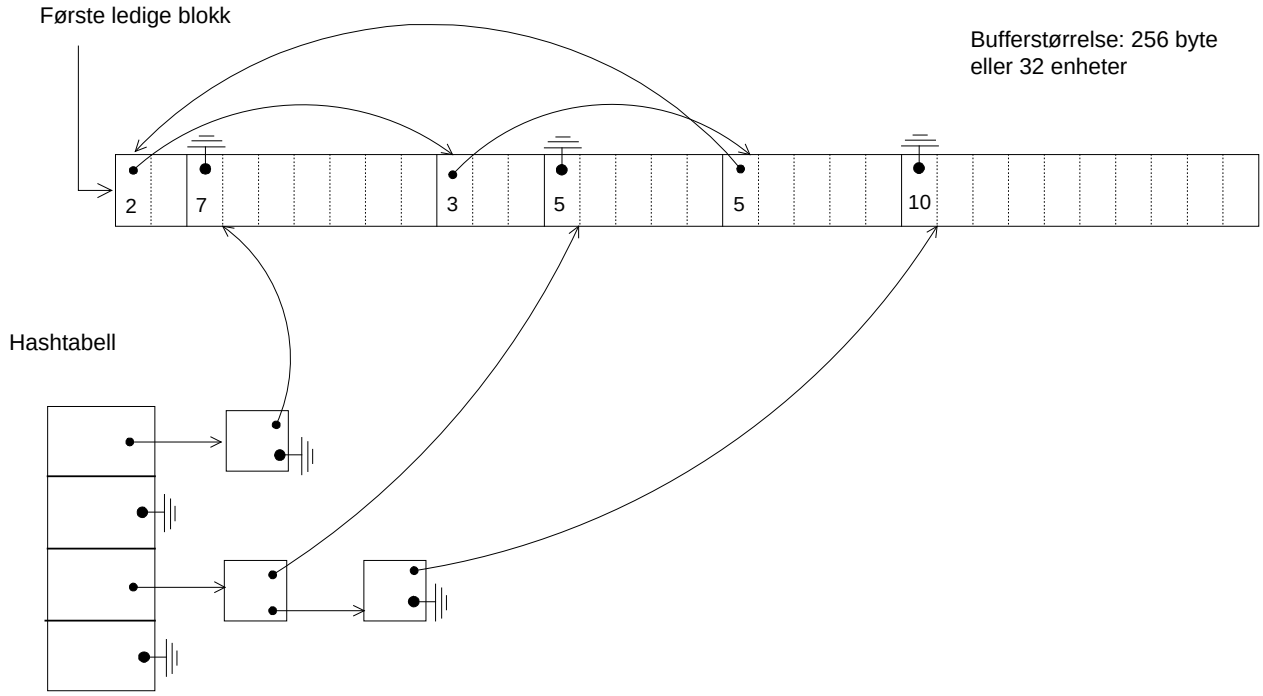
```
class Blokk {
     Blokk *neste;
     int antallEnheter;
}
```

```
class Objektbuffer {
public:
      Objektbuffer(int stoerrelse);
      ~Objektbuffer();
      char *HentObjekt(FysOID fysOID, int lengde);
      void FrigiObjekt(FysOID fysOID, bool skittent);
      void Flush();
      void NullstillStatistikk();
      Statistikk HentStatistikk();
      void SkrivUtMinnekart();
      void SkrivUtStatistikk();
*     void VelgModus(BufferModus modus);
private:
      char *AllokerBufferplass(int antallEnheter);
      Blokk *FrigiBufferplass(char *bufferadresse);
      bool KastUtObjekter(int antallEnheter);
      Objekt *FinnInnlastetObjekt(int sidenr, int offset);
      void RegistrerInnlastetObjekt(int sidenr, Objekt *objekt);
      void SlettRegistreringAvInnlasting(Objekt *objekt,
                                         int sidenr);
      void FinnNesteOffer();
      Statistikk BeregnMinneStatistikk();
}
* - Kun i versjon 2, se vedlegg A
```

Figur 3-6 Objektbuffer klassen

All intern minnehåndtering skjer i enheter, fordi det forenkler algoritmene samt på grunn av alignment restriksjoner. Alle blokkene blir da et multiplum av hodestørrelsen. Blokkhodet kan aldri være mindre enn et maskinord, det vil si vanligvis 32 bit. En enhet er størrelsen på den kontrollinformasjonen som ligger først i hver blokk. Blokkene holdes i rekkefølge av økende lageradresse, hvor den siste blokka peker på den første. Figur 3-7 viser dette for et buffer på 256 byte. Her er minnet inndelt i 3 reserverte blokker, det vil si at de er i bruk, blandet med tre ledige blokker som ikke er i bruk.

Et objekt på 42 byte vil bruke 1 enhet til kontrollinformasjonen og 6 enheter til selve objektet. I den siste enheten allokert til objektet vil 6 byte bli stående ubrukt (i



Figur 3-7 Objektbuffer oversikt

```
class Objekt {
public:
   bool skittent;
   bool laast;
   bool brukt;
   int stoerrelse;
   int offset;
   char *bufferadresse;
   Objekt *neste;
};
```

Figur 3-8 Objekt klassen

gjennomsnitt blir 50 % eller 4 byte av den siste enheten stående ubrukt). Totalt reserveres 7 enheter, hvor bare 42 av 56 byte (eller 75 %) av det reserverte minnet faktisk benyttes til selve objektet på grunn av intern fragmentering. I snitt vil det på grunn av intern fragmentering gå vekk 12 byte i overhead per objekt - i tillegg til alle de ledige blokkene som utgjør den eksterne fragmenteringen. Ved en gjennomsnittlig objektstørrelse på 175 byte (som er den objektstørrelsen som er benyttet i OO2 benchmarken), går da 6,7 % av minnet til spille på grunn av intern fragmentering.

For raskt å finne igjen objektene som ligger i bufferet og hvilken status de har, må vi ha en aksesstruktur. Den er vist i Figur 3-7. Hashtabellen, som har like mange elementer som det er sider i databasen, er en katalog over hvilke objekter som ligger i bufferet og indekseres med sidenummeret. Elementtypen er en peker til det første objektet i ei objektliste. Alle objektene i objektliste *k* kommer fra databaseside *k*. Typen til disse objektene er vist i Figur 3-8, og de lagrer en del kjøretidsinformasjon om de objektene som ligger i bufferet. Dersom ingen objekter fra ei side er innlastet, vil tilsvarende peker i hashtabellen være NULL. Hvis vi antar at den gjennomsnittlige objektstørrelsen er 175 byte og sidestørrelsen er 4096 byte, vil hver objektliste i snitt ha 23 element. Tiden det tar for å avgjøre om et objekt er innlastet vil derfor være O(1).

Tre funksjoner manipulerer hashtabellen. *RegistrerInnlastetObjekt()* registrerer at et objekt er innlastet i bufferet ved å lenke det inn i korrekt objektliste. *FinnInnlastetObjekt()* søker i hashtabellen etter aktuelt objekt og returnerer det. Dersom det ikke finnes, returneres NULL. *SlettRegistreringAvInnlasting()* sletter objektet fra lista over innlastete objekter fra samme side.

Figur 3-9 viser tilstanden til minnet etter at OO2 benchmarken er blitt kjørt 20 ganger. Minnekartet er skrevet ut med *SkrivUtMinnekart()*. Den funksjonen går

```
$....|xxxxxxxxxxxxxx|xxxxxxxxxxxxxxxxxxxxxxxxxx|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxx$.........|xxxxxxxxxxxxxxxxxx|xxxxxxxxxx|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
x|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx$.......|xxxxxxxxxxxxx|xxxxxxxxxxxxxxxxxx
xxxxxxxxxx|xxxxxxxxxxxxxxxxxxxxx|xxxxxxxxxxxxxxxxxxxxxxxxxxxxx$...|xxx
xxxxxxxxxxxxxxxxxxxxxxxx|xxxxxxxxxxxxxxx$..................|xxxxxxxxxxx
xxxxxxxxxxxxx|xxxxxxxxxxxxxxxxx|xxxxxxxxxxxxxxxxxxx$.........|xxxxxxxxx
xxxxxxxxxxxxxxxxxxxxx|xxxxxxxxxxxxxxxxxxxxx$.......|xxxxxxxxxxxxxxx$..
.................|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx$.....|xxxxxxxxxxxxxxx
xxxxxxxxxxx|xxxxxxxxxxxxxxxxx$.|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx$|xx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxx|xxxxxxxxxxxxxxxxxxxxxxxxxxxx|xxxxxxx
xxxxxxxxxxxx|xxxxxxxxxxxxx$.............|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
x|xxxxxxxxxxxxxxxxxxxxxxxxxxxxx$.........................|xxxxxxxxxxxxx
xxxx$.............|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx|xxxxxx|xxxxxxxxxxxxx
xxxxxxxxxxxxxxxxx|xxxxxxxxxxxxxxxxxxxxxxxxxxxxx$....|xxxxxx$.....
.............|xxxxxxxxxxxx|xxxxxxxxx|xxxxxxxxxx$....|xxxxxxxxxxxxxxxx
xxxxxxxx$.........|xxxxxx|xxxxxxxxx$......|xxxxxxxxxxxxxxxxxxxx$...|xxxx
xxxxxxxxxxxx|xxxxxxxxxxxxxxxxx|xxxxxxxxxxxxxxxxxxxxxxxxxxxx|xxxxxxx
xxxxx$.............|xxxxxxxxxxxxxxxxxxxxxxxxxxx|xxxxxxxxxxxxxxxxx
xxxxx|xxxxxxxxxxxxxxxxxxxxxxxx$..........|xxxxxxxxx|xxxxxxxxxxxxxxxxx
xxx$.............|xxxxxxxxxxxx|xxxxxxxxxxxxxxxxxxxxxxxxxxxx|xxxxxxx
xxxxxxxxxxxxxxxxxxxx$..|xxxxxxxxxxxxxxxxxxxxxxxxxx|xxxxxxxxxxxxxxx
xxxxxxxxx$.........|xxxxxxxxxxxxxxxxxxxxxxxx|xxxxxx|xxxxxxxxxxxxxxx
xxxxxxxxxxxxx|xxxxxxxx$.......|xxxxxxxxxx|xxxxxxxxxxxxxxxxxxxxxxxx
xx$..|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx|xxxxxxxxxxxxxxxxx$...........|
xxxxxxxxxxxxxxxxxxxxxxx$.....|xxxxxxxxxxxxxxxxxx|xxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxx|xxxxxxxxxxxxxxxxxx
```

Figur 3-9 Et minnekart

```
char *Objektbuffer::HentObjekt(FysOID fysOID, int stoerrelse)
{
   øk antall forespørsler;
   sjekk om objektet ligger i bufferet;
   if (objektet er innlastet) {
      return objekt->bufferadresse;
   }
   beregn hvor mange minneenheter som trengs;
   do {
     alloker bufferplass;
     if (bufferet er fullt) {
        kast ut objekter inntil det blir nok ledig plass;
     }
   } while (bufferplass ikke funnet);
   initialiser objektet;
   sidebuffer->Hent inn sida som objektet ligger på;
   kopier objektet fra sidebufferet til objektbufferet;
   sidebuffer->Frigi sida;
   oppdater hashtabellen med det nye objektet;
   return objekt->bufferadresse;
}
```

Figur 3-11 HentObjekt() algoritmen

sekvensielt gjennom alle blokkene i bufferet og skriver dem til skjermen. Ei reservert blokk på 40 byte skrives som "|XXXXX", der hver karakter tilsvarer en enhet, mens ei ledig blokk på 48 byte skrives som "§.....". Blokkene skrives etter stigende adresse med 80 enheter per linje. For å sjekke om ei blokk er reservert, brukes pekeren *neste* som i så tilfelle vil være NULL. Bufferet i dette tilfellet er 16384 byte stort, hvilket tilsvarer 2048 enheter. Av dem er 1749 reserverte (der også blokkhodet i reserverte blokker telles med) og 299 ledige. Den effektive minneutnyttelsen er på 80,44 % med 68 objekter og 29 ledige blokker. Den noe dårlige minneutnyttelsen i dette eksemplet skyldes at bufferet var så lite.

HentObjekt() henter et objekt fra databasen og inn i bufferet. Algoritmen er vist i Figur 3-11. Først undersøkes det om objektet allerede ligger i bufferet. Dersom ikke objektet ligger inne, brukes *AllokerBufferplass()* for å lage plass i bufferet. Objektet hentes så fra disken, og hashtabellen oppdateres. Objektet låses og en peker til objektet returneres. Ofte vil det være slik at ingen av de ledige blokkene er store nok til å romme det nye objektet. Da må et eller flere av residente objekter kastes ut, helt til ei stor nok blokk blir ledig. Funksjonen som gjør dette, *KastUtObjekter()*, er skissert med pseudokode i Figur 3-10. Offeret velges med klokkealgoritmen i *FinnNesteOffer()*. Dersom et objekt har bruktflagget satt, nullstilles det og det neste objektet undersøkes.

```
bool Objektbuffer::KastUtObjekter(int antallEnheter)
{
   while (stor nok blokk ikke er blitt frigitt) {
      finn neste offer;
      if (offeret er skittent) {
         sidebuffer->Hent sida;
         kopier objektet til sidebufferet;
         sidebuffer->Frigi sida;
      }
      frigi bufferplassen til offeret;
      oppdater hashtabellen;
      if (blokka som er frigitt er stor nok) {
         return SANN;
      }
   }
}
```

Figur 3-10 KastUtObjekter() algoritmen

```
char *Objektbuffer::AllokerBufferplass(int antallEnheter)
{
   start med den første ledige blokka;
   while (stor nok blokk ikke er funnet) {
      if (blokka er stor nok) {
         if (blokka er av eksakt størrelse){
            fjern blokka fra lediglista;
         } else {
            splitt blokka og alloker halen;
            oppdater den nye ledige blokka med nødvendig
            kontrollinformasjon;
         }
         return adressen til blokka;
      }
      if (hele lediglista er gjennomgått) {
         break;
      } else {
         gå til neste blokk;
      }
   }
   return NULL;
}
```

Figur 3-12 AllokerBufferplass() algoritmen

Skitne objekter må skrives til disken, noe som medfører at objektets hjemmeside må leses inn fra disken. *FrigiObjekt()* låser opp objektet, setter bruktflagget og oppdaterer skittentflagget.

Algoritmene for dynamisk minnehåndtering er vist i Figur 3-12 og Figur 3-13. *AllokerBufferplass()* finner ei ledig blokk i bufferet. Lediglista granskes helt til ei stor nok blokk finnes. Denne algoritmen kalles "første høvelige", i kontrast til "beste høvelige" som ser etter den minste blokka som tilfredsstiller forespørselen. Dersom blokka passer akkurat, fjernes den fra lista og returneres. Ellers splittes den, og den tilbakeværende delen lenkes inn i lista over ledige blokker. NULL returneres dersom ingen stor nok blokk finnes. Kjøretiden for ei ledigliste med *n* blokker er O(n) hvis funksjonen ikke lykkes i å finne ei blokk og O(n/2) ellers.

FrigiBufferplass() frigir ei reservert blokk. Lista over ledige blokker søkes for å finne den riktige plassen til å sette inn den frigitte blokka på. I snitt, for *n* ledige blokker, blir kjøretiden O(n/2). Dersom blokka som frigis er sammenstøtende med ei anna fri blokk (på ei av sidene), slås de sammen til ei større blokk for å minske fragmenteringen. Siden lista over ledige blokker er ordnet etter økende adresse, er det lett å bestemme om ei eller begge naboblokkene også er ledige. Den frigitte blokka returneres. Algoritmen er vist i Figur 3-13.

Flush() skriver alle skitne objekter til disken. Dersom et objekt er skittent, hentes eventuelt aktuell side inn i sidebufferet, og objektet kopieres over der. Etter at alle objektene er undersøkt, flushes sidebufferet. Statistikkinformasjonen som returneres fra

```
Blokk *Objektbuffer::FrigiBufferplass(char *bufferadresse)
{
   søk gjennom lediglista for å finne ut hvor den nye
   blokka skal settes inn;
   if (blokka til høyre er ledig) {
      slå sammen frigitt blokk med blokk til høyre;
   }
   if (blokka til venstre er ledig) {
      slå sammen frigitt blokk med blokk til venstre;
   }
   lenk den nye blokka inn i lediglista;
   return den frigitte blokka, med eventuelle sammenslåinger;
}
```

Figur 3-13 FrigiBufferplass() algoritmen

```
class Statistikk {
public:
    int antallForespoersler;
    int antallMiss;
    double minneutnyttelse;
    int antallLedigeBlokker;
    int antallObjekter;
    int antallEnheterReserverte;
    int antallEnheterLedige;
}
```

Figur 3-14 Statistikk klassen

HentStatistikk() er vist i Figur 3-14. *BeregnMinneStatistikk()* beregner minneutnyttelse, antall ledige og reserverte enheter, antall ledige blokker og antall objekter. Alle blokkene gjennomgås og reservert/ledig plass summeres. Dersom *neste* pekeren til ei blokk er NULL, vet vi at blokka er reservert. Minneutnyttelsen beregnes som summen av størrelsen til alle objektene dividert på det totale bufferminnet. Variabelen *sumStoerrelse* oppdateres ettersom objektene lastes inn og kastes ut. Den summerer altså, i byte, hvor mye av minnet som brukes til selve objektene. Objektbufferet har også en funksjon for å skrive statistikken pent formatert til skjermen, *SkrivUtStatistikk()*.

3.5.2 **Versjon 1**

Det viste seg at minnehåndteringen i versjon 0 ble ineffektiv når bufferet kom opp i en viss størrelse. For en gjennomsnittlig objektstørrelse på 175 byte og et objektbuffer på 3.9 MB, kom antall ledige blokker opp i 7000. Dersom dette kombineres med lav lokalitet, som medfører stor gjennomstrømning av objekter, ble tiden som gikk til søking i lediglista uakseptabel høy. Man må huske på at lediglista må søkes dersom et objekt ikke ligger i bufferet for å se om det finnes ei stor nok ledig blokk. I tillegg må lediglista søkes for hvert objekt som kastes ut for å finne korrekt plass for den nye ledige blokka som da blir frigjort.

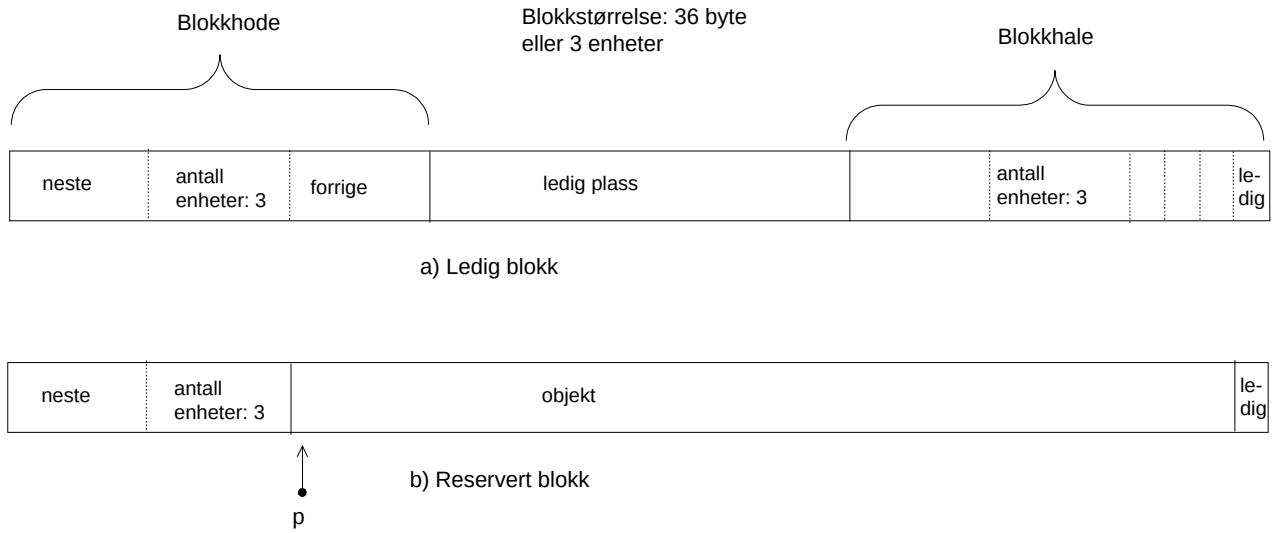
I versjon 1 av objektbufferet er algoritmene for dynamisk minnehåndtering sterkt forbedret. To vesentlige endringer er innført:

- Det er lagt inn ekstra kontrollinformasjon i halen på hver blokk. Ideen kommer fra [Knuth73].
- Lediglista er splittet i flere lediglister basert på blokkstørrelsen.

Lista over ledige blokker gjøres dobbelt lenket, fordi da kan ei blokk fjernes på konstant tid. Enhetsstørrelsen øker derfor til 12 byte og 6 byte vil i snitt være ubrukt for den siste enheten i reserverte blokker. Den nye blokklayouten er vist i Figur 3-16. Hodet utvides med en *forrige* peker, og dessuten legges det også inn informasjon i halen. Den informasjonen har til hensikt å eliminere all søking når lagerplass frigis. Når ei blokk skal frigis, må vi sjekke om naboblokkene også er ledige, og da slå dem sammen til ei større blokk. Den høyre naboblokka vet vi hvor er ved å bruke *antallEnheter* variabelen i hodet.

I versjon 0 fant vi den venstre naboblokka ved først å søke i den sorterte lediglista etter korrekt plass som den frigitte blokka skulle inn på, for deretter å sjekke om den ledige blokka også var naboblokka. I versjon 1 vil den siste byten i hver blokk brukes til dette. Lediglista trenger ikke lenger være sortert. Dersom blokka til venstre er ledig, vil *ledig* være 1. Da vil hodet på blokka kunne nås ved å bruke *antallEnheter* i halen. Den venstre blokka lenkes deretter ut av lediglista.

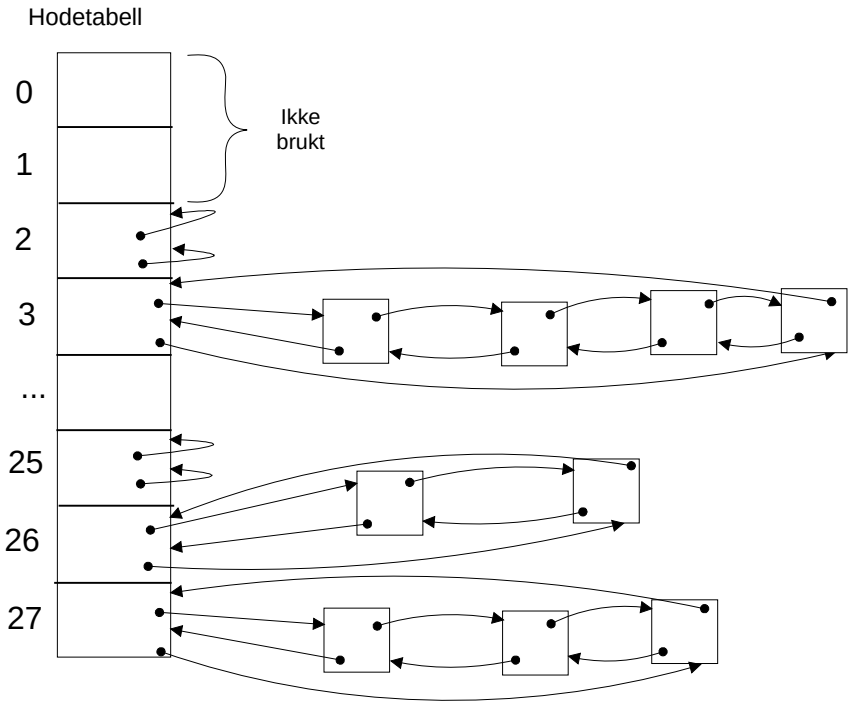
For å redusere overheaden, vil *forrige* i hodet samt hele halen, bortsett fra den siste byten, bli reservert. Ei blokk på 3 enheter vil kunne romme et objekt på 27 byte. Overheaden på grunn av den interne fragmenteringen blir derfor 8 + 6 + (1) = 14 byte per objekt. Den byten som går til *ledig* variabelen trenger ikke å tas med, fordi mye av



Figur 3-16 Blokklayout

den siste enheten vil allikevel stå ubrukt. Et problem nå er at den minste ledige blokkstørrelsen er 2 enheter; en til halen og en til hodet. Problemet er løst ved å sette *ledig* til 2 når blokkstørrelsen bare er en enhet, og ikke lenke disse små blokkene inn i lediglista. Med tiden vil blokkene (på en enhet) bli sammenslått med andre større blokker.

Eliminering av søking når lagerplass skal reserveres er gjort ved å dele lediglista i flere lediglister. Dette er vist i Figur 3-15. Hodetabellen har 28 elementer. Hvert element er et *hode* med en *neste* og *forrige* peker, som peker henholdsvis til første og siste element i aktuell ledigliste. Element *k* peker til ei liste der alle blokkene er av størrelse *k* enheter. Det siste elementet i tabellen, nummer 27, er for blokker som er større en 26 enheter. Størrelsen på tabellen er en global konstant som kan varieres i takt med antall objektstørrelser som skal lagres i databasen.



Figur 3-15 Hodetabell

Når det skal allokeres plass til et objekt på *m* enheter, er det bare å velge den første blokka i lediglista gitt ved element *m.* Dersom det ikke er noen blokker på akkurat *m* enheter, undersøkes det neste elementet helt til man kommer til den siste lediglista. Blokkene i den siste lista er av variabel størrelse, men minst 27 enheter. I de benchmarkene som er brukt er den største objektstørrelsen 300 byte og kan rommes i ei blokk på 26 enheter (inkludert hodet). Dette vil derfor gi konstant søketid.

Dersom objektbufferet vet hvilke typer objekter som skal lagres i bufferet, kan antall lediglister tilpasses dette. For eksempel trenger vi fire lediglister hvis vi har tre typer objekter på størrelse s_0, s_1 og s_2:

1. blokkstørrelse $< s_0$
2. $s_0 \leq$ blokkstørrelse $< s_1$
3. $s_1 \leq$ blokkstørrelse $< s_2$
4. blokkstørrelse $\geq s_2$

3.5.3 Versjon 2

Tester viste også at objektbufferet hadde svært dårlig kald ytelse. For å rette på det, ble størrelsen på objektbufferet gjort adaptiv. Et segment defineres til å være 27 sider, men kan endres. Når et objektbuffer på 50 segment (1350 rammer) opprettes, vil 49 segment brukes til det underliggende sidebufferet, mens bare ett segment til objektbufferet. Etter som tiden går vil objektbufferet fylles, objekt for objekt. Når objektbufferet er fullt, reduseres sidebufferet med ett segment og objektbufferet økes med ett segment. Slik fortsetter det helt til sidebufferet kun er ett segment. Denne endringen medfører at den kalde ytelsen blir omtrent som for sidebufferet.

3.6 Sekvensiell fil

I ei sekvensiell fil lagres objektene i den rekkefølgen som de settes inn i. Objektene kan være av variabel størrelse, og de settes inn på slutten av fila. Klassen (se Figur 3-17) tilbyr en forenklet versjon av denne typen filer, men tilstrekkelig til å kunne kjøre enkle testprogrammer. Der er ikke støtte for sletting av objekter i fila, og henting kan kun gjøres basert på den fysiske OID'en. Poenget med denne klassen er å gjøre det enklere å skrive testprogram.

SettInnObjekt() setter inn et objekt med en gitt lengde i fila. Først sjekkes det om der er plass til objektet på den siste sida. Dersom der er plass, hentes sida inn fra databasen, ellers allokeres ei ny side. Objektet kopieres over til sida og får nå en fysisk OID. Den er sidenummer pluss offset innenfor sida. *HentObjekt()* henter aktuelt objekt fra databasen basert på fysisk OID. Algoritmen er vist i Figur 3-18. Her ser vi også at fila enten bruker objektbufferet eller sidebufferet. Hvilken buffertype som skal benyttes velges med *VelgBuffertype()*. Et objekt kan også oppdateres med *OppdaterObjekt()*. Testprogrammene kan nå lages helt like uansett hvilken buffertype som skal brukes.

```
Class SekvensiellFil {
public:
      SekvensiellFil(bool laasSisteSide = FALSK);
      ~SekvensiellFil();
      FysOID SettInnObjekt(char *data, int lengde);
      void HentObjekt(char *data, int lengde, FysOID fysOID);
      void OppdaterObjekt(char *data, int lengde, FysOID fysOID);
      void VelgBuffertype(char buffertype);
}
```

Figur 3-17 SekvensiellFil klassen

```
void SekvensiellFil::HentObjekt(char *data, int lengde, FysOID
                                              fysOID, LogOID logOID)
{
   if (sidebufferet skal benyttes) {
      sidebuffer->Hent inn sida som objektet ligger på;
      kopier objektet fra sidebufferet til brukerminnet;
      sidebuffer->Frigi sida;
   } else {  // objektbufferet skal benyttes
      objektbuffer->Hent inn objektet;
      kopier objektet fra objektbufferet til brukerminnet;
      objektbuffer->Frigi objektet;
   }
}
```

Figur 3-18 HentObjekt() algoritmen

4. BENCHMARKENE

Sammenlikning av ytelse gjennom teoretisk analyse er nærmest umulig, bortsett fra i de helt enkleste tilfellene. Flere benchmarker har blitt foreslått, men jeg har valgt OO1. Denne benchmarken, sammen med min egen OO2, blir beskrevet i dette kapittelet. Selve implementasjonen er vektlagt mest. En detaljert spesifikasjon av OO1 kan finnes i [CattellSkeen92]. Formålet med disse benchmarkene i dette prosjektet er ikke å sammenlikne ytelsen til simulatoren med andre databasesystem, men mellom bruk av objektbuffer eller sidebuffer. OO7 benchmarken ble ikke implementert fordi den ble for kompleks. For en full beskrivelse av OO7 henviser jeg leseren til [CareyDeWittNaughton94].

4.1 Objektkatalogen

Hvert element i objektkatalogen inneholder de variablene vist i Figur 4-2. Katalogen brukes for å finne ut hvor et objekt, identifisert med en logisk OID, ligger lagret på disken. Klassen vist i Figur 4-1 viser de operasjonen som er tilgjengelige for hashtabellen som implementerer katalogen. Hashtabellen har like mange elementer som det er sider i databasen. Hashfunksjonen er H(logiskOID mod antallSiderIDatabasen). De elementene som hasher til samme plass lenkes sammen i ei liste. *SettInnElement()* setter inn et nytt element i hashtabellen, *FinnElement()* søker etter et element med en bestemt logisk OID og returnerer den fysiske OID'en, mens *OppdaterElement()* oppdaterer den fysiske OID'en til et objekt.

```
class Hashtabell {
public:
      Hashtabell();
      ~Hashtabell();
      void SettInnElement(LogOID logOID, FysOID fysOID, int
                                                  stoerrelse);
      bool FinnElement(LogOID logOID, FysOID &fysOID, int
                                                 &stoerrelse);
      void OppdaterElement(LogOID logOID, FysOID fysOID);
}
```

Figur 4-1 Hashtabell klassen

```
class Element {
public:
        LogOID logOID;
        FysOID fysOID;
        int stoerrelse;
        Element *neste;
};
```

Figur 4-2 Element klassen

4.2 OO1

Objekt Operasjoner versjon 1 er en enkel benchmark rettet mot ytelsesmåling av objektorienterte databasesystem, men den kan også benyttes mot relasjons-, nettverks-, eller hierarkiske databasesystemer. Målingene omfatter: i) Sette inn objekter, ii) hente objekter og iii) følge koblinger mellom objekter.

Benchmark databasen består av to typer objekter (se Figur 4-3) , deler og koblinger. Implementasjonen av objektbufferet er ikke effektiv med hensyn til overheaden per objekt, som er 14 byte. Det er ikke særlig vanskelig å få den ned i 6-7 byte ved å bruke 4 byte til hodet, en byte til halen (når blokka er reservert) og en allokeringsenhet på 4 byte. For å kompensere for dette, er størrelsen på objektene i benchmark databasen doblet med en *foo* tabell som ikke er en del av OO1 spesifikasjonen. Størrelsen på en del er 152 byte og på en kobling 96 byte. Overheaden på grunn av kontrollinformasjonen (intern fragmentering) blir nå 14 / (152 + 14) * 100 = 8,4 % for delene og 14 / (96 + 14) * 100 = 12,7 % for koblingene, altså ganske vesentlig. I tillegg kommer da også alle de ledige blokkene.

Databasen initialiseres med 20 000 deler og 60 000 koblinger med eksakt tre koblinger fra hver del til andre deler. Vi får samme struktur som en rettet graf hvis en node representerer en del og en rettet kant representerer koblingen. Ei side på 4096 byte rommer 26 deler eller 42 koblinger. Delene er lagret fortløpende på de 769 første sidene, av en database på totalt 2198 sider (8,6 MB), og koblingene ligger på de 1429 siste sidene.

Vi har en 1:N relasjon mellom en del og de koblingene som går fra delen, og en 1:N relasjon mellom en del og de koblingene som kommer til delen. Tre koblinger vil alltid gå fra delen, mens det vil variere hvor mange koblinger som kommer til delen. Relasjonene er implementert som en lenket liste. Det er samme teknikk som brukes i nettverksdatabaser. Del *m* peker (fysisk OID) på den første koblingen som går fra delen. Den koblingen vil peke på den neste koblingen som går fra del *m,* og så videre. Pekeren til den siste koblingen settes til (-1, -1) for å markere slutten på kjeden. Figur 4-4 viser hvordan dette er representert fysisk og logisk.

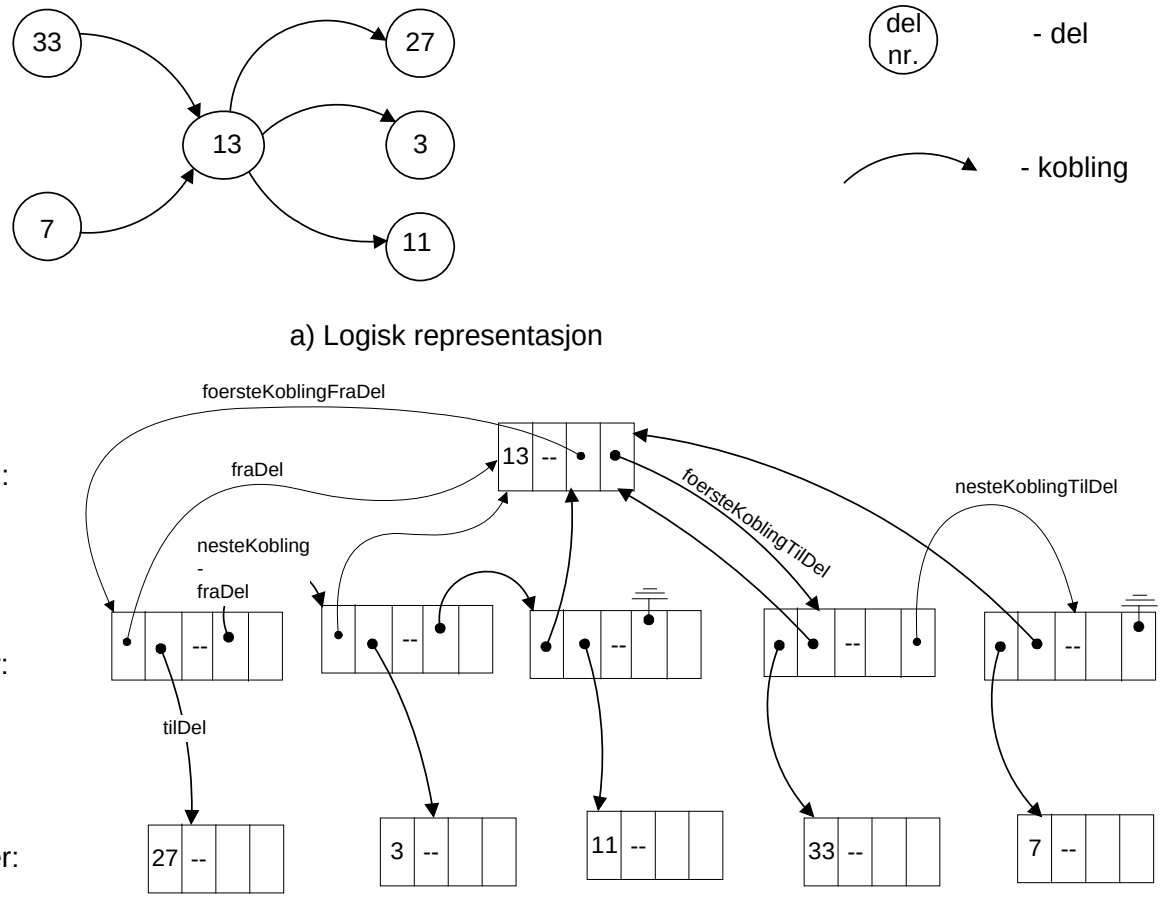
Koblingene mellom delene velges slik at de gir lokalitet i referansene. 90 prosent av koblingene fra en del går til 1 prosent av de "nærmeste" delene, og de resterende koblingene går til tilfeldig valgte deler. Nærhet er definert som nærhet i logisk OID. For del 300, vil delene med ID'er i området 200 - 400 være innenfor denne 1 prosent grensen.

Å definere lokalitet på denne måten blir en ulempe for objektbufferet, fordi det gir sidelokalitet. Ei side som nylig har blitt referert har større enn gjennomsnittlig sannsynlighet for å bli referert igjen på grunn av at en stor andel av koblingene går til nærliggende objekter. Dette gjelder også for objekter, men i en mindre grad. Benchmarken er derfor også testet med en annen type lokalitet der en kobling har 80 prosent sannsynlighet for å gå til et varmt objekt og 20 prosent sannsynlighet for å gå til et kaldt objekt. Databasen deles altså i to, en kald og en varm del. De 4000 første delene settes til å være varme (20 prosent av databasen), mens resten er kalde. God ytelse vil nå oppnås dersom bufferet klarer å holde de varme objektene, som brukes

```
Class Del {
public:
   LogOID logOID;
   char type[10];
   int x,y;
   struct tm bygget;
   char foo[76];
   FysOID foersteKoblingFraDel;
   FysOID foersteKoblingTilDel;
}
```

```
class Kobling {
public:
     FysOID fraDel;
     FysOID tilDel;
     char type[10];
     char foo[48];
     int lengde;
     FysOID nesteKoblingFraDel;
     FysOID nesteKoblingTilDel;
}
```

Figur 4-3 Benchmark database



Figur 4-4 Representasjon av relasjoner

mye, i hovedminnet. Koden for lokalitet er gjengitt i Figur 4-5.

```
If ((rand() % 10) > 0) {
   tilDel = fraDel + (rand() % (antallDeler/100))
                                    - antallDeler / 200;
   if (tilDel < (antallDeler / 200) ) {
      tilDel += antallDeler / 200;
   }
   if (tilDel > (antallDeler - antallDeler / 200) ) {
      tilDel -= antallDeler / 200;
   }
} else {
   tilDel = rand() % antallDeler;
}
```

a) sidelokalitet

```
if ((rand() % 10) > 1) {
   tilDel = (rand() % 4000);
} else {
   tilDel = (rand() % (antallDeler - 4000)) + 4000;
}
```

b) objektlokalitet

Figur 4-5 Definisjon av lokalitet

```
Void Objektoperasjoner1();
void Initialiser(int antallNyeDeler);
void SettInn(int antallNyeDeler);
void SettInnKoblinger(int antallNyeDeler);
void SlaaOpp(int antallOppslag);
int GjennomgaaFramover(FysOID delFysOID, int dybde);
int GjennomgaaBakover(FysOID delFysOID, int dybde);
void SkrivUtDel(Del del, char debugflagg);
LogOID SkrivUtKobling(Kobling kobling, char debugflagg);
void Sammenkoble(FysOID fraDelFysOID, FysOID tilDelFysOID, FysOID
                                                    koblingFysOID);
```

Figur 4-6 OO1 funksjoner

Funksjonene som implementerer OO1 benchmarken er vist i Figur 4-6. *SettInn()* setter inn deler i databasen. Hver del initialiseres og gis en logisk OID. Delen settes inn i den sekvensielle datafila, hvor den får en fysisk OID, og et nytt element legges inn i objektkatalogen slik at vi kan finne igjen delen. Når alle delene er satt inn, kalles *SettInnKoblinger()*. Den funksjonen går gjennom alle de nye delene og oppretter koblinger mellom dem. Hvilken del en kobling skal gå til bestemmes ved en lokalitetsalgoritme som forklart ovenfor. Objektkatalogen brukes for å finne fysisk OID til variablene *fraDel* og *tilDel* for koblingen. Deretter kalles *Sammenkoble()* for hver kobling. Den oppdaterer pekerlistene og pseudokode for funksjonen er vist i Figur 4-7 (se også Figur 4-4).

Benchmarken er sammensatt av tre målinger:

- *SlaaOpp()*: Henter 1000 tilfeldige deler fra databasen. For hver del kalles en null funksjon, *SkrivUtDel()*.
- *GjennomgaaFramover()*: Finner alle delene koblet til en tilfeldig valgt del, alle delene koblet til hver av dem, og så videre, ned til sju nivåer dypt (samme som dybde-først søking i en graf). Dette gir 3280 deler (siden hver del er koblet til tre andre) og 3279 koblinger som må hentes, med mulige duplikater. *GjennomgaaBakover()* gjør det samme, men følger koblingene bakover.
- *SettInn()*: Legger 100 deler med tre koblinger fra hver del inn i databasen. Endringene commites til disken.

```
void Sammenkoble(FysOID fraDelFysOID, FysOID tilDelFysOID,
                                                FysOID koblingFysOID)
{
   hent inn delen som koblingen starter på;
   if (delen har ingen koblinger fra seg) {
      oppdater delen til å peke på koblingen;
   } else {
      hent den første koblingen i lista av koblinger fra delen;
      gå fram til siste kobling i lista;
      oppdater siste kobling til å peke på den nye koblingen;
   }
   hent inn delen som koblingen slutter på;
   if (delen har ingen koblinger til seg) {
      oppdater delen til å peke på koblingen;
   } else {
      hent den første koblingen i lista av koblinger til delen;
      gå fram til siste kobling i lista;
      oppdater siste kobling til å peke på den nye koblingen;
   }
}
```

Figur 4-7 Sammenkoble() algoritmen

```
void Objektoperasjoner1()
{
   opprett en database, ei datafil og en objektkatalog;
   opprett enten et sidebuffer eller et objektbuffer på k
                                                    rammer;
   initialiser databasen med 20000 deler og 60000 koblinger;
   velg hvilken av benchmarkmålingene som skal kjøres;
   nullstill statistikk for databasen og bufferet;
   kjør målingen en gang og beregn kaldt resultat;
   kjør målingen 19 eller 49 ganger til og beregn varmt
                                                    resultat;
   skriv resultatene til fila "simuleringsdata.sdv";
}
```

Figur 4-8 Objektoperasjoner1() algoritmen

Hver av målingene kjøres 20 ganger, og antall sider lest og skrevet samt responstid registreres. Dersom objektbufferet brukes, kjøres målingene 50 ganger siden det trenger lengre tid på å bli skikkelig varmt. Først må objektbufferet fylles med objekter, som tar sin tid da dette skjer med ett og ett objekt. Deretter må vi ha en viss gjennomstrømning av objekter slik at den fragmenteringen som oppstår på grunn av dynamisk minneallokering blir framtvunget. Resultatene etter den første iterasjonen refereres til som kalde resultater, mens de varme resultatene er snittet av de ti siste iterasjonene. Et forskjellig sett av deler hentes i hver iterasjon. Merk at resultatene av gjennomgangen bakover vil variere mye, fordi antall deler som går til delen er variabelt. *Objektopersjoner1()*, se Figur 4-8, initialiserer testmiljøet, kjører en spesifikk benchmarkmåling og skriver resultatet til ei semikolondelt (SDV) fil. Ytelsesdataene kan nå importeres i Excel for analyse og graftegning.

OO1 benchmarken har flere ulemper, noen av dem er:

- Definisjonen av lokalitet
- Databasen må kjøres i ekte modus for å gjøre det mulig å følge koblingene mellom objektene
- Databasen har få objektstørrelser
- Ingen eksperimentelle betingelser som kan varieres

4.3 OO2

OO2 er en benchmark, som jeg selv har laget, med funksjonene vist i Figur 4-9. Databasen består her av 65535 objekter, hvilket krever på 2863 sider (11,2 MB). Størrelsen på objektene velges tilfeldig mellom 50 og 300 byte, som gir et snitt på 175 byte og en overhead på 14 / (175 + 14) * 100 = 7,4 prosent per objekt. *SettInnObjekter()* initialiserer databasen med objektene. Simulering av lokalitet gjøres ved å dele databasen i to, varme og kalde objekter. De varme objektene utgjør 10 prosent av databasen, 286 sider, og 90 prosent av forespørslene går til de varme objektene, referert til som 90/10 regelen. Valget av akkurat disse verdiene er applikasjonsavhengig, og 80/20 eller 70/30 kunne like gjerne vært valgt. Benchmarken kan også kjøres uten

```
void Objektoperasjoner2();
void InitialiserOO2(int antallNyeObjekter);
void SettInnObjekter(int antallNyeObjekter);
void HentObjekter(int antallOppslag, int modifikasjonsfaktor,
                                    int sammenklyngingsfaktor);
```

Figur 4-9 OO2 funksjoner

```
if ((rand() % 10) > 0) {
    logOID = ((rand() % (antallObjekter/10)) *
                        (100 / sammenklyngingsfaktor));
    } else {
    logOID = (rand() * 2 + (rand() % 2));
}
```

Figur 4-10 Valg av objekt

lokalitet slik at alle objektene har den samme sjansen til å bli referert for hver forespørsel. Dessuten kan OO2 kjøres med databasen i simuler modus.

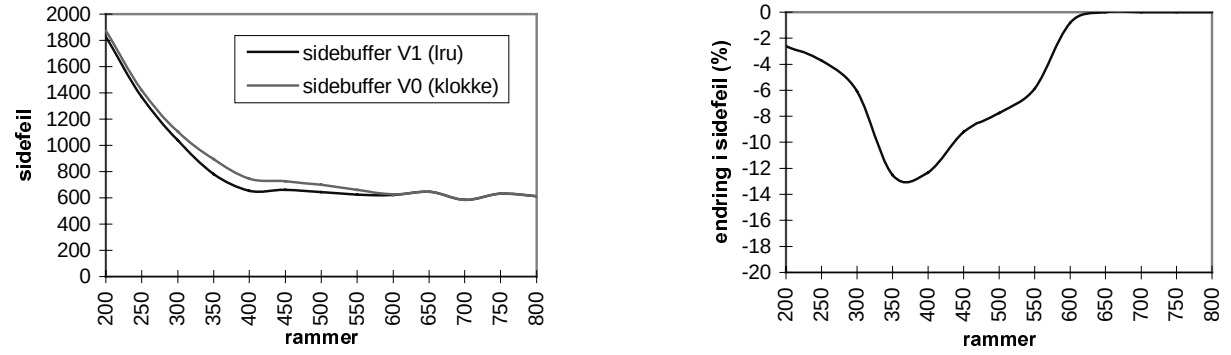
HentObjekter() henter *antallOppslag* objekter fra databasen. Jeg har satt antallet til 4000. Parameteren *modifikasjonsfaktor* angir hvor mange prosent av forespørslene som skal oppdatere objektet som blir hentet. Variabelen *sammenklyngingsfaktor* spesifiserer, i prosent, hvor godt de varme objektene skal være sammenklynget. For eksempel vil 10 % bety at 10 % av objektene på ei side vil være varme, mens 100 % vil si at alle objektene på sida er varme. Koden for å avgjøre hvilket objekt som skal velges i hver forespørsel er vist i Figur 4-10. Den er litt krøkete på grunn av *rand()* (fra standardbiblioteket) som kun returnerer et tilfeldig tall mellom 0 og 32767, mens der er 65535 objekter. *Objektoperasjoner2()* blir i prinsippet lik *Objektoperasjoner1()* (se Figur 4-8).

5. SIMULERINGSRESULTATER

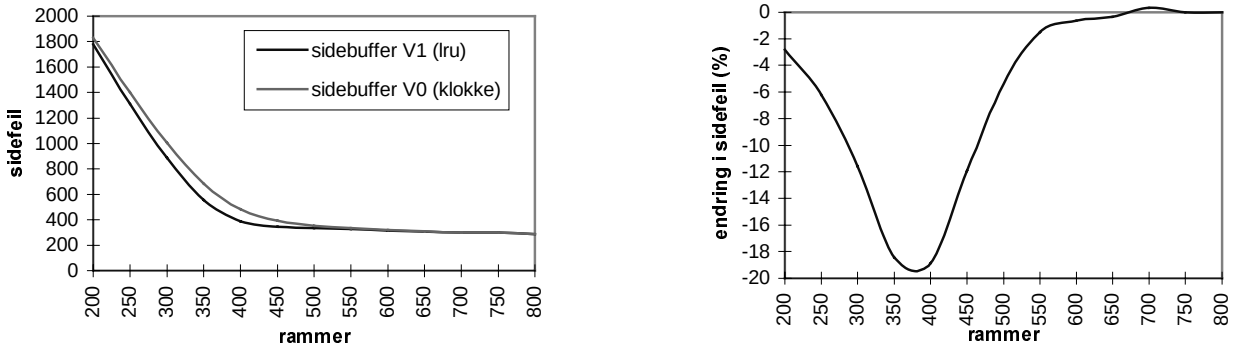
I dette kapittelet presenteres ytelsesmålingene som ble foretatt av simulatoren med benchmarkene OO1 og OO2. Vi skal hovedsakelig konsentrere oss om I/O ytelsen, målt i antall sider lest og skrevet under en sesjon, men også tiden som brukes til minnehåndteringen blir det tatt noen målinger av. Hovedspørsmålet er om objektbufferet gir bedre ytelse enn sidebufferet, eventuelt under hvilke betingelser ytelsen er bedre. Før vi kommer så langt, skal vi se på om det er forskjeller i ytelsen mellom de forskjellige versjonene av sidebufferet og objektbufferet isolert sett. Kunnskapen er relativ tynn dersom den ikke kan tallfestes; dette kapittelet vil forsøke å gjøre nettopp det. Leseren gjøres med dette oppmerksom på at mye data vil bli presentert og analysert.

5.1 Sidebuffer - LRU mot klokke

Det er viktig med en god algoritme til å velge hvilken side som skal kastes ut når bufferet er fullt, fordi disk I/O er så dyrt. Små forbedringer i den algoritmen gir store gevinster i databaseytelsen. To versjoner av sidebufferet ble derfor utviklet, en med klokke som utskiftingsalgoritme og den andre etter LRU prinsippet. Tidligere arbeid, som [EffelsbergHaerder84], har konkludert med at det er svært liten forskjell i ytelse mellom de to algoritmene. Under spesielle betingelser fant imidlertid jeg betydelige



a) kald ytelse



b) varm ytelse

Figur 5-1 Sidebuffer ytelse

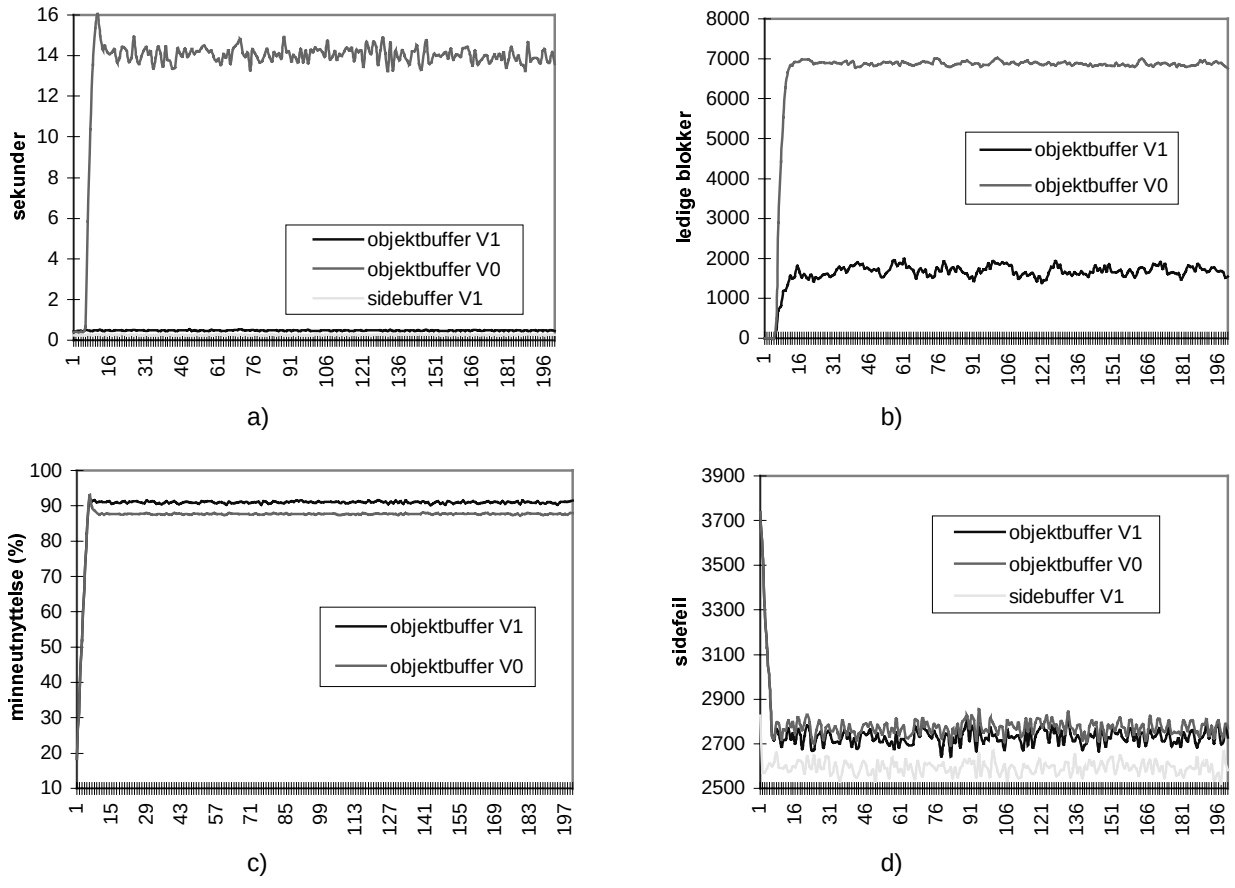
ytelsesforskjeller.

Resultatene vist i Figur 5-1 er fra OO2 benchmarken, som er gjennomgått i avsnitt 4.3. Lokaliteten er satt til 90/10, og aksessene oppdaterer ikke objektene. Det kritiske området er når bufferstørrelsen er rundt arbeidssettet som er på 286 sider. Ved en bufferstørrelse på 250 rammer får vi en reduksjon på 3,7 % i antall sidefeil for et kaldt buffer. Størst reduksjon får vi med 360 rammer, da er den 13 %. Når bufferet er varmt, blir tendensen forsterket. Opptil 19 % reduksjon i sidefeil kan nå oppnås, hvilket er ganske mye. Det som man også må merke seg er at over en viss bufferstørrelse, omtrent 600 rammer, er ytelsen så og si lik for begge utskiftingsalgoritmene. Dersom lokaliteten i referansene er lav, vil det også bidra til mindre forskjell mellom algoritmene. Siden LRU-algoritmen viste seg å være bedre, vil versjon 1 av sidebufferet brukes i resten av disse ytelsestestene.

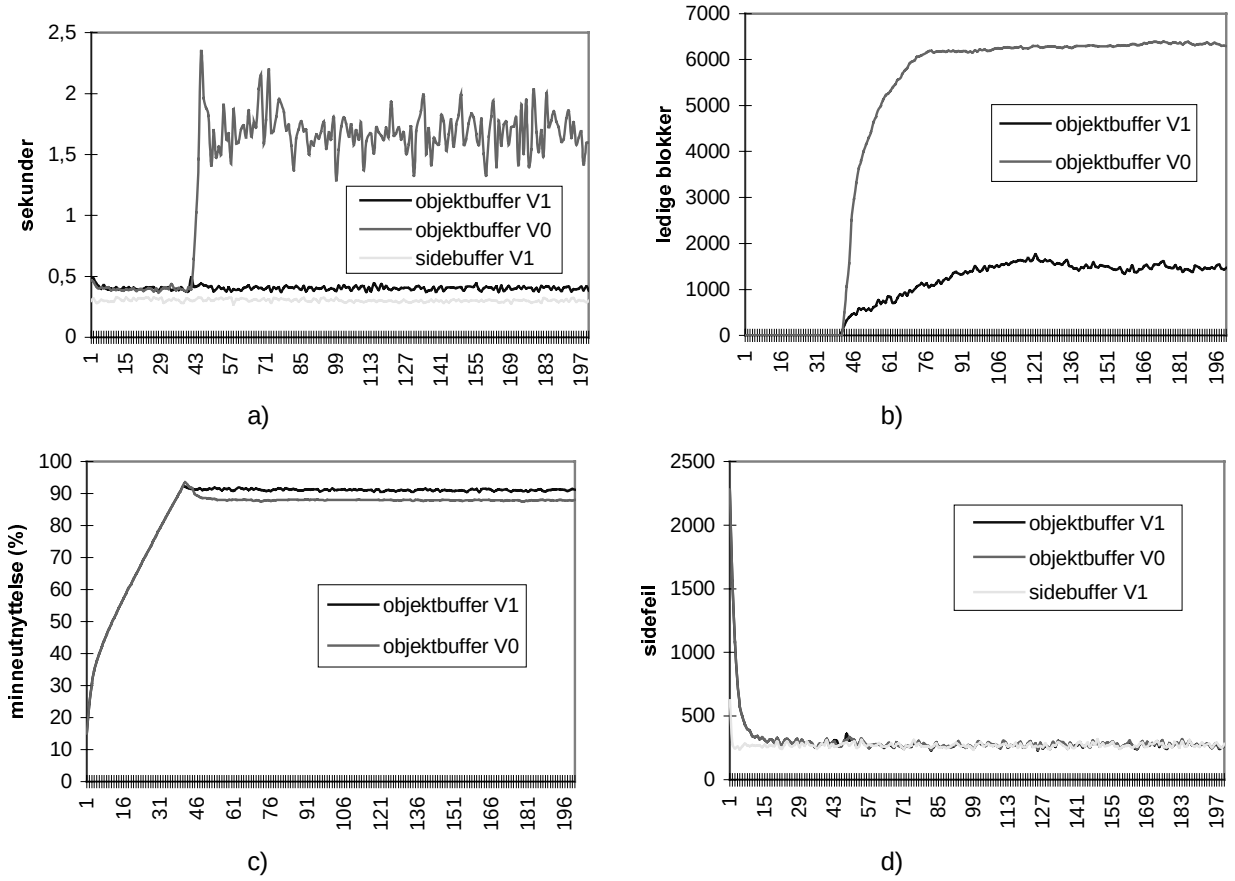
5.2 Objektbuffer V0 og V1

I versjon 1 av objektbufferet ble minnehåndteringen forbedret. Resultatet av forbedringen kan sees i Figur 5-2. OO2 ble kjørt 200 ganger, uten lokalitet og med en konstant bufferstørrelse på 1000 rammer. Denne kombinasjonen medfører for versjon 0 av objektbufferet at 72 % av de 4000 objektaksessene misser, noe som genererer mye søking for å allokere og deallokere blokker. Ved å kjøre databasen i simuler modus, måles CPU-tiden som går til minnehåndteringen på følgende måte:

```
tid = GetTickCount();
hent 4000 tilfeldige objekter;
tid = GetTickCount() - tid;
```



Figur 5-2 OO2 uten lokalitet



Figur 5-3 OO2 med lokalitet

Denne tiden ble for et varmt buffer omtrent 14 sekund for versjon 0. Versjon 1 brukte bare 0,4 sekund, som er vesentlig nærmere sidebufferet sine 0,26 sekund og en dramatisk reduksjon i forhold til versjon 0. Tidsmålingene er vist i Figur 5-2 a).

Grunnen til at versjon 0 av objektbufferet bruker så lang tid, er at lediglista på nesten 7000 blokker (se Figur 5-2 b)) må gjennomsøkes opptil flere ganger for hver objektmiss, mens i versjon 1 blir aldri denne lista gjennomsøkt. En annen interessant ting er den store reduksjonen i antall ledige blokker, til cirka 1700, ved bruk av versjon 1. Det kommer av at søking etter ei ledig blokk nå følger en best-fit strategi i steden for first-fit. Best-fit finner den blokka som passer best (uten å måtte gjennomgå lista), og i mange tilfeller vil den være av eksakt samme størrelse som objektet og det betyr færre splittinger av blokker. Mindre antall ledige blokker betyr redusert fragmentering og bedre minneutnyttelse, noe som fremkommer av Figur 5-2 c). Minneutnyttelsen (beregnet som summen av objektstørrelsene dividert på totalt bufferminne) øker fra 87,7 til 91 %. Bufferet i versjon 1 får altså plass til flere objekter, 19281 mot 18585, som gir færre objektmiss. Antall sidefeil for varmt buffer er 2866 for versjon 0, 2744 for versjon 1 og 2585 for sidebufferet. I tillegg til et betydelig kutt i CPU-tiden som brukes til minnehåndteringen, gir altså også bruk av versjon 1 av objektbufferet en reduksjon på 4,3 % i sidefeil grunnet mindre ekstern fragmentering. Objektbuffer V1 har 6,2 % flere sidefeil enn sidebufferet på grunn av fragmentering og at noe av bufferminnet benyttes til det underliggende sidebufferet (omtrent 100 rammer).

Figuren avslører også den dårlige kalde ytelsen til objektbufferet. For den første kjøringen har sidebufferet 2826 sidefeil. Objektbuffer V1 derimot har hele 3738 sidefeil, altså 32,2 % mer, og blir ikke skikkelig varmt før etter kjøring 6. Sidebufferet er varmt allerede etter en kjøring av benchmarken.

Dersom lokalitet (med 100 % sammenklynging) blir inkludert i benchmarken, blir resultatene annerledes. Med den samme bufferstørrelsen på 1000 rammer, ser vi av Figur 5-3 a) at CPU-tiden er nå 1,7 sekund for objektbuffer V0, mens den for de to

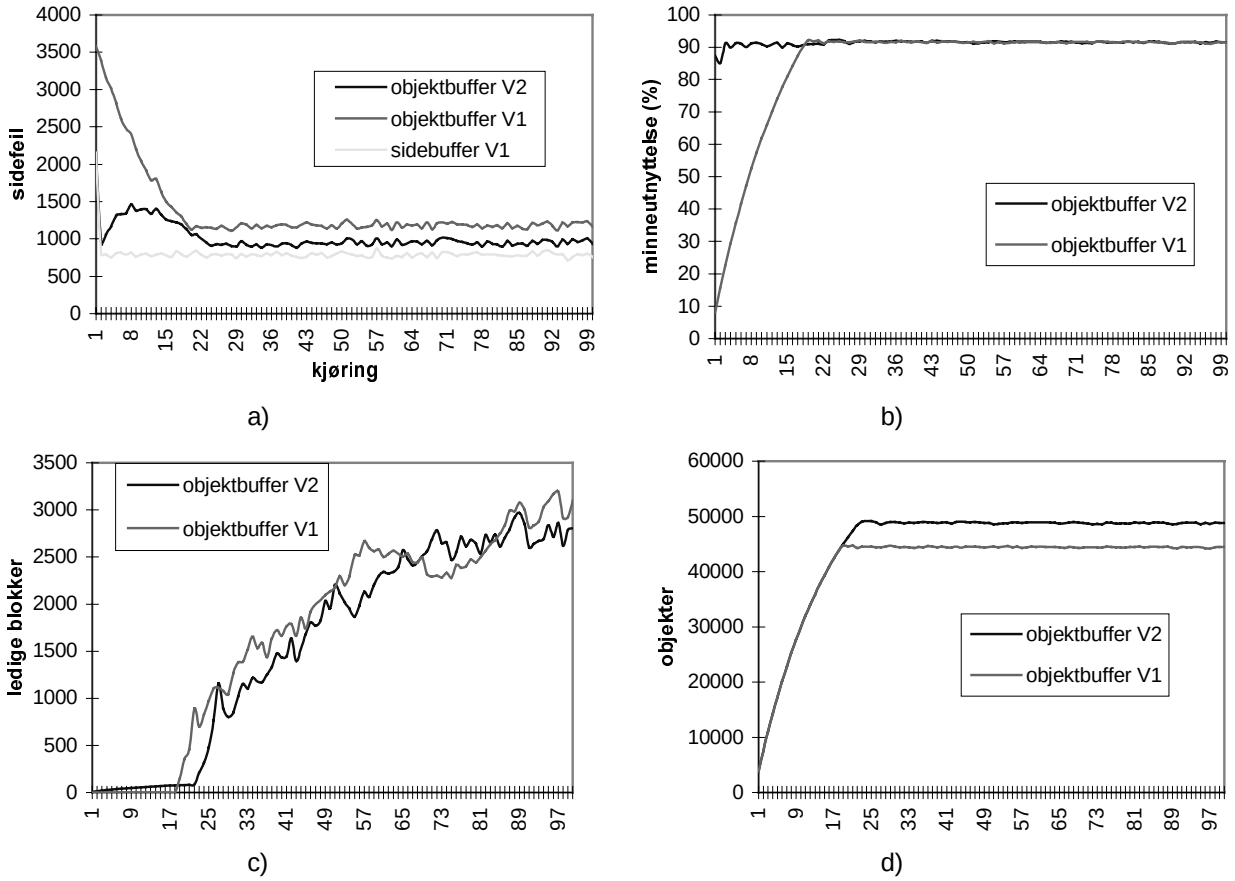
andre er omtrent uendret. Reduksjonen skyldes at bare 4,8 % av forespørslene misser slik at det blir mindre søking i lediglista når blokker skal allokeres og deallokeres. Antall ledige blokker er mindre, men det gir ikke utslag på minneutnyttelsen.

Det som kommer enda klarere fram i denne testen er den dårlige kalde ytelsen. Benchmarken må kjøres 40 ganger (se Figur 5-3 c)), det vil si at 160 000 objekter må aksesseres, før objektbufferet når en minneutnyttelse på 90 %. Sidebufferet, derimot, fylles allerede etter den første gangen. Selv om forskjellen i kald ytelse er større, er faktisk den varme ytelsen nesten lik.

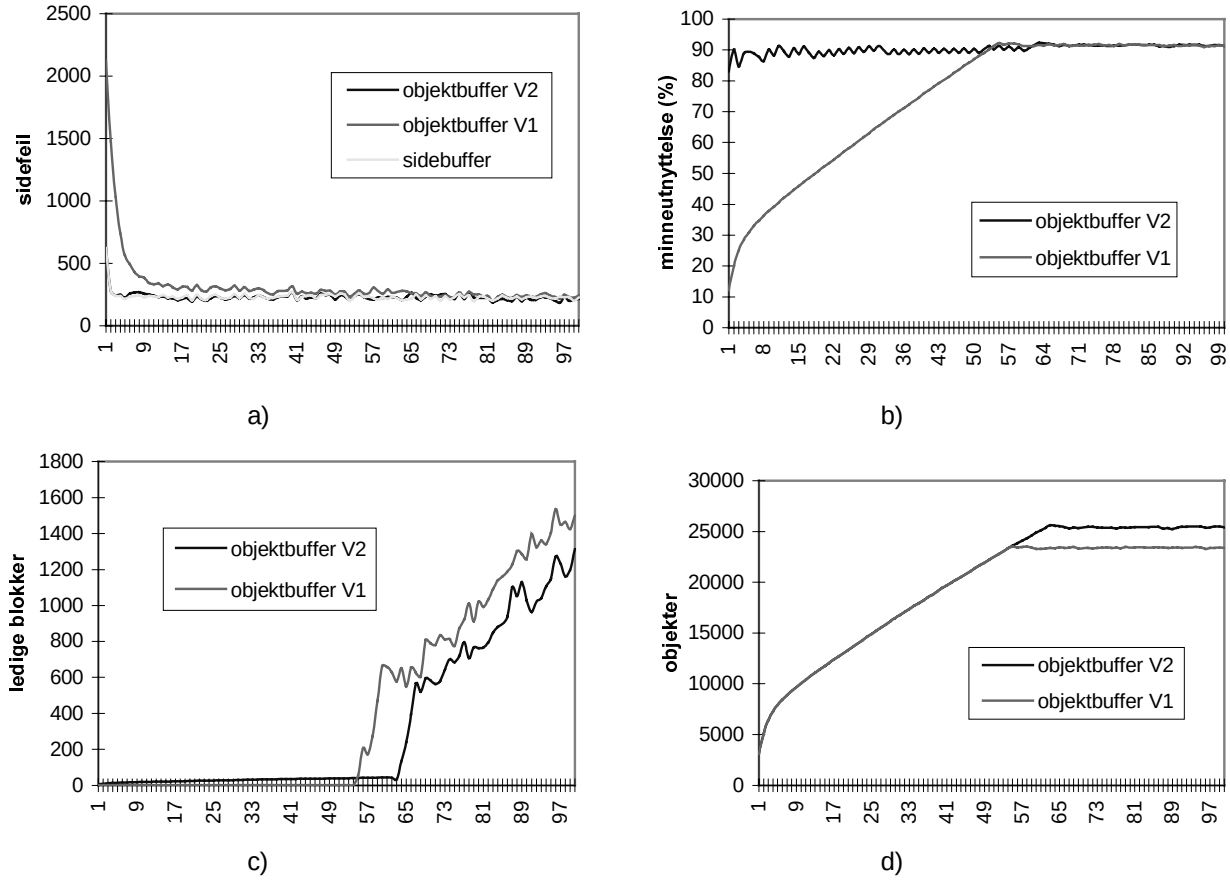
5.3 Objektbuffer V1 og V2

På grunn av den dårlige kalde ytelsen, ble versjon 2 av objektbufferet utviklet. Den versjonen varierer størrelsen på det underliggende sidebufferet dynamisk. I starten på en databasesesjon er det stort, og etter hvert som objektbufferet fylles med objekter minskes det underliggende sidebufferet, helt ned til et minimum på 27 rammer, samtidig som objektbufferets størrelse økes tilsvarende. Resultatene i Figur 5-4 er OO2 som er kjørt 100 ganger, uten lokalitet, med 100 % sammenklynging og med en konstant bufferstørrelse på 2295 rammer. Sidebufferet har best ytelse med 786 sidefeil for et varmt buffer, og dessuten blir det varmt allerede etter en kjøring. Objektbuffer V2 har 952 sidefeil, mens objekbuffer V1 har 1125.

Fra Figur 5-4 a) ser vi en kul på grafen for objektbuffer V2, fra kjøring 2 til 22. Det underliggende sidebufferet er på 84 segment, altså 2268 rammer, og fylles opp med objekter etter 1. kjøring. Objektbufferet er ett segment stort. Mellom kjøring 2 og 22 vil det vokse til 84 segment. Når størrelsen på sidebufferet reduseres, tas de 27 siste sidene i adresseområdet og kastes ut. Dette er ikke den mest gunstige måten å gjøre det på fordi de kanskje må leses inn igjen senere. Kulen vil sannsynligvis forsvinne med: a) en



Figur 5-4 OO2 uten lokalitet



Figur 5-5 OO2 med lokalitet

mer intelligent utkasting av sider, kanskje etter LRU prinsippet, og b) i aksessmønstre uten lokalitet vil det også være fordelaktig å overføre alle objektene på ei side til objektbufferet inntil det er fullt (aggressiv prehenting).

Forskjellen i virkemåten mellom de to objektbuffer versjonene er vist klart i Figur 5-4 b); bedre utnytting av bufferminnet. Siden objekbufferet fylles så sakte, er ideen å sette av lite minne (1 segment) i starten av en sesjon, og heller bruke det minnet til sidebufferet. På denne måten får objektbuffer V2 umiddelbart høy minneutnyttelse og derfor god kald ytelse. For objektbuffer V1 ser vi at benchmarken må kjøres 20 ganger før bufferet når 91 % utnyttelse, inntil da står store deler av minnet tomt. Antall ledige blokker er omtrent likt for begge. For versjon 1 er antallet 0, helt til bufferet er fullt og vi får en rask økning. Versjon 2 har en svak økning i antallet, fordi en ny ledig blokk på 27 rammer blir lenket inn etter hvert som bufferet øker i størrelse. Etter 22 kjøringer er maksimal størrelse nådd, da starter fragmenteringen for alvor med en stor økning i ledige blokker.

Med en gjennomsnittlig objektstørrelse på 175 byte, får sidebufferet plass til 53716 objekter. Objektbuffer V1 og V2 rommer henholdsvis 44404 og 48789 objekter i varm tilstand, se Figur 5-4 d). Versjon 2 tar flere objekter enn versjon 1 fordi det bare bruker 27 rammer til det underliggende sidebufferet, mens versjon 1 bruker 10 % som blir 230 rammer. Her kan man lure på hvorfor så mye som 10 % trengs til sidebufferet, men det er en avveining mellom kald ytelse og varm ytelse. Dersom prosenten minkes, blir den kalde ytelsen bedre og den varme ytelsen dårligere; og motsatt. For det adaptive objektbufferet er ikke dette en problemstilling i det hele tatt.

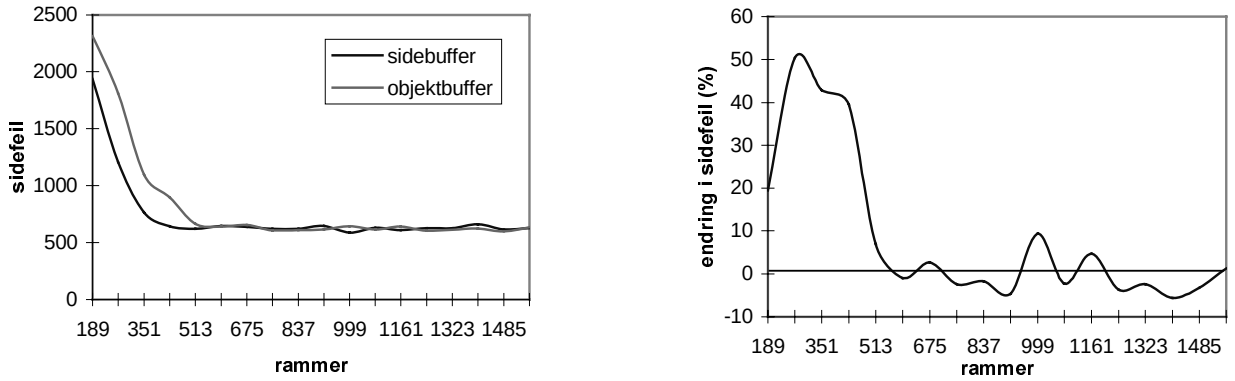
Figur 5-5 viser resultatet fra en test med de samme eksperimentelle betingelser som ovenfor, men med 90/10 lokalitet og et 1215 rammer stort buffer. Den kalde ytelsen til objektbuffer V2 er her like god som sidebufferet sin ytelse, og mye bedre enn objektbuffer V1. Den varme ytelsen er, kanskje noe overraskende, omtrent lik for alle tre. Det kommer nok av at bufferstørrelsen er mye større enn arbeidssettet, og derfor vil

90 % av aksessene gå til objekter som vil ligge i hovedminnet. Dersom vi ser på minneutnyttelsen i Figur 5-5 c), merker vi oss at nå bruker objektbufferet mye lengre tid på å fylles på grunn av den høye lokaliteten i referansene. Det medfører at den kalde ytelsen til versjon 1 blir forverret i forhold til eksperimentet ovenfor der ikke lokalitet ble brukt, men vil ikke påvirke ytelsen til sidebufferet eller versjon 2 av objektbufferet. Selv om antall ledige blokker er noe lavere for objektbuffer V2, er minneutnyttelsen for varmt buffer lik mellom objektbuffer V1 og V2; rett over 90 %. Først etter at benchmarken er kjørt 62 ganger er objektbuffer V2 fullt, da ligger 25416 objekter i bufferet (i snitt). Tilsvarende tall for V1 er 55 ganger og 23413. Sidebufferet er varmt etter bare noen ganske få kjøringer.

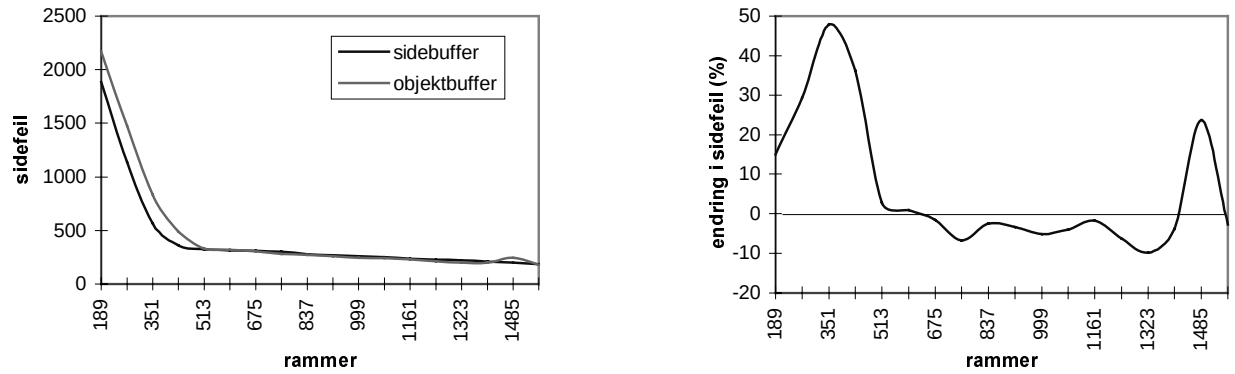
5.4 OO2

Alle resterende tester, i dette og neste avsnitt, er kjørt med sidebuffer versjon 1 og objektbuffer versjon 2. Fra nå av vil fokus være på hva som gir best ytelse; sidebuffer eller objektbuffer. Parametrene sammenklynging og modifikasjon vil være de eksperimentelle variablene sammen med bufferstørrelsen. Lokaliteten ligger fast på 90/10, selv om det kunne vært interessant å eksperimentert med den også; for eksempel 80/20 eller 70/30.

Når det gjelder størrelsesområdet som er valgt, blir dette på grunn av at det tar tid å kjøre tester, en avveining mellom et stort område med dårlig oppløsning, eller et mindre område med bedre oppløsning. Jeg har valgt å variere bufferstørrelsen mellom 189 og 1566 rammer. Det gir mulighet for å buffre fra et minimum på 6,6 % til et maksimum på 55 % av databasen. For å få nok data til å tegne pene grafer over dette størrelsesområdet, er skrittlengden satt til 81 rammer. Benchmarken må derfor kjøres 18 ganger hver gang en graf skal tegnes. Jeg har ingen grunn til å tro at en større database



a) kald ytelse



b) varm ytelse

Figur 5-6 100 % sammenklynging

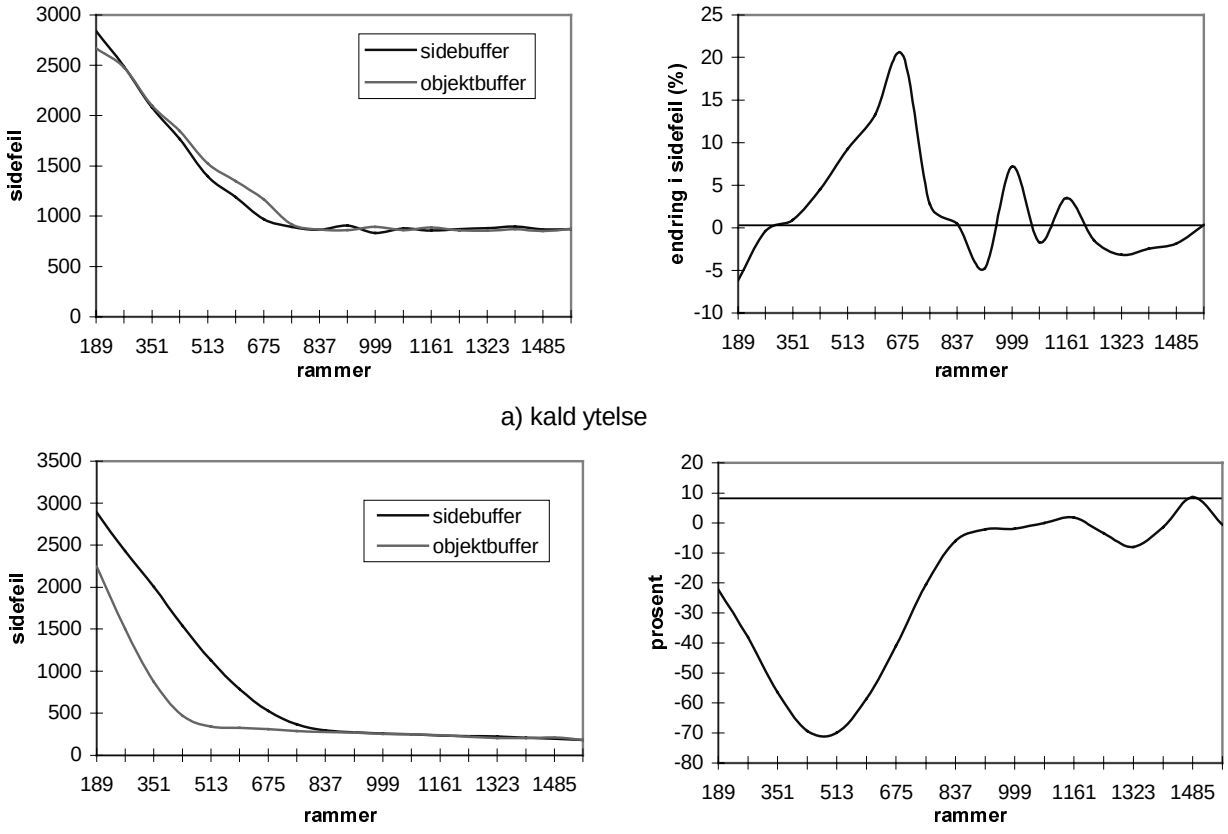
eller et større bufferstørrelsesvindu ville gitt andre resultat.

5.4.1 Sammenklynging

Figur 5-6 viser resultatene fra OO2 benchmarken med 100 % sammenklynging av de varme objektene. Kald ytelse beregnes fra den første kjøringen, mens varme ytelse er snittet av de siste 10 kjøringene av totalt 50 kjøringer (eller 20 kjøringer for sidebufferet). Som ventet kommer sidebufferet best ut når sammenklyngingen er maksimal. Den kalde ytelsen er opptil 50 % verre for objektbufferet dersom bufferstørrelsen er 280 rammer. Når bufferstørrelsen er i nærheten av arbeidssettets størrelse på 286 sider, er forskjellen størst. Med en bufferstørrelse på 550 rammer eller mer, blir ytelsen så og si lik for begge buffertypene, og antall sidefeil er da konstant. Den varme ytelsen følger samme mønster som den kalde, kanskje med litt mindre forskjeller. Vi ser også at etter som bufferstørrelsen øker, reduseres antall sidefeil for et varmt buffer. Med et stort nok buffer, 2900 rammer eller mer, vil antall sidefeil i varm tilstand være 0.

Med 50 % sammenklynging, se Figur 5-7, blir situasjonen annerledes. Den kalde ytelsen er nå nesten lik over hele området, bortsett fra når bufferstørrelsen er fra 594 til 715 rammer; da har objektbufferet 15-20 % mer sidefeil. Forklaringen ligger i at dette er den kritiske størrelsen for å kunne holde hele arbeidssettet i minnet. Det er nå for sidebufferet, på grunn av lav sammenklynging, dobbelt så stort som i forrige eksperiment - 572 sider. For objektbufferet vil det være det underliggende sidebufferet som avgjør den kalde ytelsen, fordi det er der de fleste objektene vil ligge etter en kjøring. Siden dette er noe mindre (cirka 5-6 segment) enn et rent sidebuffer, vil dette gi utslag på ytelsen i det kritiske området da det rommer en mindre andel av arbeidssettet.

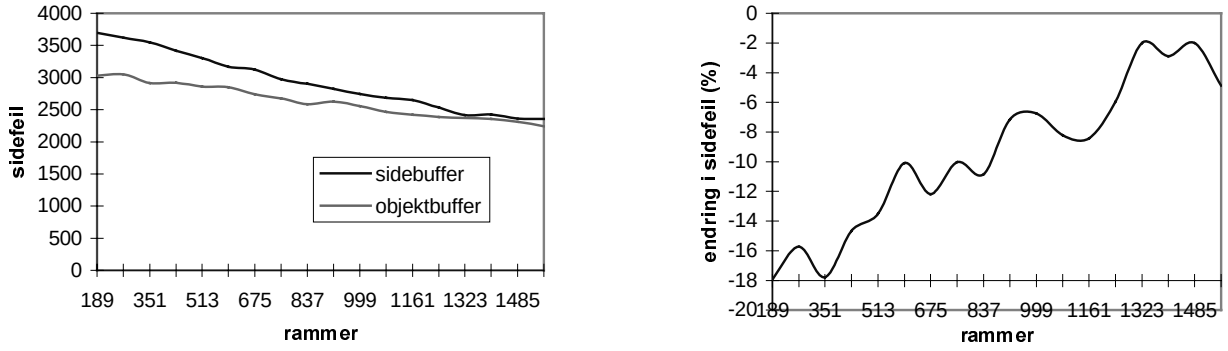
Når buffrene er varme, kommer objektbufferet klart best ut. Først fra en bufferstørrelse på 830 rammer og oppover, er ytelsen lik. Nå kommer objektbufferet sin



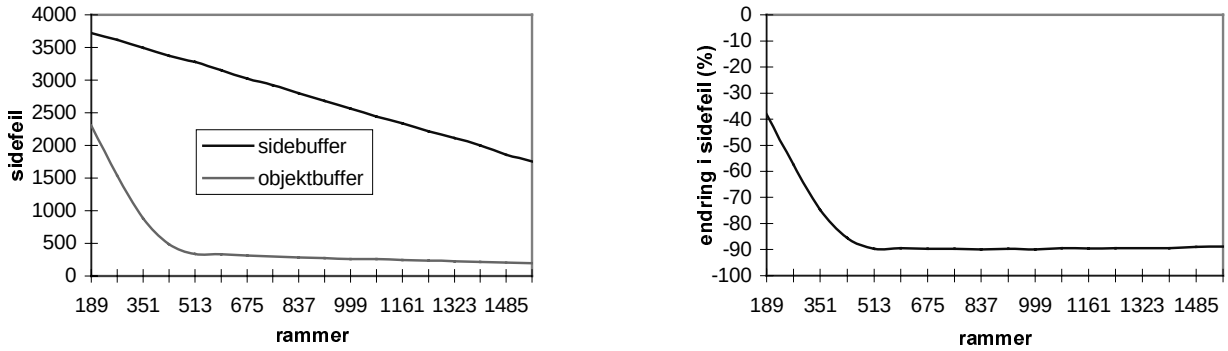
a) kald ytelse

b) varm ytelse

Figur 5-7 50 % sammenklynging



a) kald ytelse



b) varm ytelse

Figur 5-8 10 % sammenklynging

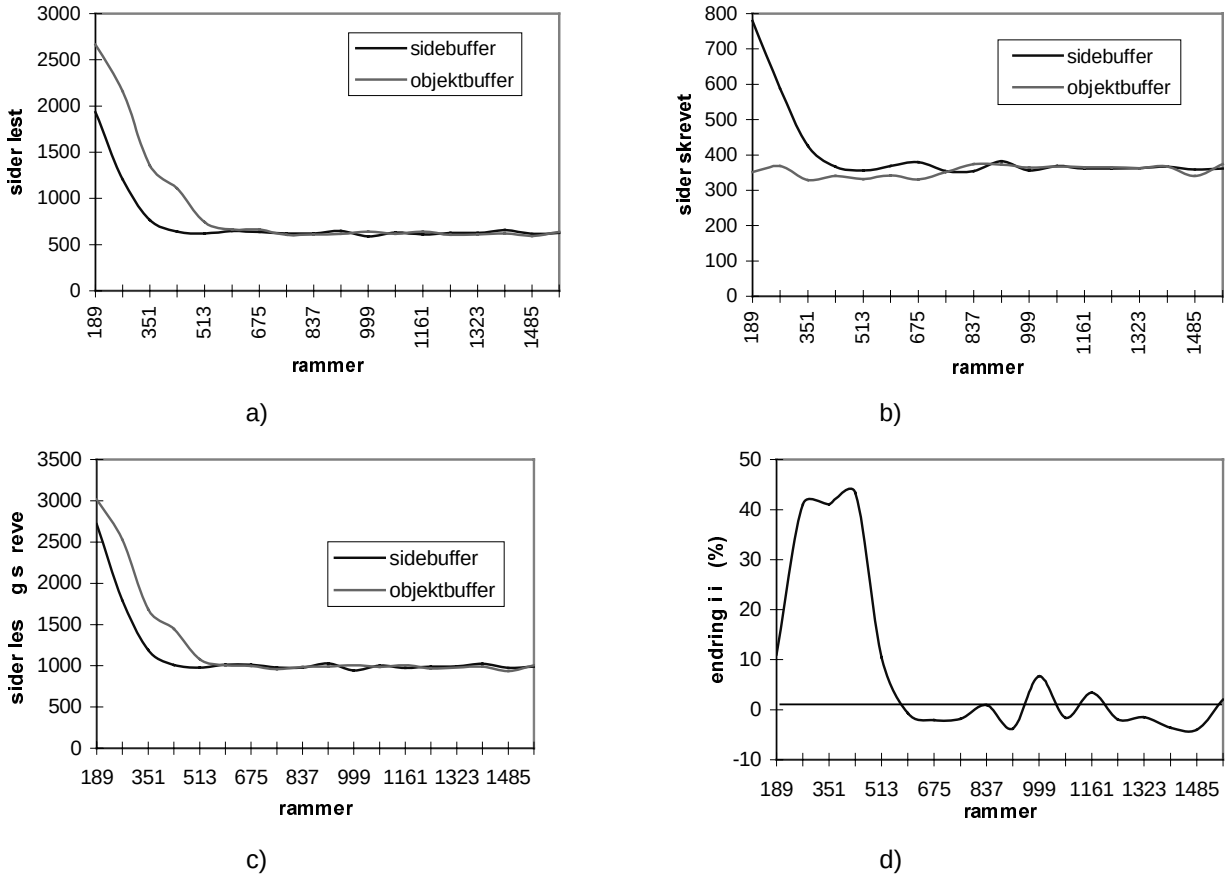
virkemåte til sin rett. I steden for en dobling av arbeidssettets størrelse, som er tilfellet for sidebufferet, forblir den konstant. Størrelsen på arbeidssettet er uavhengig av sammenklyngingen. Det kommer av at de varme objektene blir plukket ut av sin hjemmeside, slik at en slipper å buffre hele sida bare fordi det ligger noen varme objekter der. Ved et buffer på 470 rammer gir objektbufferet en reduksjon på 70 % i sidefeil - ganske betydelig med andre ord.

Ved å redusere sammenklyngingen til 10 %, øker ytelsesforskjellen ytterligere i favør til objektbufferet. Nå er også den kalde ytelsen bedre. For eksempel ved en bufferstørrelse på 351 rammer er den 18 % bedre. Den varme ytelsen til objektbufferet nærmest utklasser sidebufferet sin; med hele 90 % reduksjon i sidefeil over et stort størrelsesområde. Med så dårlig sammenklynging, må sidebufferet ha hele databasen i minnet for topp ytelse. Det betyr et buffer på 2863 rammer, mens objektbufferet begynner å få god ytelse allerede ved 500 rammer.

5.4.2 Modifikasjon

I dette avsnittet skal vi se på kald og varm ytelse når 25 % av objektaksessene oppdaterer objektet som blir hentet slik at databasen må oppdateres. Den kalde ytelsen, målt ved å kjøre OO2 med 100 % sammenklynging, er vist i Figur 5-9. For eksempel vil et sidebuffer på 351 rammer lese 766 sider; det samme som uten modifikasjon. Objektbufferet, derimot, leser 1353 sider mot 1094 sider dersom ikke objektene modifiseres; en økning på 24 %. Grunnen til dette er at objektbufferet må lese inn hjemmesida til et objekt når det er skittent. Men med et buffer på over 550 rammer, leser ikke objektbufferet flere sider enn sidebufferet. Sider skrevet går for begge buffertypene mot 363 stykker. Dette virker rimelig siden arbeidssettet er på 286 sider, og vi kan anta at mesteparten av disse sidene er skitne, samt noen av de kalde sidene.

Når sidebufferet er 432 rammer, eller mindre, øker antall sider skrevet raskt, mens objektbufferet skriver et konstant antall sider. Etter OO2 benchmarken er kjørt en gang vil objektbufferet inneholde 3120 objekter (eller 133 fulle rammer), der omtrent



Figur 5-9 kald ytelse, 100 % sammenklynging og 25 % modifikasjon

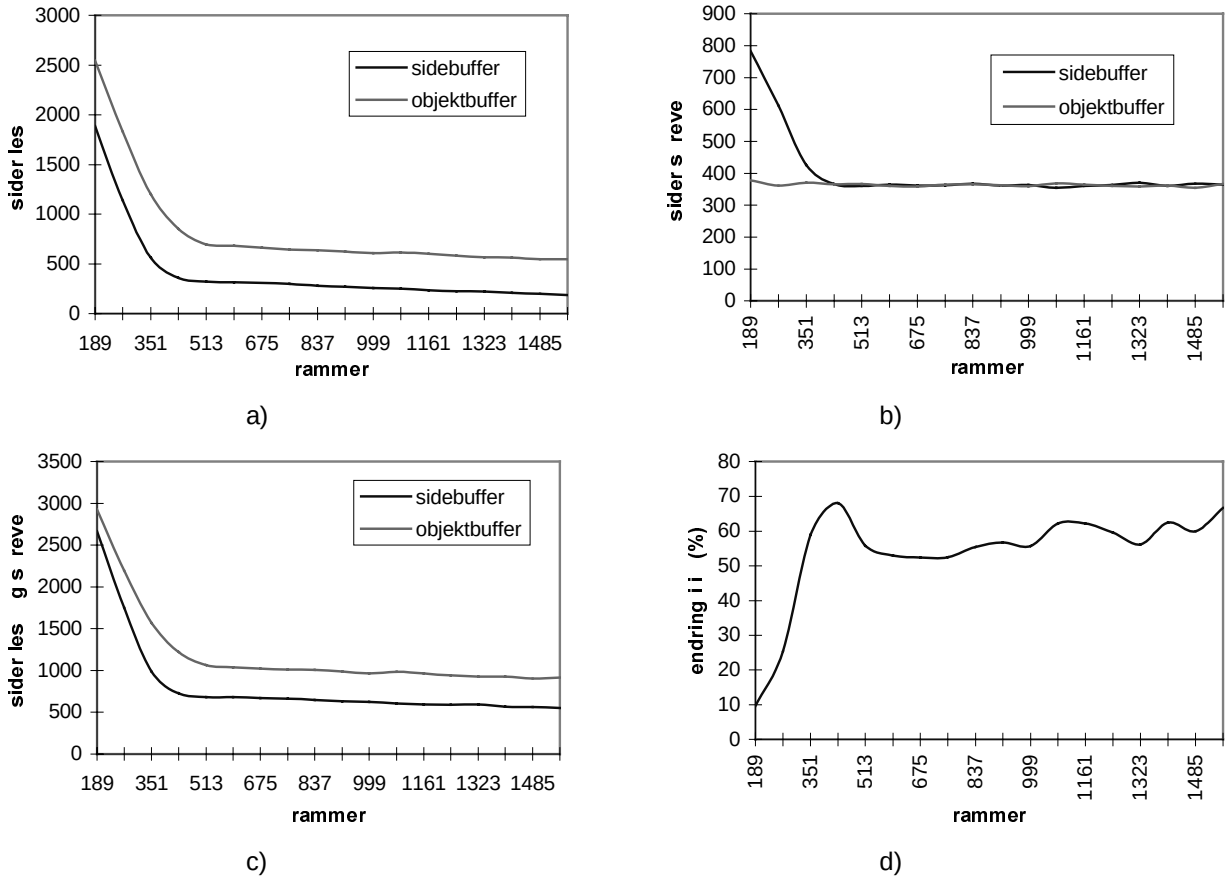
780 av disse objektene vil være skitne. Det vil derfor ikke være nødvendig å skrive skitne sider til disken før på slutten av benchmarken når *objektbuffer->Flush()* kalles. Sidebufferet virker på en annen måte. Når objektet oppdateres merkes sida umiddelbart som skitten, og må skrives til disken ved utkasting. Da bufferstørrelsen er mindre enn arbeidssettet, vil hver side i arbeidssettet måtte leses, og skrives, flere ganger.

Den kalde ytelsen totalt sett, summen av sider lest og skrevet, er for objektbufferet sin del 30 % mer enn for sidebufferet i området fra 230 til 452 rammer. Til høyre for dette området er ytelsen lik. Ytelsen til objektbufferet er svakt verre i forhold til tilsvarende målinger uten oppdateringer.

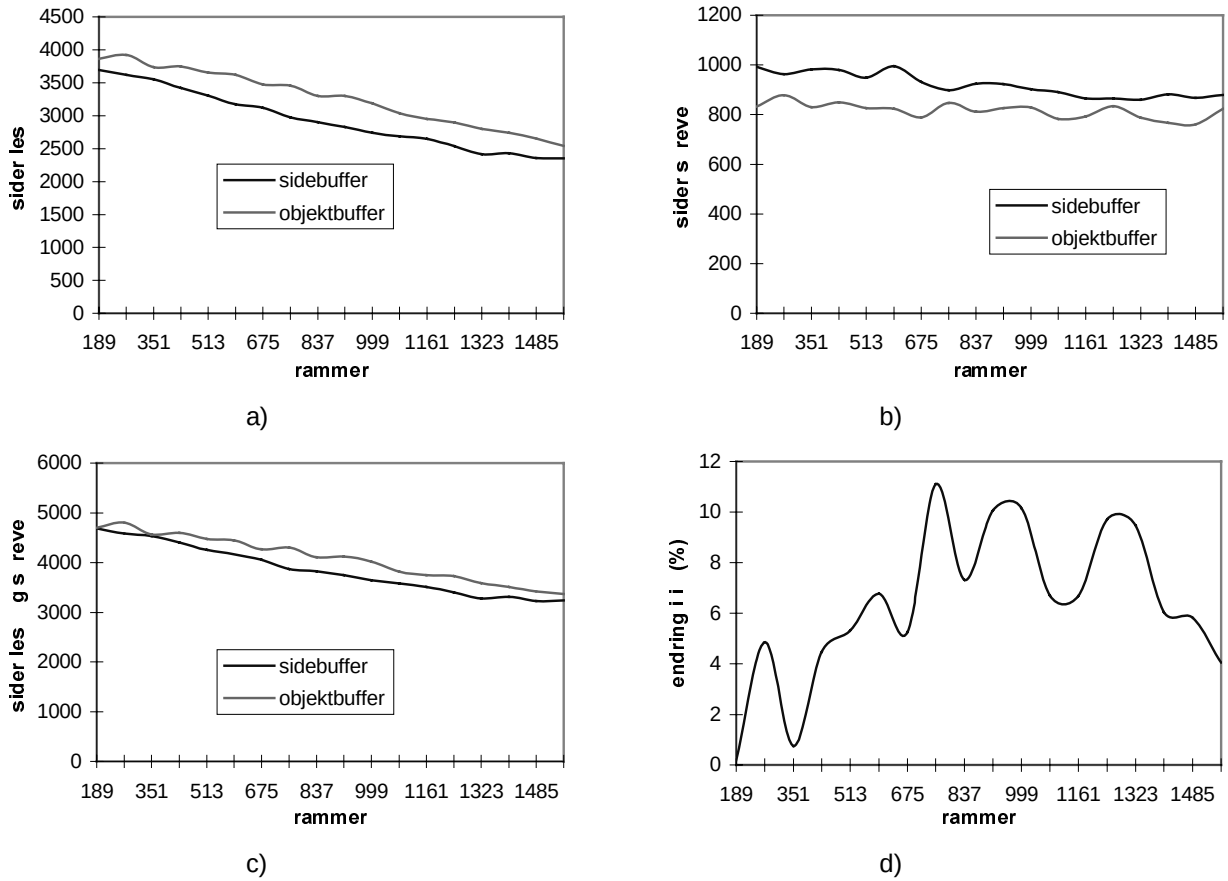
Den varme ytelsen er vist i Figur 5-10, og er vesentlig dårligere enn tilsvarende uten oppdatering. Nå er økningen i I/O over 50 % i nesten hele det målte størrelsesområdet. Objektbufferet lider av at det må lese inn hjemmesidene til de skitne objektene når de skal kastes ut eller flushes. Den konstante forskjellen i sider lest er cirka den samme som sider skrevet, noe som bekrefter at objektbufferet må lese en side ekstra for hver side som skal skrives.

For ikke å drukne i grafer og tall er resultatene fra et eksperiment med 75 % modifikasjon gitt i vedlegg B. Kombinasjonen med 50 % sammenklynging og 25 % modifikasjon er utelatt. Nå skal vi ta for oss målingene med 10 % sammenklynging og 25 % modifikasjon. Disse er vist i Figur 5-11.

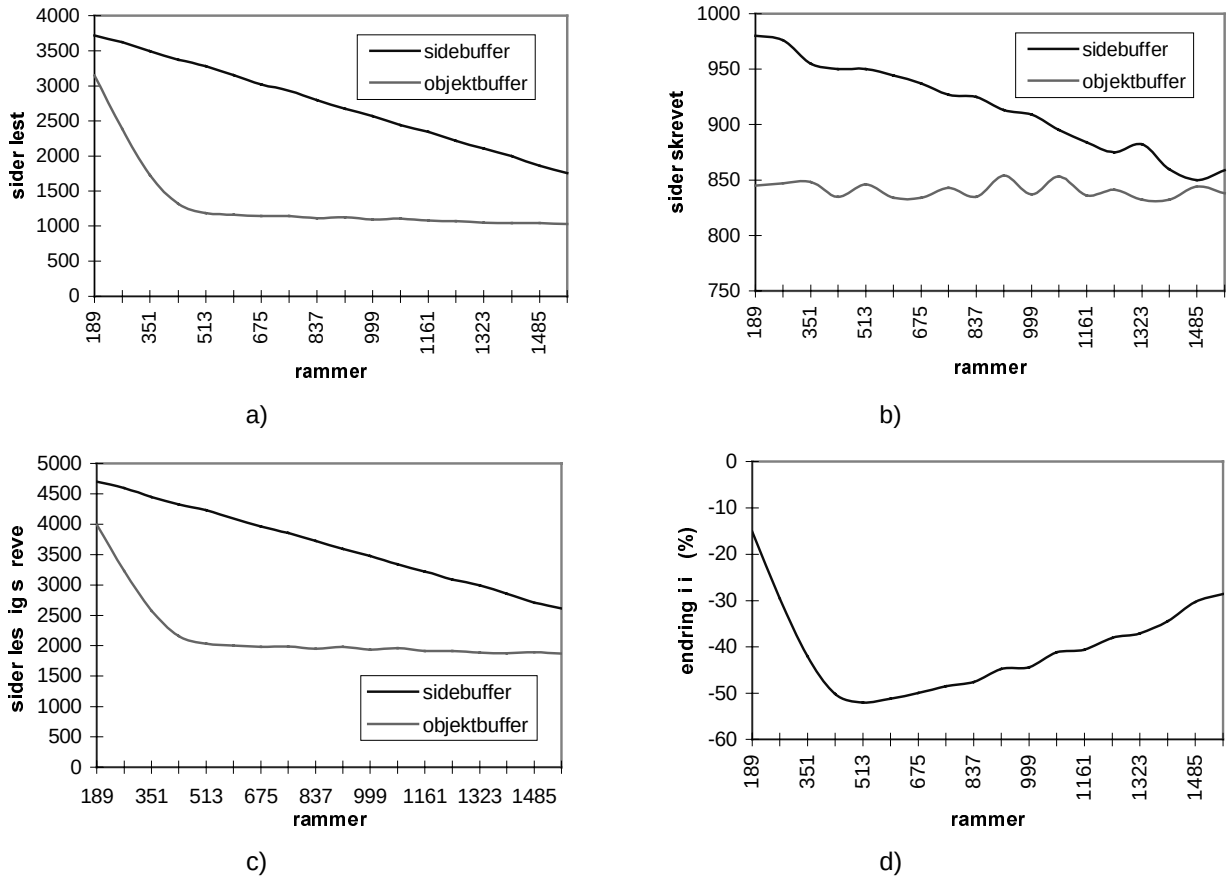
Dersom vi sammenlikner med de kalde resultatene uten oppdateringer, se Figur 5-8 a), der objektbufferet hadde mindre I/O over hele området, så er dette snudd til mer I/O over hele området. Forskjellen er ikke særlig stor; opptil 10 % mer I/O i forhold til et sidebuffer. Igjen taper objektbufferet på at flere sider må leses, mens sider skrevet er noe mindre. Siden sammenklyngingen er mye mindre nå, øker antall sider skrevet fra 363 til 8-900.



Figur 5-10 varm ytelse, 100 % sammenklynging og 25 % modifikasjon



Figur 5-11 kald ytelse, 10 % sammenklynging og 25 % modifikasjon



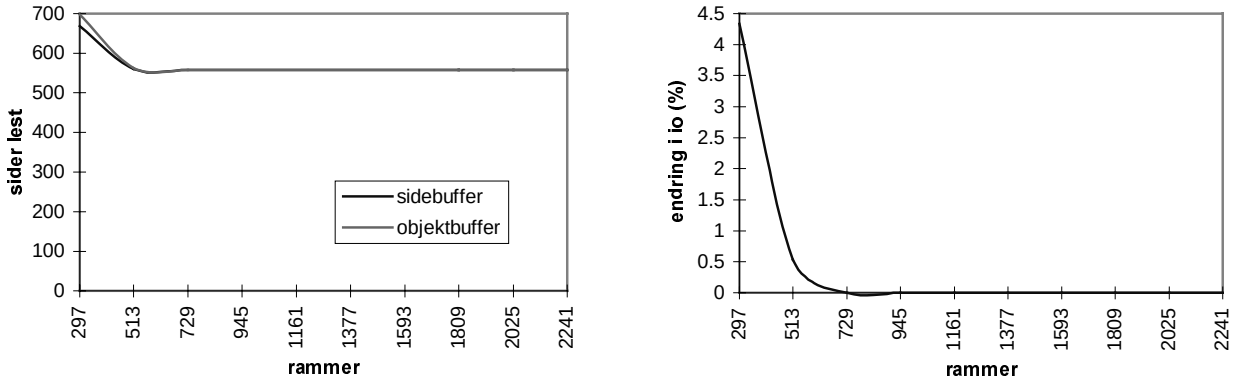
Figur 5-12 varm ytelse, 10 % sammenklynging og 25 % modifikasjon

Den varme ytelsen i Figur 5-12 viser en reduksjon i ytelsesforskjellen i forhold til Figur 5-8 b). Objektbufferet taper på økningen i sider som må leses, men tjener noe på at færre sider skrives. Sidebufferet har en jevn reduksjon i I/O med økende bufferstørrelse, mens objektbufferet har en rask reduksjon opp til 432 rammer og deretter er I/O'en omtrent konstant. I området mellom 270 og 1485 rammer gir objektbufferet mer enn 30 % reduksjon i I/O, og for 513 rammer er reduksjonen 52 %. Dette er vesentlig dårligere enn uten modifikasjon (men fremdeles bedre enn sidebufferet), hvor reduksjonen var på 90 % over et stort område.

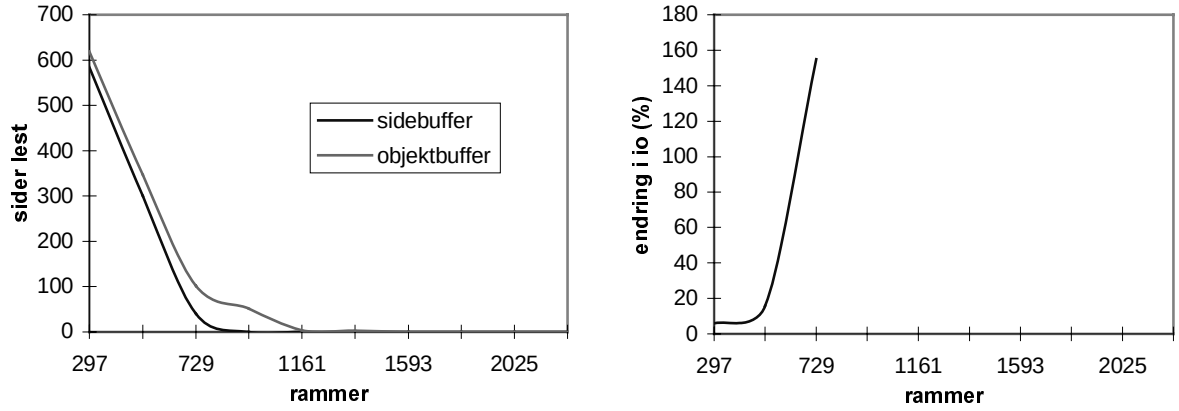
5.5 OO1

OO1 benchmarken, beskrevet i kapittel 4.2, må kjøres med databasen i ekte modus, og det tar da nødvendigvis lengre tid å utføre disse eksperimentene. OO1 databasen er på 2198 sider, der delene opptar 769 og koblingene 1429 sider. Størrelsesområdet for bufferet som målingene utføres for er nå utvidet til å ligge mellom 279 og 2241 rammer; hele databasen kan dermed buffres. Benchmarken består av målingene: SlåOpp, GjennomgåFramover, GjennomgåBakover og SettInn. Den siste målingen er utelatt. Både kald og varm ytelse måles.

SlåOpp henter 1000 tilfeldige objekter. Disse fordeler seg på 558 sider, noe som kan leses ut av Figur 5-13, som må leses inn. Grafen viser at den kalde ytelsen er omtrent lik. Dersom vi ser på den varme ytelsen, i Figur 5-13 b), har sidebufferet best ytelse - men ikke med så mye. Delene fordeler seg på 729 sider, så derfor vil et sidebuffer på mer enn 729 rammer gi null sidefeil i varm tilstand; det er årsaken til at grafen for endring i I/O ikke er tegnet over hele området. Denne målingen er uten lokalitet i referansene og av den grunn ikke så veldig spennende.



a) kald ytelse

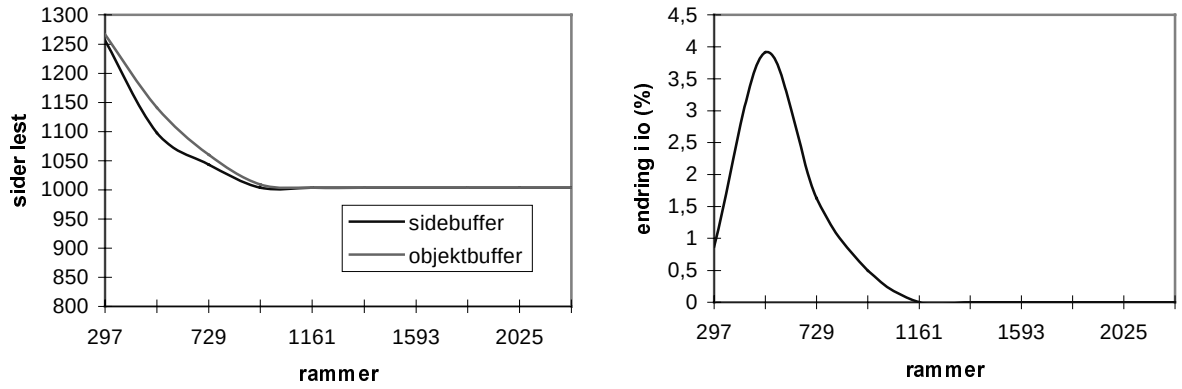


b) varm ytelse

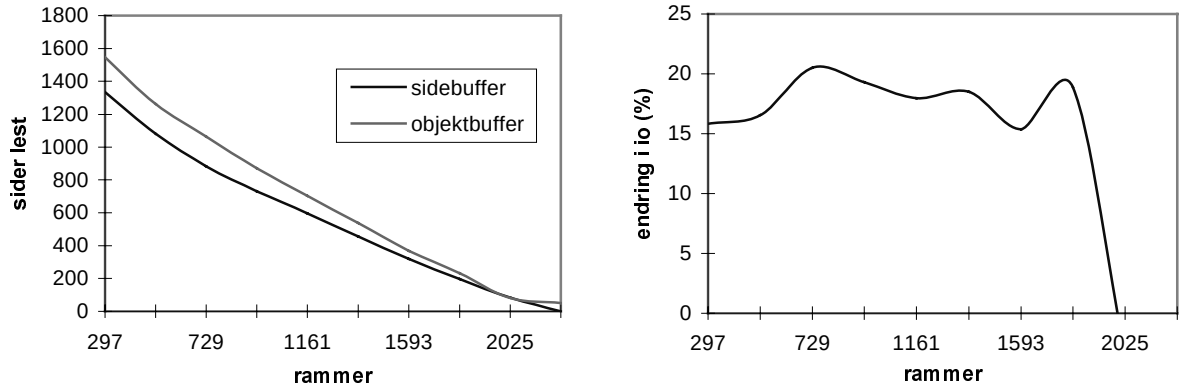
Figur 5-13 SlåOpp

Definisjonen av lokalitet i OO1 benchmarken er ikke særlig god. Den er slik at et objekt har 90 % sannsynlighet for at en kobling går til de 1 % fysisk nærmeste objektene. Når ei side er lest inn, blir den kanskje derfor referert flere ganger i gjennomgangen, og de sidene av databasen som berøres innskrenkes. Men definisjonen gir ikke noe klart arbeidssett; sider som brukes mye mer enn andre, og som ved en god uskiftingsalgoritme gjenkjennes.

Fra Figur 5-14 a) ser vi at den kalde ytelsen er så og si lik for GjennomgåFramover målingen, mens den varme ytelsen er 15 - 20 % verre (se Figur 5-14 b)) for objektbufferet, inntil bufferet er nesten like stort som databasen; da utjevnes forskjellen. Som vi ser avtar antall sider lest omvendt proporsjonalt med bufferstørrelsen, uten den karakteristiske kurveformen som lokalitet i referansene vanligvis gir. Dette forsterker inntrykket av at OO1 benchmarken ikke har en skikkelig spesifisert lokalitet, og det betyr at man kunne hatt en sideutskiftingsalgoritme etter FIFO eller RANDOM prinsippet uten det hadde fått noen betydning for ytelsen.
Et objektbuffer på 513 rammer leser 21 % mindre sider, når bufferet er kaldt, enn tilsvarende sidebuffer for GjennomgåBakover målingen. Over 1161 rammer, derimot, er datamengden som leses fra disken tilnærmet den samme. Også den varme ytelsen er forbausende lik over et stort område. Resultatene for GjennomgåBakover er vist i Figur 5-15.

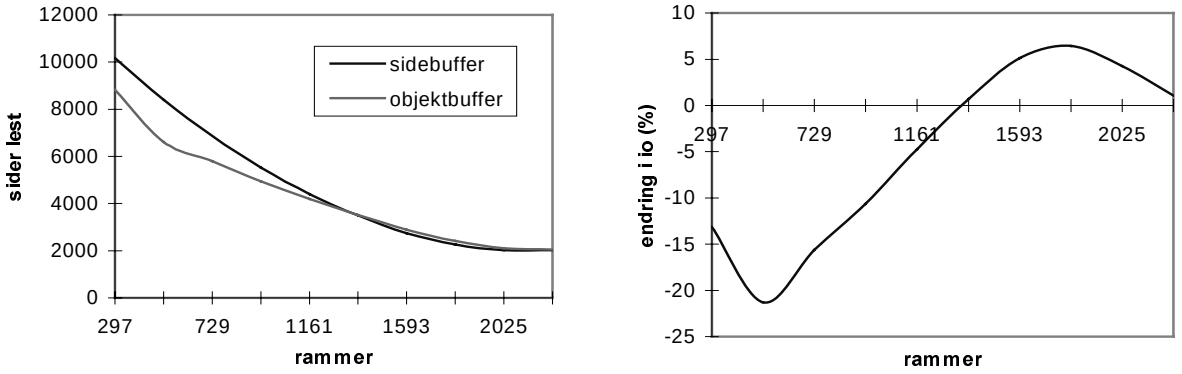


a) kald ytelse

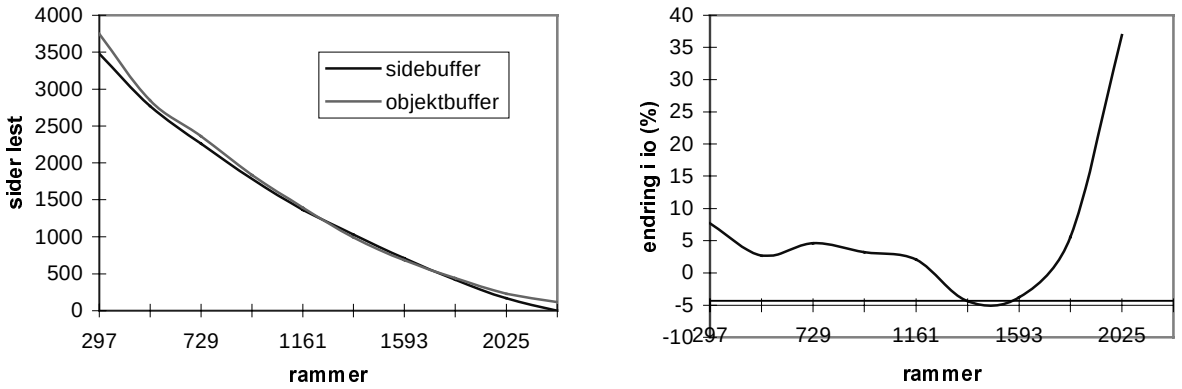


b) varm ytelse

Figur 5-14 GjennomgåFramover

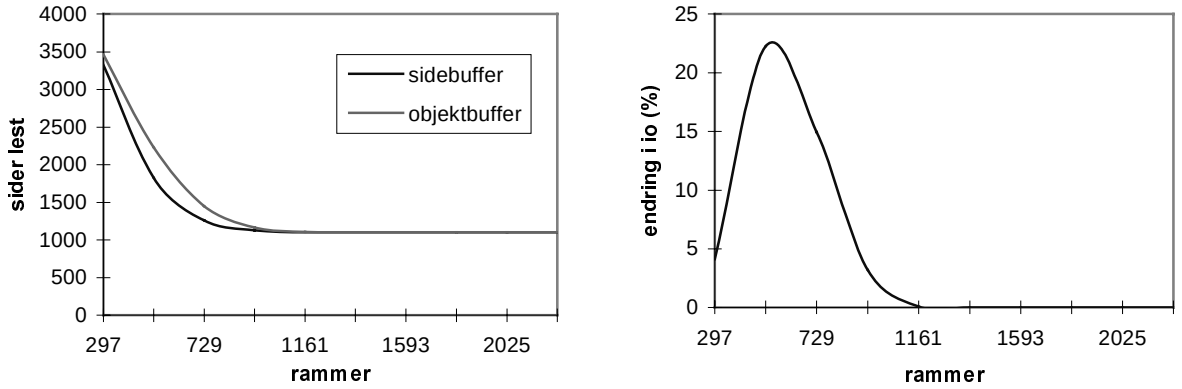


a) kald ytelse

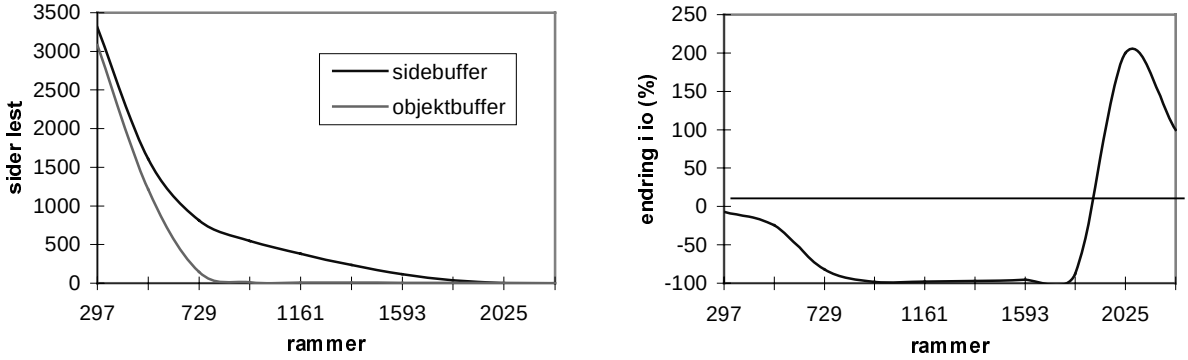


b) varm ytelse

Figur 5-15 GjennomgåBakover



a) kald ytelse



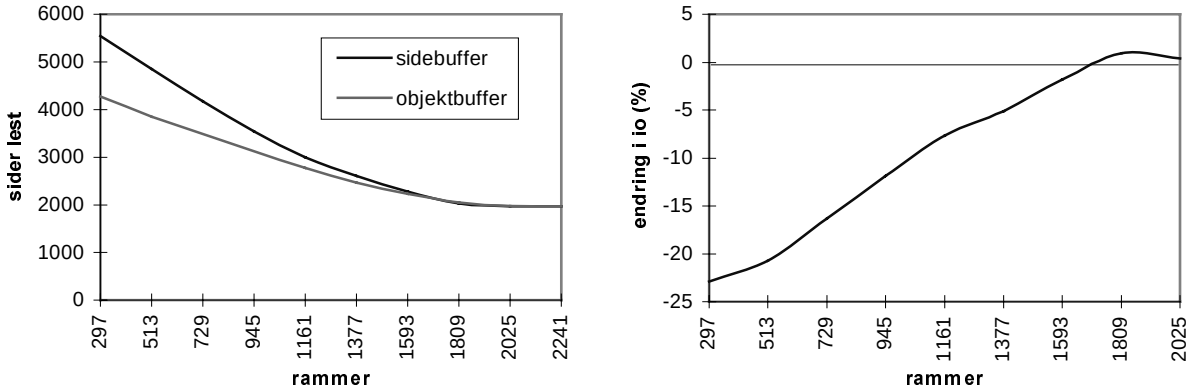
b) varm ytelse

Figur 5-16 GjennomgåFramover med objektlokalitet og sammenklynging

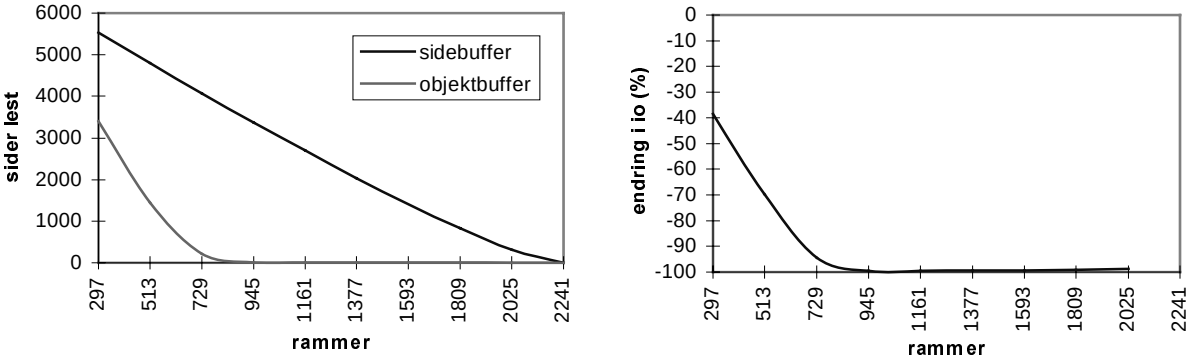
Jeg har tidligere påpekt at definisjonen av lokalitet i OO1 ikke er den aller beste. I de følgende eksperimentene er derfor den endret til: En kobling har 80 % sannsynlighet til å gå til 20 % av delene. Databasen deles nå i to, en varm og en kald del. Figur 5-16 er resultatene når de varme objektene ligger sammenklynget, og Figur 5-17 er resultatene når de ligger spredt.

Fra Figur 5-16 a) ser vi at gjennomgangen berører 1106 sider, så uansett bufferstørrelse må disse leses inn. Under 945 rammer har sidebufferet et overtak i ytelse, med et toppunkt rundt 530 rammer hvor objektbufferet har 23 % mer I/O. Når bufferet er varmt er det objektbufferet som har overtaket, med en rask reduksjon i sider lest opp til bufferet er 729 rammer. Dersom bufferstørrelsen økes ytterligere, er antall sidefeil tilnærmet null. Sidebufferet har også en rask reduksjon i sidefeil opp til 729 rammer. Da er antall sider lest 813. I området mellom 513 og 1809 rammer er reduksjonen i I/O ved bruk av objektbuffer på mellom 25 - 98 %.

Hvis vi nå sier at de varme objektene skal ligge spredt, får vi en del interessante resultater. Begge buffertypene får en sterkt forverret kald ytelse. Det minste antall sider som kan leses er 1967 sider, og det oppnås bare ved maksimal bufferstørrelse. Objektbufferet er nå best både for kald og varm ytelse. En av de store fordelene ved objektbuffer kommer nå til syne; den varme ytelsen er veldig lite påvirket av hvor godt de varme objektene er sammenklynget. Dette kan vi se ved å sammenlikne grafene for sider lest i Figur 5-16 b) og Figur 5-17 b). De er nesten identiske. Sidebufferet får en fullstendig kollaps, fordi det er avhengig av god sammenklynging.



a) kald ytelse



b) varm ytelse

Figur 5-17 GjennomgåFramover med objektlokalitet og uten sammenklynging

6. KONKLUSJON

I denne rapporten har vi sett på bruk av objektbuffer i steden for et sidebuffer. I den sammenhengen ble det utviklet en buffersimulator, som er i stand til å simulere begge buffertypene, og det ble foretatt ytelsesmålinger av simulatoren. Basert på erfaringene med implementeringen av simulatoren og resultatene fra benchmark kjøringene, vil jeg si at objektbuffer er en svært interessant og lovende bufferhåndteringsmetode. Noen av problemstillingene som dukker opp dersom et objektbasert buffer skal brukes er ekstraarbeidet med minnehåndteringen, plassen som forsvinner på grunn av fragmenteringen og den tiden som det tar å varme opp bufferet.

For at objektbufferet skal være brukbart i praksis, må det ha effektive algoritmer for dynamisk minnehåndtering. God minnehåndtering betyr minimal bruk av CPU'en ved allokering og deallokering av minneblokker, samt lav fragmentering for å få plass til flest mulig objekter. Jeg fant at den beste måten å implementere allokatoren på er ved å dele lediglista i flere lister basert på blokkstørrelsen, holde lediglista dobbeltlenket og legge inn kontrollinformasjon i både hodet og halen på reserverte og ledige blokker. Da kan allokering og deallokering av blokker, med sammenslåing av ledige naboblokker for minimal fragmentering, utføres på konstant tid.

Ytelseseksperimenter viste også, kanskje noe overraskende, en bedre minneutnyttelse ved å splitte lediglista på grunn av at vi da fikk en best-fit allokeringsstrategi. Minneutnyttelsen lå i mine eksperimenter med OO2 benchmarken på rundt 91 %. Fragmenteringen medfører at 9 prosent av minnet forsvinner. Av disse er 7,4 % intern fragmentering, mens den eksterne fragmenteringen bare utgjør 1,6 % av bufferminnet. Det er imidlertid umulig å si noe konkret om minneutnyttelsen, fordi den er avhengig av intern og ekstern fragmentering, gjennomsnittlig objektstørrelse, spredning på objektstørrelsene og størrelsen på objektene i forhold til bufferminnet.

Når objektene er svært små, kan den interne fragmenteringen bli dominerende. Det kan da være en ide å splitte allokatoren i to, en for små objekter og en for store objekter, der små objekter lagres i tabeller. Men tabellbasert allokering er ikke så enkelt som det virker, og gir ikke nødvendigvis bedre minneutnyttelse. Problemene er blant annet hvordan objekttabellene skal vokse og hvilke objekter som skal kastes ut. Jeg har mer tro på at når ei reservert blokk kommer under en viss størrelse, så legges det ikke inn kontrollinformasjon i den, og heller bruke en eller annen form for søppelinnsamling, gjerne som en bakgrunnsprosess, for å gjenvinne disse blokkene. Disse problemstillingene må utforskes videre i et framtidig arbeid for å optimalisere minneallokatoren til objektbufferet.

Generelt kan vi si at sidebufferet yter best ved høy sammenklynging, mens objektbufferet har høyest ytelse med lav sammenklynging. Dessuten har det adaptive objektbufferet nesten like god kald ytelse som det sidebaserte bufferet, men her trengs det noe mer arbeid for å se hva som skjer når en sesjon går i fra et arbeidssett til et annet. Vil det da være best å gjøre det underliggende sidebufferet stort igjen? Dersom det nye arbeidssettet har lav sammenklynging, tror ikke jeg det er gunstig, men det kan være det ved høy sammenklynging. Da vil det være en fordel å holde på de sidene som er lest inn helt til alle de varme objektene er flyttet over i det objektbaserte bufferet. Man må heller ikke glemme at dobbelbuffring, som jeg tidligere har omtalt, muligens kan være en god løsning. Problemet her ligger i å dynamisk bestemme den relative størrelsen til det sidebaserte underliggende bufferet. Den må tilpasses slik at sidebufferet i varm tilstand er omtrent like stort som de sidene av arbeidssettet som er godt sammenklynget.

Sammenklynging er den parameteren som er avgjørende for om det er fordelaktig med et objektbuffer, selv om den varme ytelsen til objektbufferet faktisk blir lite påvirket av hvor god sammenklyngingen er, fordi ettersom sammenklyngingen minker blir ytelsen til sidebufferet dramatisk verre. Opptil 90 % reduksjon i I/O ble oppnådd i mine tester. Dette er selvsagt avhengig av en rekke andre faktorer, slik som lokalitet i referansene og bufferstørrelse. Men når aksessene også oppdaterer objektene, dukker et ytelsesproblem for objektbufferet opp, fordi det gjennom en sesjon leser en god del flere sider. Grunnen er at når et objekt er skittent, må først hjemmesida leses inn før den kan skrives til disken. Allikevel viser ytelsesevalueringen at det er mulig å oppnå store reduksjoner i I/O ved bruk av objektbuffer, kanskje så mye som 50 % hvis 25 % av aksessene oppdaterer objektet som hentes.

For å kunne gi et definitivt svar på hva som er mest gunstig av disse to alternative bufferhåndteringsmetodene, vil mer implementering og eksperimentering være nødvendig for å få et bredere beslutningsgrunnlag. Men det jeg kan si er at det er fullt mulig å lage en dynamisk minneallokator med så høy ytelse at den ikke skal bli avgjørende for om utfallet blir positivt eller negativt.

VEDLEGG A: SKRIVEOPTIMALISERTE DATABASESYSTEM

Dette vedlegget beskriver kort hva skriveoptimalisering er, hvordan dette er implementert i simulatoren og resultatene fra ytelsestester.

A.1 Generelt om skriveoptimalisering

I et databasesystem er leseoperasjoner normalt mye vanligere enn skriveoperasjoner. Databasesystem er derfor ofte optimalisert for raske leseoperasjoner, selv om dette går på bekostning av skriveeffektiviteten. Etter hvert som størrelsen på hovedminnet øker, vil en kunne ha mer av hyppig aksesserte data liggende i hovedminnet. Dermed vil antall diskaksesser som er nødvendig for å lese data kunne reduseres. Mengden av data som må skrives vil derimot være den samme. Dette vil føre til at effektiv skriving av data kan bli viktigere enn effektiv lesing.

Den fundamentale ideen bak et skriveoptimalisert databasesystem er å forbedre skriveytelsen ved å buffre en sekvens av endringer av databasen i hovedminnet, og så skrive alle endringene til disken sekvensielt i en enkelt skriveoperasjon. I vanlige databasesystem medfører endringer skriving av mange små objekter, nesten alltid en aksess per objekt. Løsningen ligger i å eliminerer databasen og bare bruke loggen. Da kan endringene skrives kontinuerlig til disken. Problemet nå er at objektenes fysiske plassering endres slik at også objektkatalogen må oppdateres. Videre må store områder på disken alltid være tilgjengelige for skriving av data. Kontinuerlig rengjøring for å gjenvinne tomme segmenter blir nødvendig. Mer om skriveoptimalisering kan finnes i [NørvågBratbergsengen97] og [RosenblumOusterhout92].

A.2 Implementasjon av skriveoptimalisering i simulatoren

Objektbufferet ble til slutt utvidet med funksjonen *VelgModus()*. Den velger mellom normal og skriveoptimalisert modus. Ved kjøring i sistnevnte modus, opprettes ei ny datafil (SekvensiellFil) som skitne objekter skrives til. Datafila låser den siste sida i sidebufferet for å forhindre at den blir utkastet før den er full med skitne objekter. Når et skittent objekt skal kastes ut, settes det altså inn i *nyDatafil.* Dette gir en ny fysisk OID for objektet og objektkatalogen må derfor oppdateres. Nå skal det sies at dette er et sterkt forenklet skriveoptimalisert databasesystem, men prinsippet med å samle opp mange små ikke sekvensielle skrivninger til færre og større sekvensielle skrivninger blir demonstrert. Reduksjonene i I/O i skriveoptimaliert modus er betydelige. I normal modus, med et lite underliggende sidebuffer, må vi for hvert skittent objekt først lese inn sida, for deretter å skrive den tilbake igjen med det modifiserte objektet. Altså blir dette 2 I/O aksesser per objekt. Når objektbufferet kjøres i skriveoptimalisert modus vil aldri sider måtte leses når databasen skal oppdateres, samt at de sidene som skrives vil være fulle av skitne objekt.

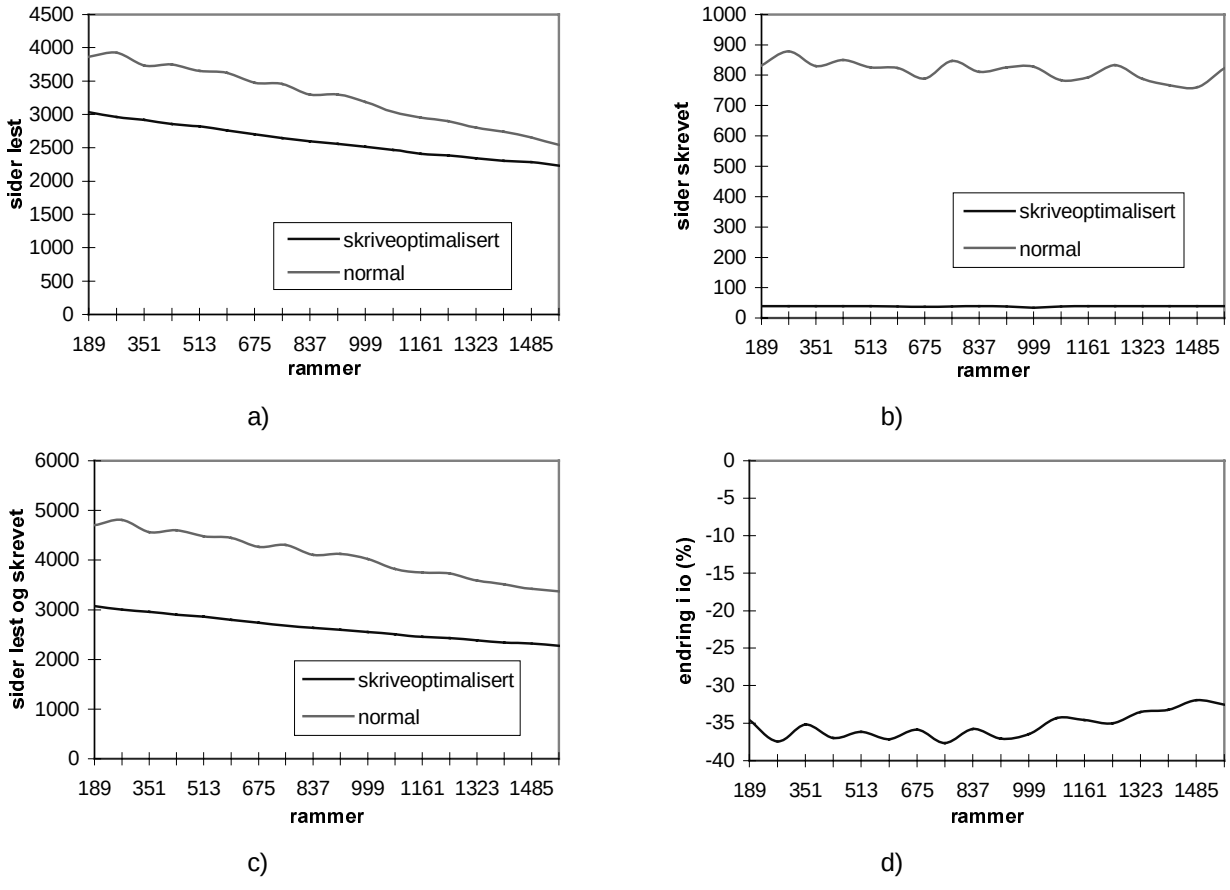
Ettersom et objekt nå vil flyttes til ei anna side dersom det endres, vil databasen vokse og sidene blir fragmenterte. Det som må gjøres, men som ikke er implementert i denne simulatoren, er å defragmentere databasen ved å pakke objektene. På den måten får vi frigjort sider. På grunn av dette blir denne simuleringen relativt urealistisk (og

derfor bare et vedlegg), men den hinter om potensialet som ligger her når det gjelder å redusere I/O.

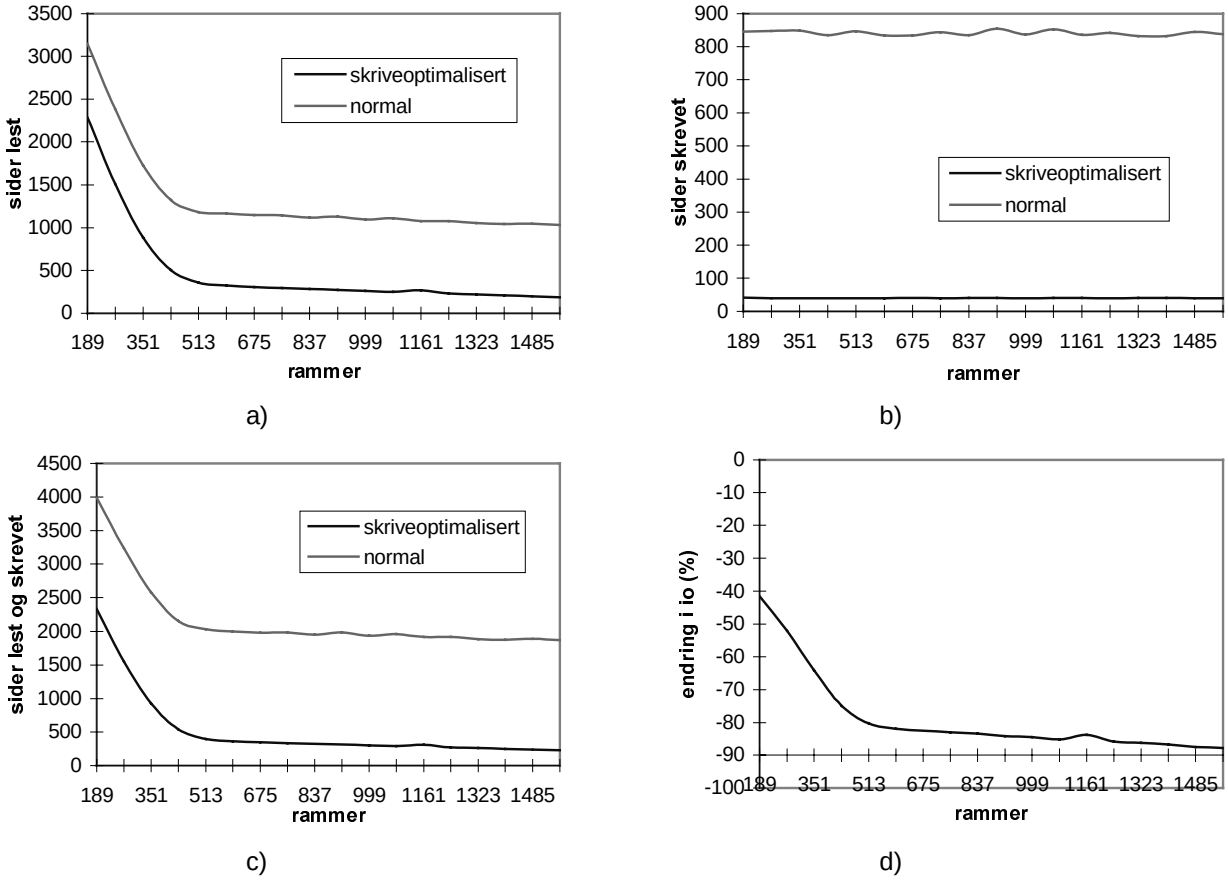
A.3 Ytelsesresultater

Ytelsen, både varm og kald, til objektbufferet er målt i skriveoptimalisert og normal modus med 25 % modifikasjon. I kald tilstand gir skriveoptimalisering en reduksjon, alt etter bufferstørrelsen, på mellom 312 til 961 sider som må leses. Reduksjonen kommer av at bufferet ikke behøver å lese inn hjemmesidene til skitne objekter. I sider skrevet, derimot, kommer en dramatisk reduksjon fra over 800 i normal modus til kun 41. Hver av disse sidene inneholder i snitt 23 skitne objekter, noe som gir 943 objekter som må skrives tilbake. Siden 25 % av de 4000 objektaksessene oppdaterer objektet, virker dette rimelig.

For den varme ytelsen blir den relative forskjellen i sider lest større, mens sider skrevet er uendret. For et buffer på 513 rammer eller mer som kjøres i skriveoptimalisert modus vil, se Figur 6-2 c), gi en imponerende reduksjon i I/O på over 80 %.



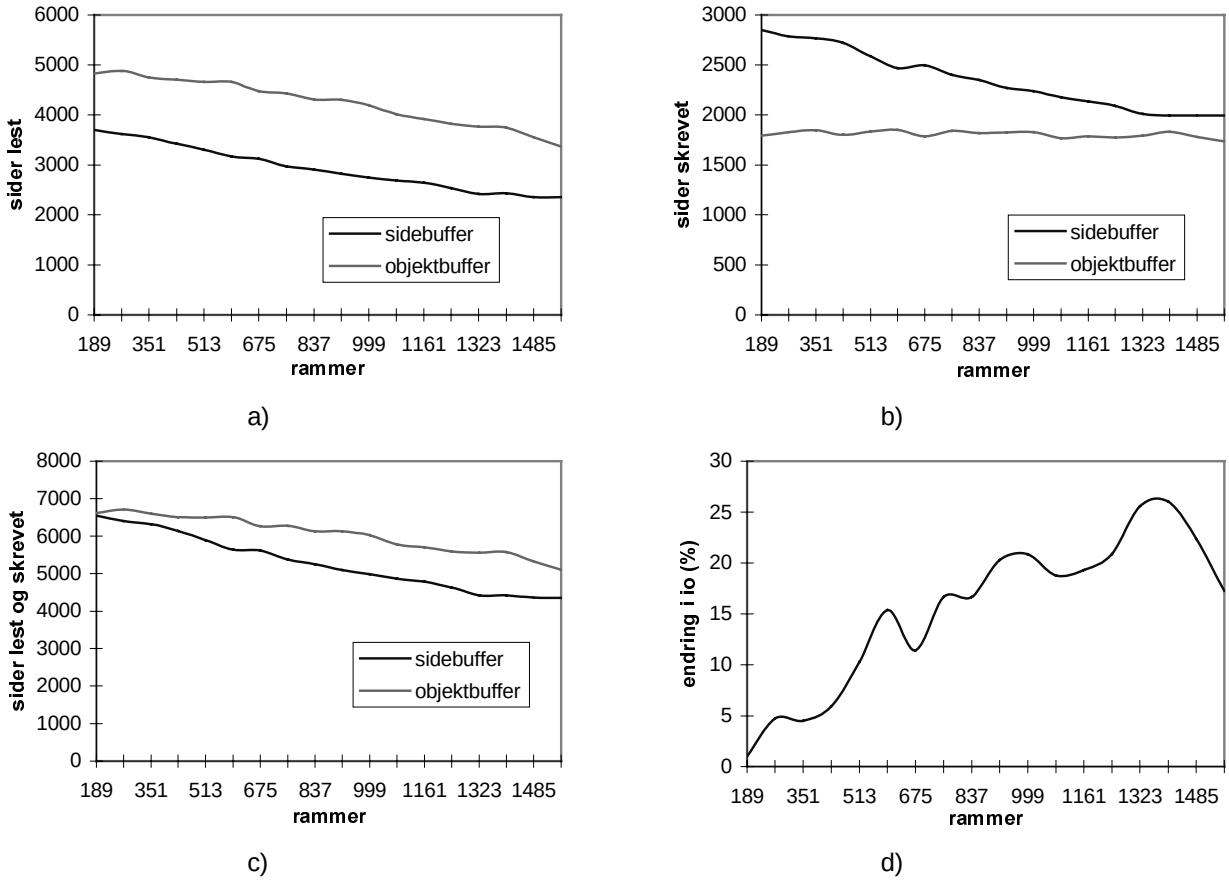
Figur 6-1 kald ytelse, sammenklynging 10 % og modifikasjon 25 %



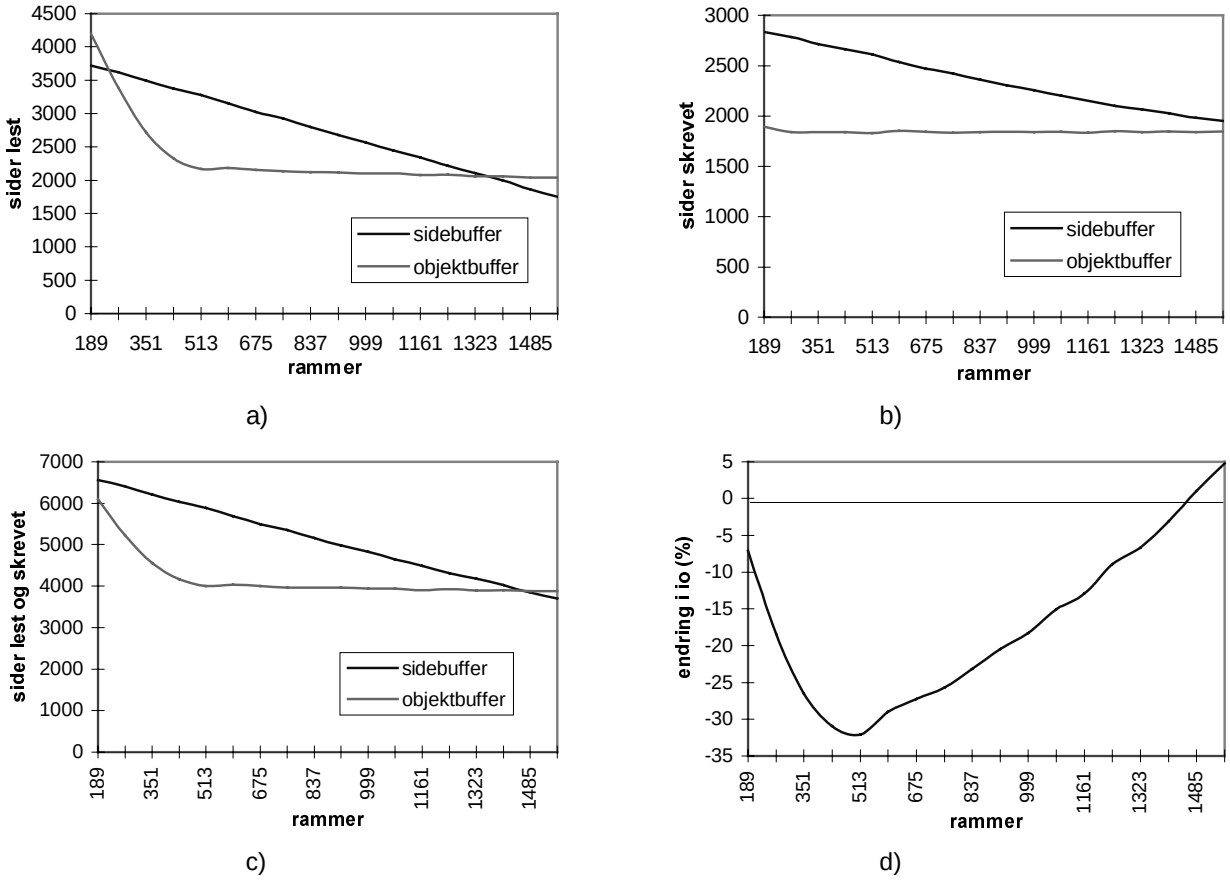
Figur 6-2 varm ytelse, sammenklynging 10 % og modifikasjon 25 %

VEDLEGG B: OO2 MED 75 % MODIFIKASJON

Med en sammenklyngingen på 10 % og en modifikasjon på hele 75 % får vi den kalde og varme ytelsen vist i henholdsvis Figur 6-3 og Figur 6-4. Figurene viser en forsterkning av den tendensen vi så ved 25 % modifikasjon; objektbufferet får problemer. Det som vi må merke oss for den kalde ytelsen er at objektbufferet leser rundt 1300 sider mer over et stort område, men skriver fortsatt mindre sider enn sidebufferet. Totalt sett er sidebufferet best. Opptil 25 % mer I/O, for et buffer på 1364 rammer, vil bli resultatet dersom objektbufferet brukes i steden for sidebufferet.
I varm tilstand leser objektbufferet 2079 sider, og fra Figur 6-4 b) ser vi at sider skrevet er omtrent likt som i kald tilstand; litt mer kanskje. Igjen lider objektbufferet av alle sidene som må leses dersom en stor andel av aksessene oppdaterer objektene. Vi får ikke den ytelsesforbedringen som vi skulle tro med så lav sammenklynging. Reduksjonen i I/O for objektbufferet er nå bare maksimalt 30 % mot 90 % uten oppdateringer. Faktisk er sidebufferet raskest for bufferstørrelser over 1485 rammer.



Figur 6-3 kald ytelse, sammenklynging 10 % og modifikasjon 75 %



Figur 6-4 varm ytelse, sammenklynging 10 % og modifikasjon 75 %

VEDLEGG C: KILDEKODE

Dette siste vedlegget inneholder hele kildekoden til simulatoren. Koden er skrevet i C++ med bruk av enkelte funksjoner fra Windows 32 API og ANSI C standardbibliotekene. Kun versjon 2 av objektbufferet og versjon 1 av sidebufferet er vedlagt. Jeg minner også om funksjonene *debug()* og *assert()* som er benyttet for å få høyere kodekvalitet. Dersom enkelte deler av koden er vanskelig å forstå, anbefales det å studere kapittel 3.

Foran hver funksjon og klasse er en kommentarblokk. Den beskriver hva funksjonen/klassen gjør og hvordan den er implementert. I tillegg beskrives hver av funksjonsargumentene. Navn som består av mer enn et ord skilles med at alle ordene etter det første ordet har stor førstebokstav. Funksjonsnavn, klassenavn og konstanter har stor førstebokstav, mens variabelnavn har liten førstebokstav. De fleste navnene er lange, beskrivende og norske, noe som medfører at det blir enklere å lese koden samt at behovet for kommentering reduseres. 3 tegn er brukt som innrykksinkrement.

C.1 main.cpp

```
// main.cpp
// Hovedprogrammet for buffersimulatoren.

#include "systemV0.h"
#include "databaseV0.h"
#include "sidebufferV1.h"
#include "objektbufferV2.h"
#include "sekvensiellfilV0.h"
#include "hashtabellV0.h"
#include <limits.h>
#include <fstream.h>
#include <stdlib.h>

// Noen globale objekter
Database *database;
Sidebuffer *sidebuffer;  // sidebuffer bruker database
SekvensiellFil *datafil;  // sekvensiellfil bruker sidebuffer
Objektbuffer *objektbuffer;
Hashtabell *katalog;  // Inneholder mappingen mellom logisk og fysisk OID
fstream simuleringsdata;  // Skriver ytelsesresultatene til denne fila

void TestDatabase();
void TestSidebuffer();
void TestObjektbuffer();
void TestSekvensiellFil();
void Objektoperasjoner1(int);
void Objektoperasjoner2(int);

// main
// Åpner fila "simuleringsdata" og kaller deretter en av benchmarkene eller
// en av klasse testfunksjonene. Bufferstørrelsen, i antall rammer, gis som
// et kommandolinje argument. Avslutter med å lukke fila.
//
// "argc" er antall kommandolinje argumenter som programmet ble startet med
// "argv" er en peker til en tabell med strenger. Hver streng inneholder
//        et argument.

void main(int argc, char *argv[])
{
   int oppstartstid = GetTickCount();

   //srand(oppstartstid);
   simuleringsdata.open("simuleringsdata.sdv", ios::out | ios::trunc);
   assert(!simuleringsdata.fail());
   printf("Starter naa testprogram\n");
   //TestDatabase();
   //TestSidebuffer();
   //TestObjektbuffer();
   //TestSekvensiellFil();
   Objektoperasjoner1(atoi(argv[1]));
   //Objektoperasjoner2(atoi(argv[1]));

   simuleringsdata.close();
   //Beep(5000, 1000);
}

// Testfunksjonene nedenfor har hatt varierende innhold, og de er ikke
// ment å være omfattende.

// TestDatabase
// Tester database klassen

void TestDatabase()
{
   int i;
   char *data  =  (char *) calloc(Sidestoerrelse, 1);
   char data2[Sidestoerrelse];

   database = new Database("testDB", 100);
   //database->VelgModus(Simuler);
   for (i = 0; i < 100; i++) {
      database->SkrivSide(i, data);
   }
   for (i = 0; i < Sidestoerrelse; i++) {
      data[i] = 'a';
   }
```

```
   for (i = 0; i < 10; i++) {
      database->SkrivSide(rand() % 100, data);
      database->LesSide(rand() % 100, data2);
   }
}

// TestSidebuffer
// Tester sidebuffer klassen

void TestSidebuffer()
{
   struct {
      int nr;
      int skitten;
   } side [300];

   database = new Database("testDB", 300);
   sidebuffer = new Sidebuffer(108);

   for (int i = 0; i < 75; i++) {
      side[i].nr = rand() % 15;
      side[i].skitten = rand() % 2;
   }

   printf("Sidebuffer start\n");
   int tid = GetTickCount();
   for (i = 0; i < 75; i++) {
      sidebuffer->HentSide(side[i].nr);
      sidebuffer->FrigiSide(side[i].nr, side[i].skitten);
   }
   tid = GetTickCount() - tid;
   printf("Kjoeretid %d ms\n", tid);
}

// TestObjektbuffer
// Tester objektbuffer klassen

void TestObjektbuffer()
{
   int i, tid, lengde;
   LogOID logOID;
   FysOID fysOID;
   int logiskeReferanser, fysiskeReferanser;
   char *foo = "Hallo verden! Hello world!";
   char baz[200];


   fysOID.sidenr = 0;
   fysOID.offset = 0;
   logOID = 0;

   //delete sidebuffer;
   //sidebuffer = new Sidebuffer(27);
   database = new Database("testDB", 3000);
   objektbuffer = new Objektbuffer(1998);
   datafil = new SekvensiellFil();

   Hashtabell *hashtabell = new Hashtabell();

   fysOID = datafil->SettInnObjekt(baz, 80);
   hashtabell->SettInnElement(0, fysOID, 80);
   fysOID = datafil->SettInnObjekt(baz, 32);
   hashtabell->SettInnElement(1, fysOID, 32);
   fysOID = datafil->SettInnObjekt(baz, 56);
   hashtabell->SettInnElement(2, fysOID, 56);
   fysOID = datafil->SettInnObjekt(baz, 32);
   hashtabell->SettInnElement(3, fysOID, 32);
   fysOID = datafil->SettInnObjekt(baz, 100);
   hashtabell->SettInnElement(4, fysOID, 100);

   hashtabell->FinnElement(0, fysOID, lengde);
   objektbuffer->HentObjekt(fysOID, lengde);
   objektbuffer->FrigiObjekt(fysOID, FALSK);

   hashtabell->FinnElement(1, fysOID, lengde);
   objektbuffer->HentObjekt(fysOID, lengde);
   objektbuffer->FrigiObjekt(fysOID, FALSK);

   hashtabell->FinnElement(2, fysOID, lengde);
   objektbuffer->HentObjekt(fysOID, lengde);
   objektbuffer->FrigiObjekt(fysOID, FALSK);

   hashtabell->FinnElement(3, fysOID, lengde);
   objektbuffer->HentObjekt(fysOID, lengde);
   objektbuffer->FrigiObjekt(fysOID, FALSK);

   int lengdetab[10] = {112,112,121,83,101,124,131,128,119,114};
   int logoidtab[10] = {0,1,4,1,7,8, 5, 3};

   for(i = 0; i < 50; i++) {
      lengde = rand() % 100 + 50;
      //lengde = lengdetab[i];
      fysOID = datafil->SettInnObjekt(baz, lengde);
      hashtabell->SettInnElement(i, fysOID, lengde);
   }

   printf("Henter tilfeldige deler fra databasen...\n");
   tid = GetTickCount();
   int k = 0;
   for(i = 0; i < 150; i++) {
      //if (rand() % 10 == 9)
      logOID = rand() % 50;
      //logOID = logoidtab[i];
      //else
      // logOID = k % 10;
      assert(
         hashtabell->FinnElement(logOID, fysOID, lengde));
       objektbuffer->HentObjekt(fysOID, lengde);
      objektbuffer->FrigiObjekt(fysOID, FALSK);
      //objektbuffer->SkrivUtMinnekart();
      if (i % 150 == 0) {
         objektbuffer->SkrivUtMinnekart();
         getchar();
      }
      k += 6;
   }
   tid = GetTickCount() - tid;

   objektbuffer->SkrivUtMinnekart();

   printf("             Kjoeretid: %d ms\n", tid);

   Statistikk stat  = objektbuffer->HentStatistikk();
   printf("Objektbuffer: Antall forespoersler: %d, Antall miss: %d, "
    "Objektmiss(%%): %.2f\n", stat.antallForespoersler, fysiskeReferanser,
                        (double)fysiskeReferanser / logiskeReferanser * 100);

   sidebuffer->HentStatistikk(logiskeReferanser, fysiskeReferanser);
   printf("Sidebuffer: Logiske referanser: %d, Fysiske referanser: %d, "
           "Sidemiss(%%): %.2f\n", logiskeReferanser, fysiskeReferanser,
                        (double)fysiskeReferanser / logiskeReferanser * 100);

   fysOID = datafil->SettInnObjekt(foo, 16);
   sidebuffer->Flush();
   char *str = objektbuffer->HentObjekt(fysOID, 16);
   objektbuffer->FrigiObjekt(fysOID, SANN);
```

```
   fysOID = datafil->SettInnObjekt(foo, 16);
   str = objektbuffer->HentObjekt(fysOID, 16);
   objektbuffer->FrigiObjekt(fysOID, FALSK);
   printf("%s\n", str);

   exit(0);
}

// TestSekvensiellFil
// Tester sekvensiell fil klassen

void TestSekvensiellFil()
{
   int i;
   char data[4096];
   char *foo = "Hallo verden! Hello world!";
   char baz[200];

   FysOID fysOID = datafil->SettInnObjekt(foo, 14);
   datafil->HentObjekt(baz, 14, fysOID);

   printf("%s\n", baz);

   for (i = 0; i < 170; i++)
      fysOID = datafil->SettInnObjekt(foo, 14);

   for (i = 0; i < 40; i++)
      datafil->SettInnObjekt(data, (rand() % 820) + 1);

   for (i = 0; i < 50; i++)
      database->SkrivSide(i, data);
}
```

C.2 OO1V0.cpp

```
// OO1V0.cpp
// Funksjoner og datastrukturer for implementasjon av OO1 benchmarken. Først
// bygges en rettet graf med deler og koblinger mellom dem. Ytelsen måles
// ved å sette inn nye deler, hente deler og følge koblinger mellom delene.

#include <time.h>
#include <fstream.h>
#include "systemV0.h"
#include "databaseV0.h"
#include "sidebufferV1.h"
#include "objektbufferV2.h"
#include "sekvensiellfilV0.h"
#include "hashtabellV0.h"

int antallDeler;  // Angir hvor mange deler som ligger i databasen

const int AntallKoblingerFraDel = 3;
const int MaksTreDybde = 8;

extern fstream simuleringsdata;  // Skriver ytelsesresultatene til denne fila
extern Database *database;
extern Sidebuffer *sidebuffer;
extern SekvensiellFil *datafil;
extern Objektbuffer *objektbuffer;
extern Hashtabell *katalog;  // Viser hvilke objekter som ligger i databasen,
                             // og hvor de ligger
```



```
// Databasen vil bestå av to typer objekter, deler og koblinger.

class Del {
public:
   LogOID logOID;
   char type[10];
   int x,y;
   struct tm bygget;
   char foo[76];  // Lagt inn for å redusere virkningen av intern fragmentering
   FysOID foersteKoblingFraDel;
   FysOID foersteKoblingTilDel;
} del;

// En kobling går mellom to deler og har retning.

class Kobling {
public:
   FysOID fraDel;
   FysOID tilDel;
   char type[10];
   char foo[48];  // Lagt inn for å redusere virkningen av intern fragmentering
   int lengde;
   FysOID nesteKoblingFraDel;
   FysOID nesteKoblingTilDel;
} kobling;

void Initialiser(int antallNyeDeler);
int GjennomgaaFramover(FysOID delFysOID, int dybde);
void SettInn(int antallNyeDeler);
void SkrivUtDel(Del del, char debugflagg);
LogOID SkrivUtKobling(Kobling kobling, char debugflagg);
void Sammenkoble(FysOID fraDelFysOID, FysOID tilDelFysOID,
                                          FysOID koblingFysOID);
void SettInnKoblinger(int antallNyeDeler);

// Objektoperasjoner1
// Initialiserer testmiljøet, kjører en spesifikk benchmarkmåling og skriver
// resultatet til ei semikolondelt fil.

// "antallRammerKommandolinje" angir hvor stort bufferet skal være under
//                             benchmark kjøringen

void Objektoperasjoner1(int antallRammerKommandolinje)
{
   int tid, sumTid, lengde;
   FysOID fysOID;
   int antallSiderLestKald, antallSiderSkrevetKald;
   int antallSiderLestVarm, antallSiderSkrevetVarm;
   int sumAntallSiderLest, sumAntallSiderSkrevet;

   database = new Database("foo.db", 2700);
   sidebuffer = new Sidebuffer(2700);
   datafil = new SekvensiellFil();
   katalog = new Hashtabell();

   srand(0);  // Gir den samme sekvensen av tilfeldige tall for hver kjøring.
   printf("Initialiserer databasen med deler og koblinger...\n");
   tid = GetTickCount();
   Initialiser(20000);  // Initialiser databasen med deler og koblinger
   tid = GetTickCount() - tid;
   printf("             Kjoeretid: %d ms\n", tid);

   printf("Flusher sidebuffer...\n");
   tid = GetTickCount();
   sidebuffer->Flush();
   tid = GetTickCount() - tid;
   printf("             Kjoeretid: %d ms\n", tid);
```

```
//database->SettAksesstid(20);
datafil->VelgBuffertype('o');
objektbuffer = new Objektbuffer(81);

double slaaOppTid, slaaOppTidKald, slaaOppTidVarm = 0;
double gjennomgaaFramoverTid, gjennomgaaFramoverTidKald,
       gjennomgaaFramoverTidVarm = 0;
double gjennomgaaBakoverTid, gjennomgaaBakoverTidKald,
       gjennomgaaBakoverTidVarm = 0;
double settInnTid, settInnTidKald, settInnTidVarm;

for (int antallRammer = antallRammerKommandolinje;
    antallRammer > antallRammerKommandolinje - 1 ; antallRammer -= 216) {
   //delete sidebuffer;
   //sidebuffer = new Sidebuffer(antallRammer);
   delete objektbuffer;
   objektbuffer = new Objektbuffer(antallRammer);

   sumAntallSiderLest = 0;
   sumAntallSiderSkrevet = 0;
   printf("----------------Antall rammer %d:\n", antallRammer);
   for (int i = 0; i < 50; i++) {
      database->NullstillStatistikk();
      sidebuffer->NullstillStatistikk();
      objektbuffer->NullstillStatistikk();

      sumTid = 0;
      putchar('\n');
      printf("Henter tilfeldige deler fra databasen...\n");
      tid = GetTickCount();
      //SlaaOpp(1000);
      tid = GetTickCount() - tid;
      printf("           Kjoeretid: %d ms\n", tid);
      sumTid += tid;
      slaaOppTid = tid;

      printf("Går gjennom framover...\n");
      tid = GetTickCount();
      LogOID delnr = rand() % antallDeler;
      katalog->FinnElement(delnr, fysOID, lengde);
      int antallDelerLest;
      antallDelerLest = GjennomgaaFramover(fysOID, 0);
      tid = GetTickCount() - tid;
      printf("          Kjoeretid: %d ms, Antall deler lest %d\n",
                                            tid, antallDelerLest);
      sumTid += tid;
      gjennomgaaFramoverTid = tid;

      printf("Går gjennom bakover...\n");
      tid = GetTickCount();
      //antallDelerLest = GjennomgaaBakover(fysOID, 0);
      tid = GetTickCount() - tid;
      printf("          Kjoeretid: %d ms, Antall deler lest %d\n",
                                            tid, antallDelerLest);
      sumTid += tid;
      gjennomgaaBakoverTid = tid;

      printf("Setter inn...\n");
      tid = GetTickCount();
      //SettInn(100);
      //sidebuffer->Flush();
      //objektbuffer->Flush();
      tid = GetTickCount() - tid;
      printf("          Kjoeretid: %d ms\n", tid);
      sumTid += tid;
      settInnTid = tid;

      database->SkrivUtStatistikk();
      sidebuffer->SkrivUtStatistikk();
      objektbuffer->SkrivUtStatistikk();

      printf("      Den summerte tiden for den %d. gangen: %d\n", i + 1,
                                                                  sumTid);
      //objektbuffer->SkrivUtMinnekart();
      int antallSiderLest, antallSiderSkrevet, foo;
      database->HentStatistikk(antallSiderLest, antallSiderSkrevet, foo);
      if (i == 0) {
         antallSiderLestKald = antallSiderLest;
         antallSiderSkrevetKald = antallSiderSkrevet;
         slaaOppTidKald = slaaOppTid / 1000;
         gjennomgaaFramoverTidKald = gjennomgaaFramoverTid / 1000;
         gjennomgaaBakoverTidKald = gjennomgaaBakoverTid / 1000;
         //settInnTidKald = settInnTid / 1000;
      }
      if ( i > 39 ) {
        sumAntallSiderLest += antallSiderLest;
        sumAntallSiderSkrevet += antallSiderSkrevet;
        slaaOppTidVarm += slaaOppTid;
        gjennomgaaFramoverTidVarm += gjennomgaaFramoverTid;
        gjennomgaaBakoverTidVarm += gjennomgaaBakoverTid;
        //settInnTidVarm += settInnTid;
      }
   }
   slaaOppTidVarm /= 10000;
   gjennomgaaFramoverTidVarm /= 10000;
   gjennomgaaBakoverTidVarm /= 10000;
   antallSiderLestVarm = sumAntallSiderLest / 10;
   antallSiderSkrevetVarm = sumAntallSiderSkrevet / 10;
   //settInnTidVarm /= 5000;

   // skriv resultatene til ei fil
   simuleringsdata << antallRammer << ';' << slaaOppTidKald << ';'
   << gjennomgaaFramoverTidKald << ';' << gjennomgaaBakoverTidKald << ';'
   << settInnTidKald << ';' << slaaOppTidVarm << ';'
   << gjennomgaaFramoverTidVarm << ';' << gjennomgaaBakoverTidVarm << ';'
   << settInnTidVarm << ';'
   << antallSiderLestKald << ';' << antallSiderSkrevetKald << ';'
   << antallSiderLestVarm << ';' << antallSiderSkrevetVarm << ';'
   << "; \n";
  }
  printf("Flusher sidebuffer...\n");
  tid = GetTickCount();
  sidebuffer->Flush();
  tid = GetTickCount() - tid;
  printf("   Kjoeretid: %d ms\n", tid);
}


// Initialiser
// Oppretter katalogen og setter inn et initielt antall deler
// med koblinger i databasen.
//
// "antallNyeDeler" angir hvor mange nye deler som skal legges inn i databasen

void Initialiser(int antallNyeDeler)
{
   antallDeler = 0;
   SettInn(antallNyeDeler);
}

// SettInn
// Setter inn deler i databasen med koblinger til andre deler.
//
// "antallNyeDeler" angir hvor mange nye deler som skal legges inn i databasen

void SettInn(int antallNyeDeler)
```

```
{
   debug('I', "OO1: Del er %d bytes og kobling er %d bytes\n",
                                        sizeof(del), sizeof(kobling));
   // Oppretting av deler
   for(int i = 0; i < antallNyeDeler; i++) {
      // Initialiser delen, ikke så veldig viktig for ytelsen.
      del.logOID = i + antallDeler;
      del.foersteKoblingFraDel.sidenr = -1;
      del.foersteKoblingFraDel.offset = -1;
      del.foersteKoblingTilDel.sidenr = -1;
      del.foersteKoblingTilDel.offset = -1;
      del.type[0] = 'a';
      del.x = 286331153;
      del.y = 572662306;
      // del.bygget = ...
      debug('I', "OO1: Setter inn del %d i databasen\n", del.logOID);
      FysOID fysOID = datafil->SettInnObjekt((char *)&del, sizeof(del));
      katalog->SettInnElement(del.logOID, fysOID, sizeof(del));
   }
   antallDeler += antallNyeDeler;
   SettInnKoblinger(antallNyeDeler);
}

// SettInnKoblinger
// Oppretter koblinger mellom delene. Hver del har tre koblinger til andre
// deler.
//
// "antallNyeDeler" angir hvor mange deler som ble satt inn i databasen

void SettInnKoblinger(int antallNyeDeler)
{
   LogOID fraDel, tilDel;
   FysOID fraDelFysOID, tilDelFysOID , koblingFysOID;
   int lengde;

   // Oppretting av koblinger mellom delene
   for(int i = 0; i < antallNyeDeler; i++) {
      fraDel = i + antallDeler - antallNyeDeler;
      assert(
         katalog->FinnElement(fraDel, fraDelFysOID, lengde));
      kobling.fraDel = fraDelFysOID;
      for(int j = 0; j < AntallKoblingerFraDel; j++) {
         if ((rand() % 10) > 1) {
             // MERK: Denne må endres når lokaliteten skal forandres.
             // Nå brukes objektlokalitet

             // sidelokalitet
             // 90% av tiden lages koblinger til de 1% nærmeste delene
             //tilDel = fraDel + (rand() % (antallDeler/100)) - antallDeler / 200;
             //if (tilDel < (antallDeler / 200) ) {
             //tilDel += antallDeler / 200;
            //}
             //if (tilDel > (antallDeler - antallDeler / 200) ) {
             //tilDel -= antallDeler / 200;
            //}

             // objektlokalitet med 80/20
            tilDel = (rand() % 4000);
         } else {
            tilDel = (rand() % (antallDeler - 4000)) + 4000;
         }
         assert(
            katalog->FinnElement(tilDel, tilDelFysOID, lengde));
         // Initialiser koblingen
         kobling.tilDel = tilDelFysOID;
         kobling.nesteKoblingFraDel.sidenr = -1;
         kobling.nesteKoblingFraDel.offset = -1;
         kobling.nesteKoblingTilDel.sidenr = -1;
         kobling.nesteKoblingTilDel.offset = -1;
         //kobling.type = ..
         kobling.lengde = 10;

         debug('t', "OO1: Setter inn en kobling mellom del %d og del %d\n",
                                                              fraDel, tilDel);
         koblingFysOID =
            datafil->SettInnObjekt((char *)&kobling, sizeof(kobling));
         Sammenkoble(fraDelFysOID, tilDelFysOID, koblingFysOID);
      }
   }
}

// Sammenkoble
// Oppdaterer de sirkulære listene av pekere hos de to delene.
//
// "fraDelFysOID" fysisk OID til den delen som koblingen starter på
// "tilDelFysOID" fysisk OID til den delen som koblingen slutter på
// "koblingFysOID" fysisk OID til koblingen

void Sammenkoble(FysOID fraDelFysOID, FysOID tilDelFysOID, FysOID koblingFysOID)
{
   // Hent inn delen som koblingen starter på
   datafil->HentObjekt((char *)&del, sizeof(del), fraDelFysOID);
   if (del.foersteKoblingFraDel.sidenr == -1) { // Ingen koblinger fra delen
      debug('S', "Sammenkoble: Del %d oppdateres med foersteKoblingFraDel "
             "= (%d, %d)\n", del.logOID, koblingFysOID.sidenr,
                                                   koblingFysOID.offset);
      del.foersteKoblingFraDel = koblingFysOID;
      datafil->OppdaterObjekt((char *)&del, sizeof(del), fraDelFysOID);
   }
   else {  // Delen har allerede koblinger
      // Henter første element i lista av koblinger som går fra aktuell del
      datafil->HentObjekt((char *)&kobling, sizeof(kobling),
                                      del.foersteKoblingFraDel );
      // Fysisk OID på den koblingen som er lest inn i hovedminnet
      FysOID naavaerendeKoblingFysOID = del.foersteKoblingFraDel;
      // Finner siste element
      while (!(kobling.nesteKoblingFraDel.sidenr == -1)) {
         naavaerendeKoblingFysOID = kobling.nesteKoblingFraDel;
         datafil->HentObjekt((char *)&kobling, sizeof(kobling),
                                        kobling.nesteKoblingFraDel);
      }
      debug('S', "Sammenkoble: Oppdaterer kobling (%d, %d) med "
             "nesteKoblingFraDel = (%d, %d)\n", naavaerendeKoblingFysOID.sidenr,
             naavaerendeKoblingFysOID.offset, koblingFysOID.sidenr,
                                                   koblingFysOID.offset);
      // Oppdaterer siste kobling i lista til å peke på den nye koblingen
      kobling.nesteKoblingFraDel = koblingFysOID;
      datafil->OppdaterObjekt((char *)&kobling, sizeof(kobling),
                                             naavaerendeKoblingFysOID);
   }
   // Hent inn delen som koblingen slutter på
   datafil->HentObjekt((char *)&del, sizeof(del), tilDelFysOID);
   if (del.foersteKoblingTilDel.sidenr == -1) { // Ingen koblinger til delen
      debug('S', "Sammenkoble: Del %d oppdateres med foersteKoblingTilDel "
             "= (%d, %d)\n", del.logOID, koblingFysOID.sidenr,
                                                   koblingFysOID.offset);
      del.foersteKoblingTilDel = koblingFysOID;
      datafil->OppdaterObjekt((char *)&del, sizeof(del), tilDelFysOID);
   }
   else {  // Delen har allerede koblinger til seg
      // Henter første element i lista av koblinger som går til aktuell del
      datafil->HentObjekt((char *)&kobling, sizeof(kobling),
                                      del.foersteKoblingTilDel );
      // Fysisk OID på den koblingen som er lest inn i hovedminnet
      FysOID naavaerendeKoblingFysOID = del.foersteKoblingTilDel;
      // Finner siste element
```

```
      while (!(kobling.nesteKoblingTilDel.sidenr == -1)) {
         naavaerendeKoblingFysOID = kobling.nesteKoblingTilDel;
         datafil->HentObjekt((char *)&kobling, sizeof(kobling),
                                    kobling.nesteKoblingTilDel);
      }
      debug('S', "Sammenkoble: Oppdaterer kobling (%d, %d) med "
        "nesteKoblingTilDel = (%d, %d)\n", naavaerendeKoblingFysOID.sidenr,
             naavaerendeKoblingFysOID.offset, koblingFysOID.sidenr,
                                                  koblingFysOID.offset);
      // Oppdaterer siste kobling i lista til å peke på den nye koblingen
      kobling.nesteKoblingTilDel = koblingFysOID;
      datafil->OppdaterObjekt((char *)&kobling, sizeof(kobling),
                                               naavaerendeKoblingFysOID);
   }
}

// SlaaOpp
// Henter tilfeldige deler fra databasen
//
// "antallOppslag" angir hvor mange tilfeldige deler som skal hentes

void SlaaOpp(int antallOppslag)
{
   int i, lengde;
   LogOID logOID;
   FysOID fysOID;

   debug('O', "Henter tilfeldige deler fra databasen\n");
   for(i = 0; i < antallOppslag; i++) {
      logOID = (rand() % antallDeler); // antallDeler;
      katalog->FinnElement(logOID, fysOID, lengde);
      datafil->HentObjekt((char *)&del, sizeof(del), fysOID);
      SkrivUtDel(del, 'O');
   }
}

// GjennomgaaFramover
// Følger koblinger fra en del til andre deler helt ned til en dybde på
// "MaksTreDybde".
//
// "delFysOID" er fysisk OID på den delen som gjennomgangen skal starte med
// "dybde" er hvor dypt søket er nå

int GjennomgaaFramover(FysOID delFysOID, int dybde)
{
   FysOID koe[AntallKoblingerFraDel];
   int hale = 0;
   static int antallDelerLest;

   if (dybde == 0) {
      antallDelerLest = 0;
   }
   if (dybde >= MaksTreDybde)
      ;  // Stopp gjennomgangen
   else {
      //katalog->FinnElement(delLogOID, delFysOID, lengde);
      datafil->HentObjekt((char *)&del, sizeof(del), delFysOID);
      if ((rand() % 10) > 7) {  // kan også legge inn oppdatering av delen
      // datafil->OppdaterObjekt((char *)&del, sizeof(del), delFysOID);
      }
      antallDelerLest++;
      SkrivUtDel(del, 'G');
      debug('G', "Dybde: %d\n", dybde);
      if (del.foersteKoblingFraDel.sidenr == -1) {  // Ingen koblinger fra delen
         debug('G', "Delen har ingen koblinger til andre deler\n");
         assert(del.foersteKoblingFraDel.sidenr != -1);
      } else {
         datafil->HentObjekt((char *)&kobling, sizeof(kobling),  // Hent første
                                          del.foersteKoblingFraDel );  // kobling
         koe[hale++] = kobling.tilDel;
         // SkrivUtKobling(kobling, 'G');
         while (kobling.nesteKoblingFraDel.sidenr != -1) {
            datafil->HentObjekt((char *)&kobling, sizeof(kobling),  // Hent neste
                                          kobling.nesteKoblingFraDel); // kobling
            koe[hale++] = kobling.tilDel;
            //SkrivUtKobling(kobling, 'G');
         }
         for (int i = 0 ; i < hale; i++)
            GjennomgaaFramover(koe[i], dybde + 1);
      }
   }
   return antallDelerLest;
}

// GjennomgaaBakover
// Følger koblingene mellom delene bakover ned til "MaksTreDybde".
//
// "delFysOID" er fysisk OID på den delen som gjennomgangen skal starte med
// "dybde" er dybden akkurat nå

int GjennomgaaBakover(FysOID delFysOID, int dybde)
{
   FysOID koe[AntallKoblingerFraDel * 5];
   int hale = 0;
   static int antallDelerLest;

   if (dybde == 0) {
      antallDelerLest = 0;
   }
   if (dybde >= MaksTreDybde)
      ;  // Stopp gjennomgangen
   else {
      //katalog->FinnElement(delLogOID, delFysOID, lengde);
      datafil->HentObjekt((char *)&del, sizeof(del), delFysOID);
      if ((rand() % 10) > 7) {
      // datafil->OppdaterObjekt((char *)&del, sizeof(del), delFysOID);
      }
      antallDelerLest++;
      SkrivUtDel(del, 'G');
      debug('G', "Dybde: %d\n", dybde);
      if (del.foersteKoblingTilDel.sidenr == -1) {  // Ingen koblinger fra delen
         debug('G', "Delen har ingen koblinger til seg\n");
         //assert(del.foersteKoblingTilDel.sidenr != -1);
      } else {
         datafil->HentObjekt((char *)&kobling, sizeof(kobling),  // Hent første
                                          del.foersteKoblingTilDel );  // kobling
         koe[hale++] = kobling.fraDel;
         // SkrivUtKobling(kobling, 'G');
         while (kobling.nesteKoblingTilDel.sidenr != -1) {
            datafil->HentObjekt((char *)&kobling, sizeof(kobling),  // Hent neste
                                          kobling.nesteKoblingTilDel); // kobling
            koe[hale++] = kobling.fraDel;
            //SkrivUtKobling(kobling, 'G');
         }
         for (int i = 0 ; i < hale; i++)
            GjennomgaaBakover(koe[i], dybde + 1);
      }
   }
   return antallDelerLest;
}

// SkrivUtDel
// Skriver ut en del
//
// "del" er delen som skal skrives ut
// "debugflagg" er hvilket debugflagg som skal være satt for at delen skal
```

```
//              skrives ut

void SkrivUtDel(Del del, char debugflagg)
{
   debug(debugflagg, "Del: logOID = %d, foersteKoblingFraDel(%d, %d), "
         "foersteKoblingTilDel(%d, %d)\n", del.logOID,
           del.foersteKoblingFraDel.sidenr, del.foersteKoblingFraDel.offset,
           del.foersteKoblingTilDel.sidenr, del.foersteKoblingTilDel.offset);
}

// SkrivUtKobling
// Skriver ut en kobling til skjermen
//
// "kobling" er koblingen som skal skrives ut
// "debugflagg" er hvilket debugflagg som skal være satt for at koblingen skal
//              skrives ut

LogOID SkrivUtKobling(Kobling kobling, char debugflagg)
{
   Del fraDel, tilDel;

   datafil->HentObjekt((char *)&fraDel, sizeof(fraDel), kobling.fraDel);
   datafil->HentObjekt((char *)&tilDel, sizeof(tilDel), kobling.tilDel);
   debug(debugflagg, "Kobling: Fra del %d til del %d\n",
                              fraDel.logOID, tilDel.logOID);
   return tilDel.logOID;
}
```

C.3 OO2.cpp

```
// OO2.cpp
// Funksjoner og datastrukturer for OO2 benchmarken. Den henter
// "antallObjekter" fra databasen som består av 65535 objekter på mellom
// 50 og 300 byte, med et snitt på 175 byte. Aksessene følger 90/10 regelen,
// det vil si at 90 % av aksessene går til 10 % av databasen.

#include <time.h>
#include <fstream.h>
#include "systemV0.h"
#include "databaseV0.h"
#include "sidebufferV1.h"
#include "objektbufferV2.h"
#include "sekvensiellfilV0.h"
#include "hashtabellV0.h"

int antallObjekter;  // Angir hvor mange objekter som ligger i databasen

extern fstream simuleringsdata;  // Fil som simuleringsresultatene skrives til
extern Database *database;
extern Sidebuffer *sidebuffer;  // sidebuffer bruker database
extern SekvensiellFil *datafil;  // sekvensiell fil bruker sidebuffer
extern Objektbuffer *objektbuffer;
extern Hashtabell *katalog;  // Oversik over hvilke objekter som ligger i
                             // i databasen og hvor de ligger.

char data[Sidestoerrelse];

void InitialiserOO2(int antallNyeObjekter);
void HentObjekter(int antallOppslag, int modifikasjonsfaktor,
                                     int sammenklyngingsfaktor);
void SettInnObjekter(int antallNyeObjekter);

// Objektoperasjoner2
```



```
// Initialiserer testmiljøet, kjører en spesifikk benchmarkmåling og skriver
// resultatet til fila "datafil".
//
// "antallRammerKommandolinje" er størrelsen på bufferet gitt som
//                             kommandolinje argument.

void Objektoperasjoner2(int antallRammerKommandolinje)
{
   int antallSiderLestKald, antallSiderSkrevetKald;
   int antallSiderLestVarm, antallSiderSkrevetVarm;
   int sumAntallSiderLest, sumAntallSiderSkrevet;

   srand(0);  // Gir samme sekvens av tilfeldige tall
   printf("Initialiserer databasen med objekter av forskjellig "
                                                 "stoerrelse...\n");
   database = new Database("foo.db", 14985);
   database->VelgModus(Simuler);  // Går raskere da
   sidebuffer = new Sidebuffer(2916);  // Bruker sidebuffer når databasen
                                       // skal initialiseres
   datafil = new SekvensiellFil();
   katalog = new Hashtabell();

   int tid = GetTickCount();
   InitialiserOO2(65535);
   tid = GetTickCount() - tid;
   printf("            Kjoeretid: %d ms\n", tid);

   printf("Flusher sidebuffer...\n");
   tid = GetTickCount();
   sidebuffer->Flush();
   tid = GetTickCount() - tid;
   printf("            Kjoeretid: %d ms\n", tid);

   database->SkrivUtStatistikk();
   objektbuffer = new Objektbuffer(81);

   double cpuTidVarm, sumCpuTid;
   Statistikk stat;  // For objektbuffer
   int modifikasjonsfaktor = 0;
   int sammenklyngingsfaktor = 100;

   simuleringsdata << "modifikasjonsfaktor ; ; ;"
                   << modifikasjonsfaktor << "; \n"
                   << "sammenklyngingsfaktor ; ; ;"
                   << sammenklyngingsfaktor << "; \n" ;

   for (int antallRammer = antallRammerKommandolinje;
        antallRammer < antallRammerKommandolinje + 1 ; antallRammer += 81) {
      //delete sidebuffer;
      //sidebuffer = new Sidebuffer(antallRammer);
      delete objektbuffer;
      objektbuffer = new Objektbuffer(antallRammer);
      datafil->VelgBuffertype('o');

      //database->SettAksesstid(20);
      //objektbuffer->VelgModus(Skriveoptimalisert);

      sumAntallSiderLest = 0;
      sumAntallSiderSkrevet = 0;
      sumCpuTid = 0;
      printf("Antall rammer: %d\n", antallRammer);
      for (int j = 0; j < 100; j++) {
         database->NullstillStatistikk();
         sidebuffer->NullstillStatistikk();
         objektbuffer->NullstillStatistikk();

         int antallObjekter = 4000;
         printf("\nHenter %d tilfeldige objekter fra databasen...\n",
```

```
                                                        antallObjekter);
        tid = GetTickCount();
        HentObjekter(antallObjekter,
                              modifikasjonsfaktor, sammenklyngingsfaktor);
        tid = GetTickCount() - tid;
        objektbuffer->Flush();
        //sidebuffer->Flush();
        //objektbuffer->SkrivUtMinnekart();

        int antallSiderLest, antallSiderSkrevet, foo;
        database->HentStatistikk(antallSiderLest, antallSiderSkrevet, foo);

        double cpuTid, ioTid;
        cpuTid = tid / 1000.0;
        ioTid = 25 * (antallSiderLest + antallSiderSkrevet) / 1000.0;

        printf("Kjoeretid for den %d. gangen: cpu: %.4f s,"
               " io: %.4f s, sum: %.4f s\n", j + 1, cpuTid, ioTid,
                                                    cpuTid + ioTid);
        objektbuffer->SkrivUtStatistikk();
        sidebuffer->SkrivUtStatistikk();
        database->SkrivUtStatistikk();
        if (j == 0) {
           antallSiderLestKald = antallSiderLest;
           antallSiderSkrevetKald = antallSiderSkrevet;
        }
        if (j > 89) {
           sumAntallSiderLest += antallSiderLest;
           sumAntallSiderSkrevet += antallSiderSkrevet;
           sumCpuTid += cpuTid;
        }
        stat = objektbuffer->HentStatistikk();
        simuleringsdata
           << j + 1 << ';' << stat.antallMiss << ';'
           << antallSiderLest << ';' << antallSiderSkrevet << ';'
           << stat.minneutnyttelse << ';' << stat.antallLedigeBlokker << ';'
           << stat.antallObjekter << ';' << cpuTid << ';' << ioTid << '\n' ;

     }
     antallSiderLestVarm = sumAntallSiderLest / 10;
     antallSiderSkrevetVarm = sumAntallSiderSkrevet / 10;
     cpuTidVarm = sumCpuTid / 10;
     simuleringsdata << antallRammer << ';'
              << antallSiderLestKald << ';' << antallSiderSkrevetKald << ';'
              << antallSiderLestVarm << ';' << antallSiderSkrevetVarm << ';'
              << cpuTidVarm << ';' << '\n' ;
   }
   //objektbuffer->SkrivUtMinnekart();
}


// InitialiserOO2
// Oppretter katalogen og setter inn et initielt antall objekter
//
// "antallNyeObjekter" er hvor mange objekter som skal legges inn i databasen

void InitialiserOO2(int antallNyeObjekter)
{
   antallObjekter = 0;
   SettInnObjekter(antallNyeObjekter);
}

// SettInnObjekter
// Setter inn objekter i databasen
//
// "antallNyeObjekter" er hvor mange objekter som skal legges inn i databasen

void SettInnObjekter(int antallNyeObjekter)
{
   // Oppretting av objekter
   for(int i = 0; i < antallNyeObjekter; i++) {
      LogOID logOID = i + antallObjekter;
      debug('I', "OO2: Setter inn objekt %d i databasen\n", logOID);

      int stoerrelse = (rand() % 251) + 50;  // Gir et snitt på 175 byte
      FysOID fysOID = datafil->SettInnObjekt(data, stoerrelse);
      katalog->SettInnElement(logOID, fysOID, stoerrelse);
   }
   antallObjekter += antallNyeObjekter;
}

// HentObjekter
// Henter objekter fra databasen og oppdaterer eventuelt noen av dem.
//
// "antallOppslag" angir hvor mange objekter som skal hentes
// "modifikajonsfaktor" er hvor stor andel av aksessene som skal oppdatere
//                      objektet som hentes
// "sammenklyngingsfaktor" er hvor godt de varme objektene skal ligge fysisk
//                         sammen på disken

void HentObjekter(int antallOppslag, int modifikasjonsfaktor,
                                          int sammenklyngingsfaktor)
{
   int i, stoerrelse;
   LogOID logOID;
   FysOID fysOID;

   debug('O', "Henter tilfeldige objekter fra databasen\n");
   assert(sammenklyngingsfaktor >= 10 &&
          sammenklyngingsfaktor <= 100);
   assert(modifikasjonsfaktor >= 0 &&
          modifikasjonsfaktor <= 100);
   for(i = 0; i < antallOppslag; i++) {
      if ((rand() % 10) > 0) {
         logOID =
            ((rand() % (antallObjekter/10)) *
                  (100.0 / sammenklyngingsfaktor));
      } else {
         logOID = (rand() * 2 + (rand() % 2));
      }
      katalog->FinnElement(logOID, fysOID, stoerrelse);
      if ((rand() % 100) < modifikasjonsfaktor) {
         datafil->OppdaterObjekt(data, stoerrelse, fysOID, logOID);
      } else {
         datafil->HentObjekt(data, stoerrelse, fysOID, logOID);
      }
   }
}
```

C.4 systemV0.h

```
// systemV0.h
// Globale variabler, typer og konstanter er definert her

#ifndef SYSTEM_H
#define SYSTEM_H

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
```

```
// Definisjon av typer
typedef int LogOID;

class FysOID {
 public:
    int sidenr;
    int offset;
};

//enum bool {FALSK, SANN};
enum BufferModus {Normal, Skriveoptimalisert};
enum DatabaseModus {Ekte, Simuler};
typedef int bool;
const int FALSK = 0;
const int SANN = 1;

// Definisjon av konstanter
const int Sidestoerrelse = 4096;
const int AntallSiderPerSegment = 27;

extern void debug(char flagg, char *format, ...);

#endif  // SYSTEM_H
```

C.5 tilbehoer.cpp

```
// tilbehoer.cpp
// Rutinen "debug"

#include "systemV0.h"
#include <stdarg.h>

bool debugflagg[256];

// debug
// Skriver formatert til skjermen på sammen måte som printf, men kun
// hvis "flagg" er satt.
//
// "flagg" angir hvilket debugflagg som må være satt for at funksjonen skal
//         skrive til skjermen
// "format" er en formatstreng som består av vanlige karakterer og
//          konverteringskarakterer.
// "..." angir et varierende antall argument


void debug(char flagg, char *format, ...)
{
   debugflagg['d'] = FALSK;  // Database
   debugflagg['r'] = FALSK;  // Ramme
   debugflagg['s'] = FALSK;  // Sidebuffer
   debugflagg['f'] = FALSK;  // Sekvensiell fil
   debugflagg['t'] = FALSK;  // Testprogram
   debugflagg['l'] = FALSK;  // Liste
   debugflagg['I'] = FALSK;  // Initialiser
   debugflagg['O'] = FALSK;  // SlaaOpp
   debugflagg['S'] = FALSK;  // Sammenkoble
   debugflagg['G'] = FALSK;  // Gjennomgaa
   debugflagg['o'] = FALSK;  // Objektbuffer
   debugflagg['h'] = FALSK;  // Hashtabell

   if (debugflagg[flagg]) {
      va_list ap;
      va_start(ap, format);
      vfprintf(stdout, format, ap);
      va_end(ap);
   }
}
```


C.6 hashtabellV0.h

```
// hashtabellV0.h
// Datastrukturer for håndtering av en hashtabell. Hashtabellen brukes til
// objektkatalogen.

#ifndef HASHTABELL_H
#define HASHTABELL_H

#include "systemV0.h"

// Denne klassen definerer elementene i den lenkede lista i hashtabellen

class Element {
public:
   Element(LogOID logOID, FysOID fysOID, int stoerrelse);

   LogOID logOID;
   FysOID fysOID;
   int stoerrelse;
   Element *neste;
};

// Den følgende klassen definerer en hashtabell. Den har like mange element
// som det er sider i databasen, og hvert element er en peker til ei liste
// av "Element"-objekter. Disse objektene hasher til samme plass i tabellen
// og lenkes derfor sammen. Navnet på denne kollisjonsløsnings strategien er
// extern kjeding. Hashtabellen er ikke laget generell, men
// tilpasset til bruk som objektkatalog. Søketiden er uavhengig av antall
// objekter i datbasen, men avhengig av antall objekter per databaseside.
// Det betyr konstant søketid, O(1). Hashfunksjonen er logisk OID mod antall
// sider i databasen.

class Hashtabell {
public:
   Hashtabell();
   ~Hashtabell();

   void SettInnElement(LogOID logOID, FysOID fysOID, int stoerrelse);
            // Setter inn et nytt element i hashtabellen

   bool FinnElement(LogOID logOID, FysOID &fysOID, int &stoerrelse);
            // Finner et element i hashtabellen basert på logisk OID

   void OppdaterElement(LogOID logOID, FysOID fysOID);
            // Oppdaterer den fysiske OID'en til et objekt

private:
   int tabellStoerrelse;  // Angir hvor hvor stor tabellen skal være
   Element **tabell;  // Selve hashtabellen
};

#endif  // HASHTABELL_H
```

C.7 hashtabellV0.cpp

```
// hashtabellV0.cpp
// Rutiner for håndtering av en hashtabell

#include "hashtabellV0.h"
#include "databaseV0.h"

extern Database *database;  // definert i main.cpp

// Element::Element
// Initialiserer objektet
//
// "nyLogOID" er logisk OID til objektet
// "nyFysOID" er fysisk OID til objektet
// "nyStoerrelse" er størrelsen i byte til objektet

Element::Element(LogOID nyLogOID, FysOID nyFysOID, int nyStoerrelse)
{
   logOID = nyLogOID;
   fysOID = nyFysOID;
   stoerrelse = nyStoerrelse;
   neste = NULL;
}

// Hashtabell::Hashtabell
// Oppretter og initialiserer tabellen

Hashtabell::Hashtabell()
{
   tabellStoerrelse = database->AntallSider();
   tabell =  new Element*[tabellStoerrelse];
   assert(tabell != NULL);
   // Initialiser tabellen
   for (int i = 0; i < tabellStoerrelse; i ++)
      tabell[i] = NULL;
}

// Hashtabell::~Hashtabell
// Sletter tabellen

Hashtabell::~Hashtabell()
{
   delete [] tabell;
}

// Hashtabell::SettInnElement
// Setter inn et nytt element i hashtabellen
//
// "logOID" er logisk OID
// "fysOID" er fysisk OID
// "stoerrelse" er størrelsen i byte

void Hashtabell::SettInnElement(LogOID logOID, FysOID fysOID, int stoerrelse)
{
   debug('h', "Hashtabell: Setter inn element logOID %d, fysOID (%d, %d), "
          "stoerrelse %d\n", logOID, fysOID.sidenr, fysOID.offset, stoerrelse);
   Element *element = new Element(logOID, fysOID, stoerrelse);
   int tabellnr = logOID % tabellStoerrelse;
   // Setter inn elementet først i lista
   element->neste = tabell[tabellnr];
   tabell[tabellnr] = element;
}

// Hashtabell::FinnElement
// Søker etter et element basert på logisk OID, og returnerer fysisk OID
//
// "logOID" er logisk OID
// "fysOID" er fysisk OID, utargument
// "stoerrelse" er størrelsen i byte, utargument

bool Hashtabell::FinnElement(LogOID logOID, FysOID &fysOID, int &stoerrelse)
{
   // Ta første element fra lista
   Element *element = tabell[logOID % tabellStoerrelse];
   bool funnet = FALSK;
   while (element != NULL  && !funnet) {
      if (element->logOID == logOID) {
         fysOID = element->fysOID;
         stoerrelse = element->stoerrelse;
         funnet = SANN;
      }
      else
         element = element->neste;
   }
   if (funnet)
      debug('h', "Hashtabell: Element med logOID %d og stoerrelse %d "
                 " ble funnet\n", logOID, stoerrelse);
   else
      debug('h', "Hashtabell: Element med logOID %d ble ikke funnet\n", logOID);
   return funnet;
}

// Hashtabell::OppdaterElement
// Oppdaterer den fysiske OID'en til et element
//
// "logOID" er logisk OID
// "fysOID" er fysisk OID

void Hashtabell::OppdaterElement(LogOID logOID, FysOID fysOID)
{
   // Ta første element fra lista
   Element *element = tabell[logOID % tabellStoerrelse];
   bool funnet = FALSK;
   while (element != NULL  && !funnet) {
      if (element->logOID == logOID) {
         element->fysOID = fysOID;
         funnet = SANN;
      }
      else
         element = element->neste;
   }
   assert(funnet);
}
```

C.8 sekvensiellfilV0.h

```
// sekvensiellfilV0.h
// Datastrukturer for håndtering av sekvensielle filer

#ifndef SEKVENSIELLFIL_H
#define SEKVENSIELLFIL_H

#include "systemV0.h"

// I ei sekvensiell fil lagres objektene i den rekkefølgen som de settes
// inn. Objektene kan være av variabel størrelse og de settes inn på slutten
// av fila. Klassen tilbyr en sterkt forenklet versjon
// av denne typer filer, men tilstrekkelig til å kunne kjøre
```

```
// et OO1 testprogram. Der er ikke støtte for sletting av
// objekter i fila, og henting kan kun gjøres basert på fysisk OID.
// Klassen gjør det enklere å skrive testprogram som er uavhengige av
// hvilken buffertype som brukes.

class SekvensiellFil {
public:
   SekvensiellFil(bool laasSisteSide = FALSK);
            // Allokerer den første sida

   ~SekvensiellFil();

   FysOID SettInnObjekt(char *data, int lengde, LogOID logOID = -1);
            // Setter inn objektet på slutten av fila

   void HentObjekt(char *data, int lengde, FysOID fysOID, LogOID logOID = -1);
            // Henter objektet fra fila og kopierer det til brukerminnet

   void OppdaterObjekt(char *data, int lengde, FysOID fysOID,
                                                   LogOID logOID = -1);
            // Oppdaterer objektet basert på fysisk OID

   void VelgBuffertype(char buffertype);
           // Veksler mellom bruk av sidebuffer eller objektbuffer

private:
   int sisteSidenr; // Peker til den siste sida i fila
   int offset;     // Angir hvor mye plass som er brukt på siste sida
                   // i fila. Det innebærer også at den peker til neste
                   // ledige plass på sida.
   bool BrukSidebuffer; // Dersom denne variabelen har verdien SANN brukes
                        // sidebufferet, ellers brukes objektbufferet
   bool laasSisteSide;  // Når objektbufferet settes i skriveoptimalisert
                       // modus, låses den siste sida for å holde den i
                       // i bufferet
};

#endif  // SEKVENSIELLFIL_H
```

C.9 sekvensiellfilV0.cpp

```
// sekvensiellfilV0.cpp
// Rutiner for håndtering av ei sekvensiell fil

#include "sekvensiellfilV0.h"
#include "databaseV0.h"
#include "sidebufferV1.h"
#include "objektbufferV2.h"
#include "systemV0.h"
#include <string.h>

extern Database *database;
extern Sidebuffer *sidebuffer;
extern Objektbuffer *objektbuffer;

// SekvensiellFil::SekvensiellFil
// Allokerer første sida fra databasen
//
// "_laasSisteSide" angir om den siste sida i fila alltid skal være i
//                  sidebufferet. Brukes av det skriveoptimaliserte
//                  objektbufferet.

SekvensiellFil::SekvensiellFil(bool _laasSisteSide)
{
   int sidenr;

   assert(
      database->AllokerSide(sidenr));
   laasSisteSide = _laasSisteSide;
   if (laasSisteSide) {
      sidebuffer->HentSide(sidenr);  // Sida holdes låst helt til den er full
   }
   debug('f', "SekvensiellFil: Allokerte side %d som foerste side i fila\n",
                                                                    sidenr);
   sisteSidenr = sidenr;
   offset = 0;
   BrukSidebuffer = SANN;  // Antar bruk av sidebuffer som default
}

// SekvensiellFil::~SekvensiellFil

SekvensiellFil::~SekvensiellFil()
{

}

// SekvensiellFil::SettInnObjekt
// Sjekker om det er plass til objektet på den siste sida i fila. Dersom ikke,
// allokeres ei ny side fra databasen, sida låses i bufferet og objektet
// kopieres fra brukerminnet til sida. Sida frigis så og offset oppdateres.
// Objektet får en fysisk OID som er sidenummer og offset på sida. Den
// returneres til kallende funksjon.
//
// "data" peker på objektet som skal lagres på fila
// "lengde" er hvor mange byte stort objektet er
// "logOID" er logisk OID

FysOID SekvensiellFil::SettInnObjekt(char *data, int lengde, LogOID logOID)
{
   assert((lengde > 0) && (lengde <= Sidestoerrelse));
   if (!(lengde <= (Sidestoerrelse - offset))) {  // objektet får ikke plass på
                                                  // sida, vi må ha ei ny
      if (laasSisteSide) {
         sidebuffer->FrigiSide(sisteSidenr, SANN);
      }
      assert(
         database->AllokerSide(sisteSidenr));
      debug('f', "SekvensiellFil: Ikke plass til objektet paa sida, "
                                        "allokerte side %d\n", sisteSidenr);
      offset = 0;
      // Øker ytelsen litt ved å ikke lese ei tom side fra databasen
      sidebuffer->HentSide(sisteSidenr, SANN);
   }
   FysOID fysOID = {sisteSidenr, offset};
   debug('f', "SekvensiellFil: Et objekt med lengde %d settes inn paa "
                           "side %d offset %d\n", lengde, sisteSidenr, offset);
   char *bufferadr;
   if (BrukSidebuffer) {
      bufferadr = sidebuffer->HentSide(sisteSidenr) + offset;
      memcpy((void *)bufferadr, (void *)data, lengde);
      if (!laasSisteSide) {
         sidebuffer->FrigiSide(sisteSidenr, SANN);  // Låser opp sida
      }
   } else {  // Bruk objektbuffer
      bufferadr = objektbuffer->HentObjekt(fysOID, lengde, logOID);
      memcpy((void *)bufferadr, (void *)data, lengde);
      objektbuffer->FrigiObjekt(fysOID, SANN);
   }
   assert(offset < Sidestoerrelse);
   offset += lengde;
   return fysOID;
```

```
}

// SekvensiellFil::HentObjekt
// Objektet hentes fra fila ut fra den fysiske OID. Den gir sidenummer og
// offset på sida. Aktuell side hentes inn i sidebufferet hvor den låses.
// Objektet kopieres deretter over i brukerminne.
//
// "data" peker på et minneområde stort nok til objektdataene
// "lengde" er hvor mange byte stort objektet er
// "fysOID" er den fysiske OID'en til objektet gitt ved sidenummer og
//          plassnummer innenfor sida
// "logOID" er logisk OID

void SekvensiellFil::HentObjekt(char *data, int lengde, FysOID fysOID,
                                                                LogOID logOID)
{
   debug('f', "SekvensiellFil: Henter objekt (%d, %d)\n", fysOID.sidenr,
                                                            fysOID.offset);
   char *bufferadr;
   if (BrukSidebuffer) {
      bufferadr = sidebuffer->HentSide(fysOID.sidenr);
      memcpy((void *)data, (void *)(bufferadr + fysOID.offset), lengde);
      sidebuffer->FrigiSide(fysOID.sidenr, FALSK);
   }  else {  // Bruk objektbuffer
      bufferadr = objektbuffer->HentObjekt(fysOID, lengde, logOID);
      memcpy((void *)data, (void *)bufferadr, lengde);
      objektbuffer->FrigiObjekt(fysOID, FALSK);
   }
}

// SekvensiellFil::OppdaterObjekt
// Objektet hentes fra fila utifra den fysiske OID. Den gir sidenummer og
// offset på sida. Aktuell side hentes inn i sidebufferet hvor den låses.
// Objektet kopieres deretter over på sida, og sida frigis.
//
// "data" peker på et minneområde stort nok til objektdataene
// "lengde" er hvor mange byte stort objektet er
// "fysOID" er den fysiske OID'en til objektet gitt ved sidenummer og
//          plassnummer innenfor sida
// "logOID" er logisk OID

void SekvensiellFil::OppdaterObjekt(char *data, int lengde, FysOID fysOID,
                                                                LogOID logOID)
{
   assert((lengde > 0) && (lengde <= Sidestoerrelse));
   debug('f', "SekvensiellFil: Oppdaterer objekt side %d offset %d\n",
                                        fysOID.sidenr, fysOID.offset);
   char *bufferadr;
   if (BrukSidebuffer) {
      bufferadr = sidebuffer->HentSide(fysOID.sidenr) + fysOID.offset;
      memcpy((void *)bufferadr, (void *)data, lengde);
      sidebuffer->FrigiSide(fysOID.sidenr, SANN); // Låser opp sida
   } else {  // Bruk objektbuffer
      bufferadr = objektbuffer->HentObjekt(fysOID, lengde, logOID);
      memcpy((void *)bufferadr, (void *)data, lengde);
      objektbuffer->FrigiObjekt(fysOID, SANN);
   }
}

// SekvensiellFil::VelgBuffertype
// Velger om den sekvensielle fila skal bruke sidebuffer eller objektbuffer
//
// "buffertype" angir enten objektbuffer ved 'o', eller sidebuffer ved 's'

void SekvensiellFil::VelgBuffertype(char buffertype)
{
   assert(buffertype == 'o' || buffertype == 's');
   if (buffertype == 'o')
      BrukSidebuffer = FALSK;
   else  // buffertype == 's'
      BrukSidebuffer = SANN;
}
```


C.10 objektbufferV2.h

```
// objektbufferV1.h
// Datastrukturer for håndtering av objektbuffer. Endringen fra versjon 1 er
// at størrelsen på objektbufferet og det underliggende sidebufferet er
// dynamisk. Dette gir bedre kald ytelse. Størrelsen endres segmentvis, der
// et segment er "AntallSiderPerSegment" sider. Endringer fra versjon 0 er
// splitting av lediglista i flere lister basert på blokkstørrelsen, og
// innlegging av kontrollinformasjon i halen på ledige og reserverte blokker.

#ifndef OBJEKTBUFFER_H
#define OBJEKTBUFFER_H

#include "systemV0.h"
#include "sekvensiellfilV0.h"

const int DebugSkrivUtMinneKart = FALSK;
const int AntallLediglister = 27;

// Denne klassen brukes for å lettere kunne overføre all relevant statistikk-
// informasjon mellom funksjoner.

class Statistikk {
public:
   int antallForespoersler;
   int antallMiss;
   double minneutnyttelse;  // I prosent
   int antallLedigeBlokker;
   int antallObjekter;
   int antallEnheterReserverte;
   int antallEnheterLedige;
};

// Ei blokk er et sett av kontinuerlige minneceller. Etter at dynamisk
// minneallokering har vært i operasjon en stund, vil bufferet inndeles i
// reserverte og ledige blokker av forskjellig størrelse. De ledige blokkene
// lenkes sammen i ei liste. Burde nok kalt denne klassen "Blokkhode".

class Blokk {
public:
   Blokk();

   Blokk *neste;  // Peker til neste ledige blokk
   int antallEnheter;  // Er blokkstørrelsen i enheter
   Blokk *forrige;  // Peker til forrige ledige blokk
};

// Denne klassen definerer den kontrollinformasjonen som legges inn i halen på
// ei blokk. Hensikten med det er å kunne deallokere blokker,
// med sammenslåing av ledige naboblokker, på konstant tid. "foo" og "baz"
// hjelper til med å få den rette minnelayouten.

class Blokkhale {
public:
   int baz;
   int antallEnheter;
   char foo1, foo2, foo3;
   char ledig;
```

```
};

// Objekter av denne klassen brukes for å lagre kjøretids-
// informasjon om objekter som ligger i objektbufferet. Ikke alle variablene
// er like nødvendige, og de kan kuttes ved optimalisering.
// Burde nok kalt denne klassen "ObjektDescriptor".

class Objekt {
public:
   Objekt();

   ~Objekt();

   bool skittent;
   bool laast;
   bool brukt;
   int stoerrelse;
   int offset;
   char *bufferadresse;  // Unødvendig, men gjør ting litt enklere
   Objekt *neste;
   LogOID logOID;  // Måtte ha denne på grunn av skriveoptimalisering, ellers
                   // er den unødvendig
};

// Et objektbuffer forsøker å holde utvalgte objekter i hovedminnet, slik at
// man ved gjenntatt bruk skal slippe å måtte lese dem inn fra disken igjen.
// Siden bare et begrenset antall objekter får plass i bufferet, er det svært
// viktig å holde på dem som brukes mest. For at permanent databuffring skal
// være effektivt, må der være lokalitet i referansene. Det vil si at en stor
// andel av forespørslene refererer en liten del av databasen.

class Objektbuffer {
public:
   Objektbuffer(int stoerrelse);

   ~Objektbuffer();

   char *HentObjekt(FysOID fysOID, int lengde, LogOID logOID = -1);
            // Henter inn objektet i bufferet. Objektet låses for å
            // hindre utkasting når det brukes av høyere lag.

   void FrigiObjekt(FysOID fysOID, bool skittent);
           // Låser opp objektet, og registrerer om det eventuelt er skittent

   void Flush();
            // Skriver alle skitne objekter til disken

   void NullstillStatistikk();
            // Setter antall logiske og fysiske referanser til null

   Statistikk HentStatistikk();
           // Returnerer statistikk informasjon siden siste nullstilling

   void SkrivUtMinnekart();
           // Skriver ut et karakterkart over bufferminnet

   void SkrivUtStatistikk();
           // Skriver ut statisikk informasjon

   void VelgModus(BufferModus modus);
           // Velger mellom Normal og Skriveoptimalisert modus

private:
   char *AllokerBufferplass(int antallEnheter);
           // Allokerer plass i bufferet, returnerer NULL dersom forespørselen
           // ikke kan tilfredsstilles

   Blokk *FrigiBufferplass(char *bufferadresse);
```



```
           // Deallokerer plassen som "bufferadresse" peker på.
           //"Bufferadresse" må være allokert med "AllokerBufferplass".

   bool KastUtObjekter(int antallEnheter);
           // Kaster ut objekter fra bufferet helt til en blokk som er
           // "antallEnheter" stort blir ledig.

   Objekt *FinnInnlastetObjekt(int sidenr, int offset);
           // Søker i hashtabellen etter aktuelt objekt og returnerer det.
           // Dersom objektet ikke er innlastet returneres NULL.

   void RegistrerInnlastetObjekt(int sidenr, Objekt *objekt);
           // Registrerer at et objekt er innlastet.

   void SlettRegistreringAvInnlasting(Objekt *objekt, int sidenr);
           // Sletter registreringen av at objektet er innlastet i minnet.

   void FinnNesteOffer();
           // Finner det neste objektet som skal kastes ut.

   Statistikk BeregnMinneStatistikk();
           // Går gjennom bufferminnet, og beregner ledige og opptatte enheter
           // samt minneutnyttelse

   Blokk *foersteLedigeBlokk;  // Den første blokka som er ledig
   int bufferStoerrelse;  // Antall enheter
   int sumStoerrelse;  // Summen av alle objektene i bufferet, i byte
   int antallSider;
   int maksAntallSider;
   int antallAllokerteEnheter;

   char *buffer;  // Sammenhengende minneområde til buffring av objekter
   Objekt **hashtabell;          // Like mange elementer i tabellen som
                                 // antall sider i databasen. Bruker
                                 // sidenr i den fysiske OID som indeks
                                 // i tabellen.
   Objekt *offer;  // Det objektet som skal kastes ut
   int offerSidenr;  // Angir hvilken side offeret ligger på

   int antallForespoersler;
   int antallMiss;

   BufferModus modus;  // Angir om bufferet kjører i skriveoptimalisert eller
                       // normal modus
   SekvensiellFil *nyDatafil;  // I skriveoptimalisert modus blir skitne
                               // objekter overført til ei ny fil
   Blokk **hodetabell;  // Ett hode for hver ledigliste
   Blokk *hode;  // Hodet til den aktive lista

};

#endif  // OBJEKTBUFFER_H
```

C.11 objektbufferV2.cpp

```
// objektbufferV2.cpp
// Rutiner for å håndtere et objektbuffer.

#include "systemV0.h"
#include "databaseV0.h"
#include "sidebufferV1.h"
#include "objektbufferV2.h"
#include "hashtabellV0.h"
```

```
// Disse er definert i main.cpp
extern Database *database;
extern Sidebuffer *sidebuffer;
extern Hashtabell *katalog;

// Blokk::Blokk
// Initialiserer blokka

Blokk::Blokk()
{
   neste = NULL;
   forrige = NULL;
   antallEnheter = 0;
}

// Objekt::Objekt
// Setter flaggene skittent, laast og brukt til FALSK. Bufferadresse og neste
// får verdien NULL.

Objekt::Objekt()
{
   skittent = FALSK;
   laast = FALSK;
   brukt = FALSK;
   bufferadresse = NULL;
   neste = NULL;
}

// Objekt::~Objekt

Objekt::~Objekt()
{

}

// Objektbuffer::Objektbuffer
// Allokerer et område i hovedminnet som skal brukes til å buffre objekter fra
// databasen. Hashtabellen opprettes og nullstilles. Størrelsen på den
// finnes ved å se hvor mange sider databasen er. Den første ledige blokka
// settes til å være hele bufferminnet, og til slutt nullstilles statistikk-
// variablene. All intern minnehåndtering skjer i enheter, der en enhet er
// størrelsen på den kontrollinformasjonen som ligger først i hver ledig blokk.
// Det underliggende sidebufferet opprettes også.
//
// "antallSiderIBuffer" er bufferstørrelsen i sider

Objektbuffer::Objektbuffer(int antallSiderIBuffer)
{
   int i;
   assert((antallSiderIBuffer % AntallSiderPerSegment) == 0);
   assert(((AntallSiderPerSegment * Sidestoerrelse) % sizeof(Blokk)) == 0);
   delete sidebuffer;
   sidebuffer = new Sidebuffer(antallSiderIBuffer - AntallSiderPerSegment);

   maksAntallSider = antallSiderIBuffer - AntallSiderPerSegment;
   antallSider = AntallSiderPerSegment;  // Start med et segment stort objekt-
                                         // buffer
   antallAllokerteEnheter = 0;
   modus = Normal;
   sumStoerrelse = 0;

   assert(
      buffer = (char *) new Blokk[bufferStoerrelse]);
   // Aksesserer hver side i bufferminnet for å få lest de inn i minnet
   for (i = 0; i < maksAntallSider * Sidestoerrelse; i += 4096) {
      buffer[i] = 'a';
   }
   bufferStoerrelse = (AntallSiderPerSegment * Sidestoerrelse - 1)
                                 / sizeof(Blokk) + 1;  // I enheter

   debug('o', "Objektbuffer: Minste allokeringsenhet er %d\n", sizeof(Blokk));
   debug( 'o', "Objektbuffer: Allokerte %d byte (%d enheter) for buffer.\n",
                     AntallSiderPerSegment * Sidestoerrelse, bufferStoerrelse);
   // I starten er bufferet tomt, med kun en ledig blokk
   hodetabell = new Blokk*[AntallLediglister + 1];
   for (i = 0; i <= AntallLediglister; i ++) {
      hode = new Blokk();
      hode->neste = hode;
      hode->forrige = hode;
      hodetabell[i] = hode;
   }

   Blokk *blokk = (Blokk *) buffer;
   hode = hodetabell[AntallLediglister];
   hode->neste = blokk;
   hode->forrige = blokk;
   blokk->neste = hode;
   blokk->forrige = hode;
   blokk->antallEnheter = bufferStoerrelse;

   Blokkhale *blokkhale = (Blokkhale *) (blokk + blokk->antallEnheter - 1);
   blokkhale->ledig = SANN;
   blokkhale->antallEnheter = blokk->antallEnheter;

   // Initialiser hashtabell
   int antallSiderIDatabase = database->AntallSider();
   hashtabell =  new Objekt*[antallSiderIDatabase];
   assert(hashtabell != NULL);
   for (i = 0; i < antallSiderIDatabase; i ++)
      hashtabell[i] = NULL;

   offer = NULL;
   offerSidenr = -1;
   NullstillStatistikk();
}

// Objektbuffer::~Objektbuffer
// Sletter bufferområdet og hashtabellen.

Objektbuffer::~Objektbuffer()
{
   delete buffer;
   delete [] hashtabell;
}

// Objektbuffer::HentObjekt
// Henter et objekt fra databasen og inn i bufferet. Først undersøkes det om
// objektet allerede ligger i bufferet. Sidenummeret i den fysiske OID'en
// brukes som indeks i en hashtabell som gir en peker til det første objektet
// i ei lenket liste av objekter. Denne lista er de objektene fra sida som er
// residente i bufferet. Dersom ikke objektet ligger inne, brukes
// "AllokerBufferplass" for å lage plass i bufferet. Objektet hentes så fra
// disken, og aktuell objektliste oppdateres. Objektet låses og en peker
// til objektet returneres.
//
// "fysOID" er fysisk OID, altså sidenummer og offset i databasen
// "storrelse" er objektets størrelse i byte
// "logOID" er logisk OID, er nødvendig på grunn av skriveoptimalisering

char *Objektbuffer::HentObjekt(FysOID fysOID, int stoerrelse, LogOID logOID)
{
   Objekt *objekt = FinnInnlastetObjekt(fysOID.sidenr, fysOID.offset);
   antallForespoersler++;
   if (objekt != NULL) { // Objektet ligger i bufferet
      debug('o', "Objektbuffer: Objektet (%d, %d) ble funnet i bufferet\n",
```

```
                                                fysOID.sidenr, fysOID.offset);
      return objekt->bufferadresse;
   }
   antallMiss++;  // Objektet ligger ikke i bufferet

   // Objektbufferet øker automatisk sin størrelse, og minker det underliggende
   // sidebufferets størrelse, når det er fullt.
   if (antallSider + AntallSiderPerSegment <= maksAntallSider) {
      if (antallAllokerteEnheter > 0.99 * bufferStoerrelse) {
         sidebuffer->ReduserAntallSegmenter();
         antallSider += AntallSiderPerSegment;
         int antallNyeEnheter = (AntallSiderPerSegment * Sidestoerrelse - 1)
                                / sizeof(Blokk) + 1;

         // Setter inn den nye blokka i lediglista
         Blokk *blokk = ((Blokk *) buffer) + bufferStoerrelse;
         hode = hodetabell[AntallLediglister];

         blokk->neste = hode->neste;
         hode->neste->forrige = blokk;
         hode->neste = blokk;
         blokk->forrige = hode;
         blokk->antallEnheter = antallNyeEnheter;

         Blokkhale *blokkhale =
                        (Blokkhale *) (blokk + blokk->antallEnheter - 1);
         blokkhale->ledig = SANN;
         blokkhale->antallEnheter = blokk->antallEnheter;
         bufferStoerrelse += antallNyeEnheter;
      }
   }
   assert(
      objekt = new Objekt());  // Bør i grunnen ikke bruke assert her
   objekt->offset = fysOID.offset;
   objekt->laast = SANN;
   objekt->stoerrelse = stoerrelse;
   objekt->logOID = logOID;
   int antallEnheter = (stoerrelse - 4) / sizeof(Blokk) + 2;
   char *bufferadresse;
   do {
      bufferadresse = AllokerBufferplass(antallEnheter);
      if (bufferadresse == NULL) {  // Ikke ledig plass i bufferet
         KastUtObjekter(antallEnheter);
      }
   } while (bufferadresse == NULL);

   objekt->bufferadresse = bufferadresse;
   debug('o', "Objektbuffer: Henter objektet (%d, %d) fra disken\n",
                                       fysOID.sidenr, fysOID.offset);
   // Hent inn sida fra databasen, og kopier objektet fra
   // sidebufferet til objektbufferet
   char *sidebufferadr = sidebuffer->HentSide(fysOID.sidenr);
   memcpy((void *)bufferadresse, (void *)(sidebufferadr + fysOID.offset),
                                                          stoerrelse);
   RegistrerInnlastetObjekt(fysOID.sidenr, objekt);
   sidebuffer->FrigiSide(fysOID.sidenr, FALSK);
   return objekt->bufferadresse;
}

// Objektbuffer::FrigiObjekt
// Objektet låses opp, bruktflagget settes og skittentflagget oppdateres.
//
// "fysOID" er fysisk OID, altså sidenummer og offset i databasen
// "skittent" er et flagg som angir om objektet er endret den tiden det var
//            låst, og derfor må skrives tilbake til disken ved utkasting

void Objektbuffer::FrigiObjekt(FysOID fysOID, bool skittent)
{
   Objekt *objekt = FinnInnlastetObjekt(fysOID.sidenr, fysOID.offset);
   assert(objekt);  // Objektet må ligge i bufferet
   objekt->laast = FALSK;  // lås opp
   objekt->brukt = SANN;
   if (!objekt->skittent) {  // Dersom ikke allerede er skittent
      objekt->skittent = skittent;
   }
   debug('o', "Objektbuffer: Objektet (%d, %d) ble laast opp, skittent = %d\n",
                                  fysOID.sidenr, fysOID.offset, objekt->skittent);
}

// Objektbuffer::Flush
// Skriver alle skitne objekter til disken. Dersom et objekt er skittent,
// hentes eventuelt aktuell side inn i sidebufferet, og objektet kopieres
// over der. Etter at alle objektene er undersøkt, flushes sidebufferet.

void Objektbuffer::Flush()
{
   Objekt *objekt = NULL;
   int sidenr = -1;

   while (sidenr < database->AntallSider()) {
      while (objekt == NULL) { // Spol nedover i hashtabellen
         sidenr++;
         if (sidenr >= database->AntallSider()) { // I tilfelle at siste objekt
                                                  // i hashtabellen er NULL
            break;
         }
         objekt = hashtabell[sidenr];
      }
      if (objekt == NULL) {
         break;
      }
      assert(objekt);
      if (objekt->skittent) {
         if (modus == Normal) {
            char *sidebufferadr =
                     sidebuffer->HentSide(sidenr) + objekt->offset;
            memcpy((void *)sidebufferadr, (void *)objekt->bufferadresse,
                                                   objekt->stoerrelse);
            sidebuffer->FrigiSide(sidenr, SANN); // Låser opp sida
            objekt->skittent = FALSK;
            objekt = objekt->neste;
         } else {  // skriveoptimalisert modus
            // Flytt objektet over på ei ny side
            FysOID fysOID =
              nyDatafil->SettInnObjekt(objekt->bufferadresse,
                                                   objekt->stoerrelse);
            Objekt *gammeltObjekt = objekt;
            Objekt *nyttObjekt = new Objekt();
            *nyttObjekt = *objekt;
            objekt = objekt->neste;
            nyttObjekt->offset = fysOID.offset;
            katalog->OppdaterElement(nyttObjekt->logOID, fysOID);
            SlettRegistreringAvInnlasting(gammeltObjekt, sidenr);
            RegistrerInnlastetObjekt(fysOID.sidenr, nyttObjekt);
            nyttObjekt->skittent = FALSK;
         }
      } else {
         objekt = objekt->neste;  // Gå til neste objekt
      }
   }
   sidebuffer->Flush();
}

// Objektbuffer::AllokerBufferplass
// Finner ei ledig blokk i bufferet. Fritt lager holdes som flere dobbelt
// lenkede lister av ledige blokker basert på blokkstørrelse. Hodetabellen
```

```
// har for hver ledigliste et hode. Det peker på første og siste blokk i
// lediglista for aktuell størrelsesklasse. Blokkene holdes i vilkårlig
// rekkefølge. Blokka finnes ved å ta den første blokka fra ei liste som
// ikke er tom, og som også er >= "antallEnheter". Dersom
// blokka passer akkurat, fjernes den fra lista og returneres. Ellers splittes
// den, og den tilbakeværende delen lenkes inn i korrekt liste.
// Dersom ingen stor nok blokk finnes, returneres NULL.
//
// "antallEnheter" er hvor stor plass som ønskes i bufferet

char *Objektbuffer::AllokerBufferplass(int antallEnheter)
{
   debug('o', "Objektbuffer: Allokerer %d enheter for objekt\n",
                                                  antallEnheter);
   assert(antallEnheter <= bufferStoerrelse);

   // Må først velge den riktige lediglista
   if (antallEnheter >= AntallLediglister) {
      hode = hodetabell[AntallLediglister];
   } else {
      for (int i = antallEnheter; i <= AntallLediglister; i++) {
         hode = hodetabell[i];
         if (hode->neste != hode) {
            break;
         }
      }
   }

   if (hode->neste == hode) {  // Ingen ledige blokker, vi må kaste ut objekter
      debug('o', "Objektbuffer: Ingen ledige blokker\n");
      return NULL;
   }

   Blokk *blokk = hode->neste;  // Velg første blokka fra lediglista
   Blokkhale *blokkhale;
   assert(blokk != NULL);
   bool funnet = SANN; // Antar at vi finner ei blokk
   while (funnet) {
      if (blokk->antallEnheter >= antallEnheter) {  // stor nok
         if (blokk->antallEnheter == antallEnheter ||
             blokk->antallEnheter == antallEnheter) {  // eksakt -> fjerner
                                                       // blokka fra lediglista
            blokk->forrige->neste = blokk->neste;
            blokk->neste->forrige = blokk->forrige;
            debug('o', "Objektbuffer: Fant eksakt blokk\n");
         } else { // Splitter blokka og allokerer halen
            debug('o', "Objektbuffer: Splitter blokk\n");
            blokk->antallEnheter -= antallEnheter;
            assert(blokk->antallEnheter >= 1);

            blokk->forrige->neste = blokk->neste;
            blokk->neste->forrige = blokk->forrige;

            // Oppdaterer den nye ledige blokka med nødvendig
            // kontrollinformasjon
            blokkhale = (Blokkhale *) (blokk + blokk->antallEnheter - 1);
            blokkhale->antallEnheter = blokk->antallEnheter;
            blokkhale->ledig = SANN;

            if (blokk->antallEnheter == 1) {
               blokkhale->ledig = 2;
            } else {
               if (blokk->antallEnheter >= AntallLediglister) {
                  hode = hodetabell[AntallLediglister];
               } else {
                  hode = hodetabell[blokk->antallEnheter];
               }
               blokk->neste = hode->neste;
               hode->neste->forrige = blokk;
               hode->neste = blokk;
               blokk->forrige = hode;
            }

            blokk += blokk->antallEnheter;
            blokk->antallEnheter = antallEnheter;

         }
         blokk->neste = NULL;  // Dette markerer at blokka er reservert
         blokkhale = (Blokkhale *) (blokk + blokk->antallEnheter - 1);
         blokkhale->ledig = FALSK;

         antallAllokerteEnheter += antallEnheter;
         if (DebugSkrivUtMinneKart)
            SkrivUtMinnekart();
         return (char *) (blokk + 1) - 4;
      }
      if (blokk == hode) {  // Gått gjennom hele lista
         debug('o', "Objektbuffer: Ingen av de ledige blokkene var store nok\n");
         break;
      } else {  // Gå til neste ledige blokk
         blokk = blokk->neste;
      }
   }
   return NULL;
}

// Objektbuffer::FrigiBufferplass
// Frigir ei blokk. På grunn av at det er lagt inn kontrollinformasjon i halen,
// trenger vi ikke å søke etter korrekt plass. Dersom blokka som frigis
// er sammenstøtende med ei anna fri blokk (på ei av sidene), slås de sammen
// til ei større blokk for å minske fragmenteringen. Vi finner starten på den
// venstre blokka ved å benytte kontrollinformasjoen i halen på den, mens den
// høyre blokka finnes ved å gå tilsvarende antall enheter til høyre som
// størrelsen på den frigitte blokka. Til slutt returneres den frigitte blokka.
//
// "bufferadresse" peker på den blokka som skal frigis

Blokk *Objektbuffer::FrigiBufferplass(char *bufferadresse)
{
   Blokk *blokk = (Blokk *)(bufferadresse + 4) - 1; // Den nye blokka som skal
                                                    // settes inn i lista
   Blokk *venstreBlokk, *hoeyreBlokk;
   antallAllokerteEnheter -= blokk->antallEnheter;

   // Sjekker om den venstre blokka er ledig
   if (!((char *)blokk == buffer)) {
      Blokkhale *venstreBlokkhale = (Blokkhale *)(blokk - 1);
      if (venstreBlokkhale->ledig) {
         debug('o', "Objektbuffer: Slaar sammen frigitt blokk med blokk til "
                                                              "venstre\n");
         venstreBlokk = blokk - venstreBlokkhale->antallEnheter;
         if (!(venstreBlokkhale->ledig == 2)) {  // Spesialtilfelle for ledige
                                                 // blokker på en enhet
            venstreBlokk->forrige->neste = venstreBlokk->neste;
            venstreBlokk->neste->forrige = venstreBlokk->forrige;
         }
         venstreBlokk->antallEnheter += blokk->antallEnheter;
         blokk = venstreBlokk;
      }
   }
   // Sjekker om den høyre blokka er ledig
   if (!(blokk + blokk->antallEnheter == (Blokk *)buffer + bufferStoerrelse)) {
      hoeyreBlokk = blokk + blokk->antallEnheter;
      if (hoeyreBlokk->neste != NULL) {
         debug('o', "Objektbuffer: Slaar sammen frigitt blokk med blokk til "
                                                              "hoeyre\n");
```

```
         if (hoeyreBlokk->antallEnheter > 1) {
            hoeyreBlokk->forrige->neste = hoeyreBlokk->neste;
            hoeyreBlokk->neste->forrige = hoeyreBlokk->forrige;
         }
         blokk->antallEnheter += hoeyreBlokk->antallEnheter;
      }
   }

   Blokkhale *blokkhale = (Blokkhale *)(blokk + blokk->antallEnheter - 1);
   blokkhale->antallEnheter = blokk->antallEnheter;
   blokkhale->ledig = SANN;

   // Koble den nye blokka inn på riktig ledigliste
   if (blokk->antallEnheter >= AntallLediglister) {
      hode = hodetabell[AntallLediglister];
   } else {
      hode = hodetabell[blokk->antallEnheter];
   }
   blokk->neste = hode->neste;
   hode->neste->forrige = blokk;
   hode->neste = blokk;
   blokk->forrige = hode;

   if (DebugSkrivUtMinneKart)
      SkrivUtMinnekart();
   return blokk;
}

// Objektbuffer::KastUtObjekter
// Kaster ut objekter helt til ei blokk på "antallEnheter" blir ledig. Dersom
// et objekt har bruktbitet satt, nullstilles det, og det neste
// objektet undersøkes. Skitne objekter må skrives til disken først.
// "FinnNesteOffer()" bestemmer hvilket objekt som skal ut.
//
// "antallEnheter" angir hvor stor blokk som ønskes

bool Objektbuffer::KastUtObjekter(int antallEnheter)
{
   bool blokkFunnet = FALSK;
   while (!blokkFunnet) {
      FinnNesteOffer();
      debug('o', "Objektbuffer: Kaster ut objekt (%d, %d)\n", offerSidenr,
                                                              offer->offset);
      if (offer->skittent) {
         char *sidebufferadr =
                     sidebuffer->HentSide(offerSidenr) + offer->offset;
         memcpy((void *)sidebufferadr, (void *)offer->bufferadresse,
                                           offer->stoerrelse);
         sidebuffer->FrigiSide(offerSidenr, SANN); // Låser opp sida
      }
      Blokk *blokk = FrigiBufferplass(offer->bufferadresse);
      Objekt *forrigeOffer = offer;
      offer = offer->neste;
      SlettRegistreringAvInnlasting(forrigeOffer, offerSidenr);
      if (blokk->antallEnheter >= antallEnheter) {
         blokkFunnet = SANN;  // Stor nok blokk er frigjort
      }
   }
   return SANN;
}

// Objektbuffer::FinnInnlastetObjekt
// Søker i hashtabellen etter aktuelt objekt og returnerer det. NULL returneres
// dersom det ikke finnes.
//
// "sidenr" er den sida i databasen som objektet ligger på
// "offset" er hvor på sida objektet ligger

Objekt *Objektbuffer::FinnInnlastetObjekt(int sidenr, int offset)
{
   Objekt *objekt = hashtabell[sidenr];  // Starter med første objekt fra sida
   while (objekt != NULL) {  // Søk gjennom lista
      if (objekt->offset == offset) {
         return objekt;  // Objektet ligger i bufferet
      }
      else
         objekt = objekt->neste;  // Gå til neste element i lista
   }
   return NULL;
}

// Objektbuffer::RegistrerInnlastetObjekt
// Registrerer at et objekt er innlastet i bufferet ved å lenke det inn
// i korrekt objektliste. Hashtabellen har en objektpeker for hver side i
// databasen. Den peker til ei liste av objekter som er innlastet fra den
// sida.
//
// "sidenr" er den sida i databasen som objektet ligger på
// "nyttObjekt" er det nye objektet som er blitt innlastet

void Objektbuffer::RegistrerInnlastetObjekt(int sidenr, Objekt *nyttObjekt)
{
   Objekt *objekt = hashtabell[sidenr];  // Første objekt i lista
   assert(sidenr < database->AntallSider());
   if (objekt == NULL) { // Lista er tom
      nyttObjekt->neste = NULL;
      hashtabell[sidenr] = nyttObjekt;
   } else {
      while (objekt->neste != NULL) {  // Spol fram til siste objekt
         objekt = objekt->neste;
      }
      objekt->neste = nyttObjekt;  // Lenk inn nytt objekt
      nyttObjekt->neste = NULL;
   }
   sumStoerrelse += nyttObjekt->stoerrelse;  // Oppdater hvor mange byte med
                                             // objekter som ligger i bufferet
}

// Objektbuffer::SlettRegistreringAvInnlasting
// Sletter objektet fra lista over innlastete objekter fra samme side.
//
// "objekt" er det objektet som er blitt kastet ut
// "sidenr" er den sida som objektet ligger på

void Objektbuffer::SlettRegistreringAvInnlasting(Objekt *objekt, int sidenr)
{
   Objekt *objektForan =  // Objektet foran det som skal kastes ut
               hashtabell[sidenr];  // Første objekt i objektlista for
                                    // aktuell side
   assert(objekt);
   if (objektForan == objekt) {  // Objektet er først i lista
      hashtabell[sidenr] = objekt->neste;
   }
   else {
      while (objektForan->neste != objekt) {  // Finn det objektet som ligger
         objektForan = objektForan->neste;    // foran det som skal slettes
      }
      objektForan->neste = objekt->neste;
   }
   sumStoerrelse -= objekt->stoerrelse;      // Oppdater hvor mange byte med
                                             // objekter som ligger i bufferet
   delete objekt;
}

// Objektbuffer::FinnNesteOffer
// Objekt utskiftingsalgoritmen som brukes er en tilnærming til LRU-algoritmen,
```

```
// og virker nesten likt som FIFO utskifting. Når et objekt er valgt, inspiserer
// vi bruktbitet. Dersom det er 0, kastes objektet ut. Derimot hvis bitet er 1,
// nullstilles bitet, og vi fortsetter til neste objekt. Dette kalles klokke-
// algoritmen.

void Objektbuffer::FinnNesteOffer()
{
   bool offerFunnet = FALSK;
   while (!offerFunnet) {
      while (offer == NULL) { // Spol nedover i hashtabellen
         offerSidenr = (offerSidenr + 1) % database->AntallSider();
         offer = hashtabell[offerSidenr];
      }
      if (offer->brukt) {  // Gå bortover i objektlista
         offer->brukt = FALSK;
         debug('o', "Objektbuffer: Objekt (%d, %d) fikk en ny sjanse\n",
                                          offerSidenr, offer->offset);
         offer = offer->neste;
      } else {
         offerFunnet = SANN;
      }
   }
}

// Objektbuffer::NullstillStatistikk
// Nullstiller antall forespørsler og antall miss.

void Objektbuffer::NullstillStatistikk()
{
   antallForespoersler = 0;
   antallMiss = 0;
}

// Objektbuffer::HentStatistikk
// Henter all relevant statisikkinformasjon til bufferet, og returnerer den.

Statistikk Objektbuffer::HentStatistikk()
{
   Statistikk statistikk = BeregnMinneStatistikk();
   statistikk.antallForespoersler = antallForespoersler;
   statistikk.antallMiss = antallMiss;
   return statistikk;
}

// Objektbuffer::BeregnMinneStatistikk
// Beregner minneutnyttelse, antall ledige og reserverte blokker. Alle
// blokkene gjennomgås og reservert/ledig plass summeres. Dersom neste
// pekeren er NULL, vet vi at blokka er reservert. Dette kunne vært gjort med
// kun et bit, men ble valgt for enkelhet.

Statistikk Objektbuffer::BeregnMinneStatistikk()
{
   int antallEnheterReserverte = 0;
   int antallEnheterLedige = 0;
   int antallObjekter = 0;
   int antallLedigeBlokker = 0;

   Blokk *blokk = (Blokk *)buffer;  //  Start med første blokk i bufferet

   while((antallEnheterLedige + antallEnheterReserverte) < bufferStoerrelse) {
      if (blokk->neste == NULL) {  // blokka er opptatt
         antallEnheterReserverte += blokk->antallEnheter;
         antallObjekter++;
      } else {  // blokka er ledig
         antallEnheterLedige += blokk->antallEnheter;
         antallLedigeBlokker++;
      }
      blokk += blokk->antallEnheter;  // Gå til neste blokk
   }

   Statistikk statistikk;
   statistikk.minneutnyttelse =
              sumStoerrelse / ((double) bufferStoerrelse * 12) * 100;
            //((1 - ((antallObjekter * 14 + antallEnheterLedige * 12)
            /// ((double) bufferStoerrelse * 12) )) * 100;
   statistikk.antallLedigeBlokker = antallLedigeBlokker;
   statistikk.antallObjekter = antallObjekter;
   statistikk.antallEnheterReserverte = antallEnheterReserverte;
   statistikk.antallEnheterLedige = antallEnheterLedige;

   return statistikk;
}

// Objektbuffer::SkrivUtStatistikk
// Skriver statistikkinformasjonen til skjermen

void Objektbuffer::SkrivUtStatistikk()
{
   Statistikk s = BeregnMinneStatistikk();

   printf("Objektbuffer: Antall forespoersler: %d, Antall miss: %d, "
            "Objektmiss(%%): %.2f\n", antallForespoersler, antallMiss,
                        (double)antallMiss / antallForespoersler * 100);
   printf("\tEnheter: Totalt: %d, Reserverte: %d, "
          "Ledige: %d, Minneutnyttelse: %.2f %% \n", bufferStoerrelse,
          s.antallEnheterReserverte, s.antallEnheterLedige,
                                                        s.minneutnyttelse);
   printf("\tDet er %d objekter i bufferet, og %d ledige blokker\n",
                                     s.antallObjekter, s.antallLedigeBlokker);
}

// Objektbuffer::SkrivUtMinnekart
// Går gjennom alle blokkene i bufferet og skriver dem til skjermen. Ei
// reservert blokk på 60 byte skrives som "|XXXXX", mens ei ledig blokk på
// 72 byte skrives som "§......". Blokkene skrives sortert etter stigende
// adresse, med 80 enheter per linje.

void Objektbuffer::SkrivUtMinnekart()
{
   const int Linjeskift = 80;  // Er antall karakterer per linje
   Blokk *blokk = (Blokk *)buffer;  //  Start med første blokk i bufferet
   int enheterUtskrevet = 0;

   while( enheterUtskrevet < bufferStoerrelse) {
      char ch;
      if (blokk->neste == NULL) {  // Blokka er opptatt
         ch = 'X';
         putchar('|');  // Indikerer start på reservert blokk
         enheterUtskrevet++;
      } else {  // Blokka er ledig
         ch = '.';
         putchar('$');  // Indikerer start på ledig blokk
         enheterUtskrevet++;
      }
      if (enheterUtskrevet % Linjeskift == 0) {
         putchar('\n');
      }
      for (int i = 0; i < blokk->antallEnheter - 1; i++) {
         putchar(ch);
         enheterUtskrevet++;
         if (enheterUtskrevet % Linjeskift == 0) {
            putchar('\n');
         }
      }
      blokk += blokk->antallEnheter;  // Gå til neste blokk
   }
```

```
}

// Objektbuffer::VelgModus
// Velger mellom normal og skriveoptimalisert modus. Ved kjøring i sistnevnte
// modus opprettes ei ny datafil som skitne objekter skrives til. Datafila
// låser den siste sida si i sidebufferet for rask skriving.
//
// "modus" kan være Normal eller Skriveoptimalisert

void Objektbuffer::VelgModus(BufferModus nyttModus)
{
   modus = nyttModus;
   if (modus == Normal) {
      delete nyDatafil;
   }
   if (modus == Skriveoptimalisert) {
      delete nyDatafil;
      nyDatafil = new SekvensiellFil(SANN);  // Med låst siste side
   }
}
```

C.12 sidebufferV1.h

```
// sidebufferV1.h
// Datastrukturer for håndtering av et sidebuffer. I versjon 1 er LRU brukt som
strategi
// for utskifting av sider i steden for klokkealgoritmen. Ellers er alt som
// før.

#ifndef SIDEBUFFER_H
#define SIDEBUFFER_H

#include "systemV0.h"

// Denne klassen definerer en ramme. En ramme inneholder administrativ
// informasjon som brukes når ei side skal kastes ut av bufferet. Da må en
// vite hvilken side som ligger i rammen, og om den er skitten eller låst av
// høyere lag. I tillegg trenger klokke sideerstatningsalgoritmen å vite om
// sida har vært brukt i forrige sirkulering.

class Ramme {
public:
   Ramme(); //Initialiserer flaggene

   ~Ramme();

   int sidenr;       // Disksiden som ligger i rammen
   bool skitten;      // Angir om sida er endret siden den ble lest inn
   bool laast;        // Angir om sida er låst av høyere lag
   char *bufferadresse;

   // Disse brukes til LRU-lista
   Ramme *neste;
   Ramme *forrige;
};

// Den følgende klassen definerer et sidebuffer. Et sidebuffer har som oppgave
// å være et grensesnitt mellom hovedminnet og disken. Bufferet består av rammer
// som hver er like stor som ei diskside.

class Sidebuffer {
public:
   Sidebuffer(int antallRammerIBuffer);
            // Oppretter et bufferbasseng i hovedminnet som er stort
            // nok til å romme et visst antall sider

   ~Sidebuffer();
            // Sletter bufferbassenget

   char *HentSide(int sidenr, bool sidaErTom = FALSK);
           // Henter sida inn i en ledig ramme i bufferet desom den ikke
           // allerede ligger inne. Sida låses for å hindre utkasting
           // når den brukes av høyere lag.

   void FrigiSide(int sidenr, int skitten = FALSK);
           // Låser opp sida. Selve skrivingen til disken, dersom sida
           // er skitten, utsettes til bufferet er fullt og sida blir
           // offer for utkasting.

   void Flush();
            // Skriver alle skitne sider til disken

   void NullstillStatistikk();
            // Setter antall logiske og fysiske referanser til null

   void HentStatistikk(int &logiskeReferanser, int &fysiskeReferanser);
            // Henter antall logiske og fysiske referanser siden
            // forrige nullstilling

   void SkrivUtStatistikk();
            // Skriver statistikkinformasjon til skjermen

   void ReduserAntallSegmenter();
            // Reduserer minneområdet som brukes til å buffre databasesider
            // med ett segment, det vil si "AntallSiderPerSegment" sider

private:
   Ramme *FinnOffer();
            // Finner, etter en LRU-strategi, den sida som skal kastes ut

   int antallLedigeRammer;  // Holder oversikt over om bufferet er fullt
   int offernr;  // Rammenummeret til offeret

   void OppdaterLRUListe(Ramme *ramme);
            // Flytter rammen først i LRU-lista

   void FjernFraLRUListe(Ramme *ramme);
            // Fjerner rammen fra LRU-lista

   int antallRammer;   // Angir hvor mange sider det er plass til i bufferet

   char *buffer;  // Peker til et område i hovedminnet som er stort
                  // nok til å lagre et visst antall sider
   Ramme *rammer;
            // En tabell til å lagre administrativ informasjon om
            // hver ramme

   int antallLogiskeReferanser;  // Angir det totale antall sideforespørsler
   int antallFysiskeReferanser;  // Er antall sideforespørsler til sider som ikke
                                 // ligger i bufferet, og som må hentes fra disken
   Ramme **oversettingstabell;  // Oversetter sidenummer til rammenummer. Dersom
                                // sida ikke er innlaster, er innslaget NULL.
   Ramme *hode;  // Peker på første og siste element i LRU-lista
};

#endif  // SIDEBUFFER_H
```

C.13 sidebufferV1.cpp

```
// sidebufferV1.cpp
// Rutiner for å håndtere et sidebuffer.

#include "sidebufferV1.h"
#include "databaseV0.h"

extern Database *database;

// Ramme::Ramme
// Setter flaggene skitten, låst og brukt til FALSK

Ramme::Ramme()
{
   sidenr = -1;
   skitten = FALSK;
   laast = FALSK;
   neste = NULL;
   forrige = NULL;
}

// Ramme::~Ramme

Ramme::~Ramme()
{

}

// Sidebuffer::Sidebuffer
// Allokerer et område i hovedminnet til å buffre sider fra databasen.
// Oppretter også tabellen med rammer, og setter rammepekeren
// til første rammen.
//
// "antallRammerIBuffer" er størrelsen på bufferbassenget gitt ved antall rammer

Sidebuffer::Sidebuffer(int antallRammerIBuffer)
{
   int i;

   // Antall rammer må være et helt antall segment
   assert((antallRammerIBuffer % AntallSiderPerSegment) == 0);
   // Initialiserer ei tom LRU-liste
   hode = new Ramme();
   hode->neste  = hode;
   hode->forrige = hode;

   antallRammer = antallRammerIBuffer;
   antallLedigeRammer = antallRammer;
   offernr = -1;
   // Nullstiller statistikk variabler
   antallLogiskeReferanser = 0;
   antallFysiskeReferanser = 0;

   debug('s', "Sidebuffer: Allokerer %d bytes til buffer\n",
                           antallRammer * Sidestoerrelse);
   buffer = new char [antallRammer * Sidestoerrelse];
   for (i = 0; i < antallRammer * Sidestoerrelse; i += 4096) {
      buffer[i] = 'a';
   }
   assert(buffer != NULL);
   rammer = new Ramme [antallRammer];
   assert(rammer != NULL);
   for (i = 0; i < antallRammer; i++) {
      rammer[i].bufferadresse = buffer + i * Sidestoerrelse ;
      assert(rammer[i].bufferadresse != NULL);
   }

   // Initialiser oversettingstabellen
   int antallSider = database->AntallSider();
   assert(
      oversettingstabell = new Ramme*[antallSider]);
   for (i = 0; i < antallSider; i++) {
      oversettingstabell[i] = NULL;  // Angir at sida ikke er innlastet
   }
}

// Sidebuffer::~Sidebuffer
// Sletter bufferet og ramme- og oversettingstabellen

Sidebuffer::~Sidebuffer()
{
   delete [] buffer;
   delete [] rammer;
   delete [] oversettingstabell;
}

// Sidebuffer::HentSide
// Henter ei side fra databasen og inn i bufferet. Dersom sida ikke
// allerede ligger i bufferet og det også er fullt, må ei annen side kastes
// ut. Rammen som rammepekeren indikerer sjekkes. Dersom tilhørende
// side er brukt siden sist gang, settes bruktflagget til FALSK og ramme-
// pekeren flyttes til neste ramme. Dette fortsetter helt til en har funnet
// ei ubrukt side. Den skrives så til databasen dersom den er skitten.
// Sida låses for å forhindre utkasting mens høyere lag bruker den.
// Returnerer peker til sida, eller NULL dersom alle sidene i bufferet er låst.
//
// "sidenr" er nummeret på den sida som ønskes innlastet i hovedminnet
// "sidaErTom" angir at sida er tom og det er derfor ikke nødvendig å lese den
//             inn fra disken

char *Sidebuffer::HentSide(int sidenr, bool sidaErTom)
{
   antallLogiskeReferanser++;

   if (oversettingstabell[sidenr] != NULL) {  // Er sida innlastet?
      Ramme *ramme = oversettingstabell[sidenr];
      debug('s', "Sidebuffer: Side %d funnet i bufferet\n", sidenr);
      ramme->laast = SANN;
      return ramme->bufferadresse;
   }
   // Sida er ikke i bufferet og vi trenger derfor en ramme
   Ramme *offer = FinnOffer();
   if (offer->sidenr != -1) {
      debug('s', "Sidebuffer: Kaster ut side %d fra bufferet\n",
                                                   offer->sidenr);
      oversettingstabell[offer->sidenr] = NULL;  // Merker at sida er utkastet
   }
   // Dersom sida er skitten skrives den til disken
   if (offer->skitten) {
      debug('s', "Sidebuffer: Side %d skitten, maa skrives til databasen\n",
                                                       offer->sidenr);
      database->SkrivSide(offer->sidenr, offer->bufferadresse);
   }
   char *bufferadresse = offer->bufferadresse;
   debug('s', "Sidebuffer: Henter side %d inn i bufferet\n", sidenr);
   oversettingstabell[sidenr] = offer;
   if (!sidaErTom) {
      database->LesSide(sidenr, bufferadresse);
      antallFysiskeReferanser++;  // sidemiss
   }

   // Initialiser rammeflaggene
   offer->laast = SANN;
```

```
   offer->sidenr = sidenr;
   offer->skitten = FALSK;

   return bufferadresse;
}

// Sidebuffer::FrigiSide
// Sida låses opp, merkes eventuelt skitten og bruktflagget settes
//
// "sidenr" er sida som skal låses opp
// "skitten" angir om sida er endret den tiden den har vært låst

void Sidebuffer::FrigiSide(int sidenr, int skitten)
{
   Ramme *ramme = oversettingstabell[sidenr];

   assert(ramme != NULL);
   assert(ramme->sidenr == sidenr);

   ramme->laast = FALSK;
   OppdaterLRUListe(ramme);
   if (!ramme->skitten) {
      ramme->skitten = (bool) skitten;
   }
   debug('s', "Sidebuffer: Laaser opp side %d , skitten = %d\n",
                                    sidenr,  ramme->skitten);
}

// Sidebuffer::Flush
// Går gjennom alle rammene og skriver skitne sider til disken

void Sidebuffer::Flush()
{
   debug('s', "Sidebuffer: Flusher alle skitne sider\n");
   for (int i = 0; i < antallRammer; i++) {
      if (rammer[i].skitten) {
         database->SkrivSide(rammer[i].sidenr, rammer[i].bufferadresse);
         rammer[i].skitten = FALSK;
      }
   }
}

// Sidebuffer::NullstillStatistikk
// Nullstiller statistikkvariablene

void Sidebuffer::NullstillStatistikk()
{
   antallLogiskeReferanser = 0;
   antallFysiskeReferanser = 0;
}

// Sidebuffer::HentStatistikk
// Henter antall logiske og fysiske referanser siden forrige nullstilling.
//
// "logiskeReferanser" er antall logiske referanser
// "fysiskeReferanser" er antall fysiske referanser

void Sidebuffer::HentStatistikk(int &logiskeReferanser, int &fysiskeReferanser)
{
   logiskeReferanser = antallLogiskeReferanser;
   fysiskeReferanser = antallFysiskeReferanser;
}

// Sidebuffer::SkrivUtStatistikk
// Skriver statistikkinformasjonen til skjermen

void Sidebuffer::SkrivUtStatistikk()
{
   printf("Sidebuffer: Antall forespoersler: %d, Antall miss: %d, "
             "Sidemiss(%%): %.2f\n", antallLogiskeReferanser,
             antallFysiskeReferanser,
             (double)antallFysiskeReferanser / antallLogiskeReferanser * 100);
}

// Sidebuffer::FinnOffer
// Finner den sida som skal kastes ut på grunn av at bufferet er fullt og ei ny
// side skal inn. Den første ulåste sida fra starten på LRU-lista velges som
// offer. LRU-lista vedlikeholdes ettersom sidene brukes.

Ramme *Sidebuffer::FinnOffer()
{
   Ramme *offer;

   if (antallLedigeRammer > 0) {
      antallLedigeRammer--;
      offernr++;
      offer = &rammer[offernr];
      debug('s', "Sidebuffer: Bufferet er ikke fullt, %d ledige rammer, "
                    "offernr: %d\n", antallLedigeRammer, offernr);
   } else {
      assert(
         offer = hode->forrige);  // Ta den første rammen i LRU-lista
      while (offer->laast) {
         offer = offer->forrige;
         assert(offer);
      }
   }
   return offer;
}

// Sidebuffer::OppdaterLRUListe
// Rammen fjernes fra LRU-lista og settes inn fremst i lista, såfremt den ikke
// allerede ligger først.
//
// "ramme" er den rammen som den nylig brukte sida ligger i

void Sidebuffer::OppdaterLRUListe(Ramme *ramme)
{
   assert(ramme != NULL);

   if (ramme == hode->neste) {  // Rammen ligger allerede først i lista
      return;
   }
   FjernFraLRUListe(ramme);  // Må først ta rammen ut av LRU-lista

   // Setter inn rammen som første element i LRU-lista
   ramme->neste = hode->neste;
   hode->neste->forrige = ramme;
   hode->neste = ramme;
   ramme->forrige = hode;
}

// Sidebuffer::FjernFraLRUListe
// Fjerner rammen fra LRU-lista.
//
// "ramme" er den rammen som skal fjernes fra LRU-lista

void Sidebuffer::FjernFraLRUListe(Ramme *ramme)
{
   if (ramme->neste == NULL && ramme->forrige == NULL) {
      ;  // Rammen er ikke i LRU-lista, og vi trenger derfor ikke fjerne den
         // fra lista
   } else {  // Rammen er et element i lista
      ramme->forrige->neste = ramme->neste;
      ramme->neste->forrige = ramme->forrige;
   }
```

```
}

// Sidebuffer::ReduserAntallSegmenter
// Reduserer minneområdet som brukes til å buffre databasesider
// med ett segment, det vil si "AntallSiderPerSegment" sider. Sidene i de
// siste rammene kastes ut, og rammene frigis. Denne funksjonen brukes av det
// adaptive objektbufferet.

void Sidebuffer::ReduserAntallSegmenter()
{
   assert(antallRammer >= AntallSiderPerSegment);

   Ramme *ramme;
   for (int i = antallRammer - AntallSiderPerSegment; i < antallRammer; i++) {
      ramme = &rammer[i];
      FjernFraLRUListe(ramme);
      if (ramme->sidenr != -1) {
         debug('s', "Sidebuffer: Kaster ut side %d fra bufferet\n",
                                          ramme->sidenr);
         oversettingstabell[ramme->sidenr] = NULL;  // Merker at sida er utkastet
      }
      // Dersom sida er skitten skrives den til disken
      if (ramme->skitten) {
         debug('s', "Sidebuffer: Side %d skitten, maa skrives til databasen\n",
                                          ramme->sidenr);
         database->SkrivSide(ramme->sidenr, ramme->bufferadresse);
      }
   }
   antallRammer -= AntallSiderPerSegment;
}
```

C.14 databaseV0.h

```
// database.h
// Datastrukturer for å simulere en database.

#ifndef DATABASE_H
#define DATABASE_H

#include "systemV0.h"

// Den følgende klassen definerer en database. Den er inndelt i et sett med
// sider. Sidene leses og skrives fra ei ordinær Windowsfil.

class Database {
public:
   Database(char *databasenavn, int antallSiderIDatabase);
            // Oppretter ei fil og en allokeringstabell

   ~Database();
            // Sletter fila og allokeringstabellen

   void LesSide(int sidenr, char *data);
           // Leser ei side fra disken og inn i hovedminnet

   void SkrivSide(int sidenr, char *data);
           // Skriver ei side fra hovedminnet til disken

   bool AllokerSide(int &sidenr);
           // Finner ei ledig side i databasen og allokerer den

   void DeallokerSide(int sidenr);
           // Frigir ei side i databasen

   int AntallSider();
            // Returnerer antall sider i databasen

   void SettAksesstid(unsigned int aksesstid);
            // Setter aksesstiden i millisekund

   void HentStatistikk(int &antallSiderLest, int &antallSiderSkrevet,
                                             int &antallAllokerteSider);
            // Henter noen interessante statisikkverdier

   void NullstillStatistikk();
            // Nullstiller statistikkvariablene

   void SkrivUtStatistikk();
            // Skriver statistikkinformasjonen til skjermen

   void VelgModus(DatabaseModus modus);
            // Velger om databasen skal kjøre i ekte eller simuler modus

private:
   bool *sidenErOpptatt;
           // En tabell, med like mange elementer som det er sider i
           // databasen, som brukes til å holde oversikt over hvilke sider
           // som er ledige. Dersom tabellelementet er FALSK er siden ledig,
           // ellers er siden i bruk.

   char *navn;  // Er navnet på databasen og Windowsfila
   int antallSider;  // Angir hvor mange fysiske sider som databasen rommer
   int antallAllokerteSider;

   int antallSiderLest;
   int antallSiderSkrevet;

   HANDLE hFil;   // Håndtak til Windows fila
   int aksesstid;  // Antall millisekund for å overføre ei side fra
                   // databasen til hovedminnet eller omvendt. Dette
                   // inkluderer posisjonerings-, rotasjons- og
                    // overføringstid.
   DatabaseModus modus;  // Kan være ekte eller simuler. I sistnevnte modus
                         // utføres ikke selve den fysiske I/O'en.
};

#endif  // DATABASE_H
```

C.15 databaseV0.cpp

```
// databaseV0.cpp
// Rutiner for å simulere en database. Selve lagringen skjer på ei ordinær
// Windowsfil. Databasen består av et antall sider som kan leses/skrives
// og allokeres/deallokeres.

#include "databaseV0.h"

// Database::Database
// Initialiser en simulert database. Oppretter ei Windowsfil som
// skal brukes til å lagre sidene på, og en allokeringstabell som viser
// om sidene er ledige eller i bruk.
//
// "databasenavn" blir brukt som navn på Windowsfila.
// "antallSiderIDatabase" muliggjør en dymanisk bestemmelse av database-
//                          størrelsen.
```

```
Database::Database(char *databasenavn, int antallSiderIDatabase)
{
   navn = databasenavn;
   hFil = CreateFile(navn,                 /* opprett ei ny fil      */
             GENERIC_READ | GENERIC_WRITE, /* åpne med read-write    */
             0,                            /* ingen deling           */
             (LPSECURITY_ATTRIBUTES) NULL, /* ingen sikkerhet        */
             CREATE_ALWAYS,                /* overskriv eksisterende */
             FILE_ATTRIBUTE_NORMAL |       /* normal fil             */
             FILE_FLAG_NO_BUFFERING ,      /* ingen buffring         */
             (HANDLE) NULL);               /* ingen attr. template   */

   antallSider = antallSiderIDatabase;
   aksesstid = 0;
   antallAllokerteSider = 0;
   antallSiderLest = 0;
   antallSiderSkrevet = 0;

   sidenErOpptatt = new bool[antallSider];
   for (int i = 0; i < antallSider; i++)
      sidenErOpptatt[i] = FALSK;

   // LPCTSTR  zRootPathName,  // address of root path
    DWORD  SectorsPerCluster;  // address of sectors per cluster
    DWORD  BytesPerSector;   // address of bytes per sector
    DWORD  FreeClusters;  // address of number of free clusters
    DWORD  Clusters;   // address of total number of clusters

   GetDiskFreeSpace(
    NULL,  // address of root path
    &SectorsPerCluster,   // address of sectors per cluster
    &BytesPerSector,   // address of bytes per sector
    &FreeClusters,  // address of number of free clusters
    &Clusters // address of total number of clusters
   );

   debug('d', "Database: Bytes per sektor %d\n", BytesPerSector);

   modus = Ekte;
}

// Database::~Database
// Rydder opp ved å lukke fila og slette allokeringstabellen

Database::~Database()
{
   CloseHandle(hFil);
   delete sidenErOpptatt;
}

// Database::LesSide
// Simulerer en database (disk) ved å lese ei side fra ei Windowsfil.
//
// "sidenr" er nummeret på den fysiske sida i databasen
// "data" er adressen til et dataområde i hovedminnet hvor sida skal kopieres
//        til

void Database::LesSide(int sidenr, char *data)
{
   DWORD nBytesRead;
   DWORD antallTicks = GetTickCount();

   assert(sidenr < antallSider);
   debug('d', "Database: Leser side %d fra databasen %s\n", sidenr, navn);
   if (modus == Ekte) {
      SetFilePointer (hFil, sidenr * Sidestoerrelse, NULL, FILE_BEGIN);
      ReadFile(hFil, data, Sidestoerrelse, &nBytesRead, NULL) ;
   }
   antallSiderLest++;
   while (GetTickCount() < (antallTicks + aksesstid))
      ;
}

// Database::SkrivSide
// Simulerer en database (disk) ved å skrive sida til ei Windowsfil.
//
// "sidenr" er nummeret på den fysiske sida i databasen
// "data" er adressen til et dataområde i hovedminnet hvor sida skal kopieres
//        fra

void Database::SkrivSide(int sidenr, char *data)
{
   DWORD n;
   DWORD antallTicks = GetTickCount();

   assert(sidenr < antallSider);
   debug('d', "Database: Skriver side %d til databasen %s\n", sidenr, navn);
   if (modus == Ekte) {
      SetFilePointer(hFil, sidenr * Sidestoerrelse, NULL, FILE_BEGIN);
      WriteFile(hFil, data, Sidestoerrelse, &n, NULL) ;
   }
   antallSiderSkrevet++;
   while (GetTickCount() < (antallTicks + aksesstid))
      ;
}

// Database::AllokerSide
// Finner ei ledig side ved å søke gjennom allokeringstabellen, hvor
// den så blir merket som opptatt. Dersom det ikke er noen ledige sider,
// returneres FALSK.
//
// "sidenr" er nummeret på aktuell side

bool Database::AllokerSide(int &sidenr)
{
  for (int i = 0; i < antallSider; i++) {
      if (!sidenErOpptatt[i]) {
         debug('d', "Database: Sidenr %d er allokert\n", i);
         sidenr = i;
         sidenErOpptatt[i] = SANN;
        antallAllokerteSider++;
         return SANN;
      }
   }
   return FALSK;
}

// Database::DeallokerSide
// Merker sida som ledig i allokeringstabellen.
//
// "sidenr" er nummeret på aktuell side

void Database::DeallokerSide(int sidenr)
{
   assert(sidenr < antallSider);
   sidenErOpptatt[sidenr] = FALSK;
   antallAllokerteSider--;
   debug('d', "Database: Sidenr %d er deallokert\n", sidenr);
}

// Database::AntallSider
// Returnerer antall sider som databasen kan romme

int Database::AntallSider()
{
   return antallSider;
```

```
}

// Database::SettAksesstid
// Setter aksesstiden i millisekund

void Database::SettAksesstid(unsigned int nyAksesstid)
{
   aksesstid = nyAksesstid;
}

// Database::HentStatistikk
// Henter antall leste, skrevne og allokerte sider

void Database::HentStatistikk(int &_antallSiderLest, int &_antallSiderSkrevet,
                                                    int &_antallAllokerteSider)
{
   _antallSiderLest = antallSiderLest;
   _antallSiderSkrevet = antallSiderSkrevet;
   _antallAllokerteSider = antallAllokerteSider;
}

// Database:: NullstillStatistikk
// Nullstiller antall sider lest og skrevet

void Database::NullstillStatistikk()
{
   antallSiderLest = 0;
   antallSiderSkrevet = 0;
}

// Database::SkrivUtStatistikk
// Skriver statistikkinformasjonen til skjermen

void Database::SkrivUtStatistikk()
{
   printf("Database: Sider lest: %d, Sider skrevet: %d, "
             " Allokerte sider: %d (%.0f kb)\n", antallSiderLest,
             antallSiderSkrevet, antallAllokerteSider,
             antallAllokerteSider * Sidestoerrelse / (double) 1024);
}

// Database::VelgModus
// Velger mellom normal og simuler modus. Ved kjøring i sistnevnte modus
// utføres ikke selve den fysiske I/O'en.
//
// "nyttModus" kan være Ekte eller Simuler

void Database::VelgModus(DatabaseModus nyttModus)
{
   assert((nyttModus == Ekte) || (nyttModus == Simuler));
   modus = nyttModus;
}
```

LITTERATURREFERANSER

[AhoHopcroftUllman83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman: **Data Structures and Algorithms.** Addison-Wesley 1983.

[AhoSethiUllman86] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman**: Compilers: Principles, Techniques, and Tools.** Addison-Wesley 1986.

[BancilhonDelobelKanellakis92] Francois Bancilhon, Claude Delobel, and Paris Kanellakis: **BUILDING AN OBJECT-ORIENTED DATABASE SYSTEM, THE STORY OF O_2.** Morgan Kaufmann, 1992.

[Bratbergsengen96] Kjell Bratbergsengen**: Filsystemer, Lagring og behandling av store datamengder.** NTNU 1996.

[Carey...94] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, Michael J. Zwilling: **Shoring Up Persistent Applications.** In Proceedings of ACM-SIGMOD Conference on the Management of Data, May 1994.

[CareyDeWittNaughton94] Michael J. Carey, David J. DeWitt and Jeffrey F. Naughton: **The OO7 Benchmark.** Computer Sciences Department, University of Wisconsin-Madison, 1994.

[CareyDeWittRichardsonShekita86] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita: **Object and File Management in the EXODUS Extensible Database System.** In Proceedings of the Twelfth International Conference on Very Large Data Bases

[Cattell94] R.G.G. Cattell: **Object Data Management, Revised Edition.** Addison-Wesley 1994.

[Cattell96] R.G.G. Cattell: **The Object Database Standard: ODMG - 93, Release 1.2.** Morgan Kaufmann 1996.

[CattellSkeen92] R.G.G Cattell and J. Skeen: **Object Operations Benchmark.** Sun Microsystems. ACM 1992.

[ChouDeWitt86] Hong-Tai Chou and David J. DeWitt: **An Evaluation of Buffer Management Strategies for Relational Database Systems.** In Proceedings of the International Conference on Very Large Data Bases, august 1985.

[CopelandKhoshafianSmithValduriez86] George P. Copeland, Setrag N. Khoshafian, Marc G. Smith, and Patrick Valduriez: **BUFFERING SCHEMES FOR PERMANENT DATA.** Microelectronics and Computer Technology Corporation Austin, Texas 1986.

[DetlefsDosserZorn93] David Detlefs and Al Dosser, Systems Research Center, Digital Equipment Corporation. Benjamin Zorn, Department of Computer Science, University of Colorado**: Memory Allocation Costs in Large C and C++ Programs.** Technical Report 1993.

[EffelsbergHaerder84] Wolfgang Effelsberg and Theo Haerder: **Principles of Database Buffer Management.** In ACM Transactions on Database Systems, Vol. 9, No. 4, December 1984, Pages 560-595

[ElmasriNavathe94] Ramez Elmasri and Shamkant B. Navathe: **Fundamentals of Database Systems, Second Edition.** Benjamin/Cummings. Georgia and Texas 1994.

[HornickZdonik87] Mark F. Hornick and Stanley B. Zdonik: **A Shared, Segmented Memory System for an Object-Oriented Database.** Brown University 1987.

[Humphrey89] Watts S. Humphrey: **Managing the Software Process.** Addison-Wesley 1989.

[Håndbok96] Rolv Bræk, Jacob Hygen, Sverre Høysæter, Trond Johansen og Per Scott: **Håndbok i systemarbeid.** Tapir forlag. Norge 1994.

[KemperKossmann92] Alfons Kemper and Donald Kossmann: **Dual-Buffering Strategies in Object Bases.** In Proceedings of the International Conference on Very Large Data Bases, 1993.

[KerninghanRitchie88] Brian W. Kernighan and Dennis M. Ritchie**: The C Programming Language**, **Second Edition.** Prentice Hall 1988.

[Knuth73] D. E. Knuth: **The Art of Computer Programming Vol. III: Sorting and Searching.** Addison-Wesley 1973.

[LefflerMcKusickKarelsQuarterman] Samuel J. Leffler, Marshall Kirk McKusick, and John S. Quarterman: **The Design and Implementation of the 4.3BSD UNIX Operating System.** Addison-Wesley 1989.

[LervikLjosland94] Else Lervik og Mildrid Ljosland: **Grunnleggende programmering i C++.** Ad Notam Gyldendal 1993.

[LervikLjosland94] Else Lervik og Mildrid Ljosland: **Objektorientert programmering i C++.** Ad Notam Gyldendal 1994.

[MossSinofsky87] J. Eliot B. Moss and Steven Sinofsky**: Managing Persistent Data with Mneme: Designing a Reliable, Shared Object Interface.** In Proceedings of the International Workshop on Object-Oriented Database Systems, 1988.

[NittelDittrich95] Silvia Nittel and Klaus R. Dittrich: **A Storage Server for the Efficient Support of Complex Objects.** In Proceedings of the International Workshop on Persistent Object Systems, 1996.

[NørvågBratbergsengen97] Kjetil Nørvåg and Kjell Bratbergsengen: **Write Optimized Objec-Oriented Database Systems.** In Proceedings of SCCC, 1997.

[OrfaliHarkeyEdwards96] Robert Orfali, Dan Harkey, and Jeri Edwards: **The Essential Distributed Objects Survival Guide.** John Wiley & Sons 1996.

[RosenblumOusterhout92] Mendel Rosenblum and John K. Ousterhout: **The Design and Implementation of a Log-Structured File System.** In Proceedings of the Thirteenth ACM Symposium on Operating System Principles, 1991

[Schildt94] Herbert Schildt: **Teach yourself C++, Second Edition.** Osborne McGraw-Hill 1994.

[SeltzerBosticMcKusickStaelin93] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin**: An Implementation of a Log-Structured File System for UNIX**. In 1993 Winter USENIX - January 25-29, San Diego.

[SilberschatzGalvin94] Abraham Silberschatz and Peter B. Galvin: **OPERATING SYSTEM CONCEPTS, FOURTH EDITION.** Addison-Wesley 1994.

[SinghalKakkadWilson92] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson: **Texas: An Efficient, Portable Persistent Store.** In Proceedings Fifth Int'l. Workshop on Persistent Object Systems, San Miniato, Italy, September 1992.

[Sommerville95] Ian Sommerville: **Software Engineering, Fifth Edition.** Addison-Wesley. Lancaster 1995.

[StubbsWebre89] Daniel F. Stubbs and Neil W. Webre**: Data Structures with Abstract Data Types and Pascal, Second Edition.** Brooks/Cole 1989.

[WilsonJohnstoneNeelyBoles95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles: **Dynamic Storage Allocation: A Survey and Critical Review.** In Proceedings 1995 Int'l Workshop on Memory Management, Kinross, Scotland, UK, September 27-29, 1995, Springer Verlag.

STIKKORDREGISTER

N

O

P

R

S

T

U

V

W