

Cache-Oblivious Algorithms in Practice

Speciale af
Jesper Holm Olsen &
Søren Christian Skov

18. december 2002

Program

- Historisk udvikling
- Ideal-cache modellen
- To cache-oblivious prioritetskøer
- Begrænset adresserum
- Performance af de to prioritetskøer
- Konklusion

Historisk udvikling

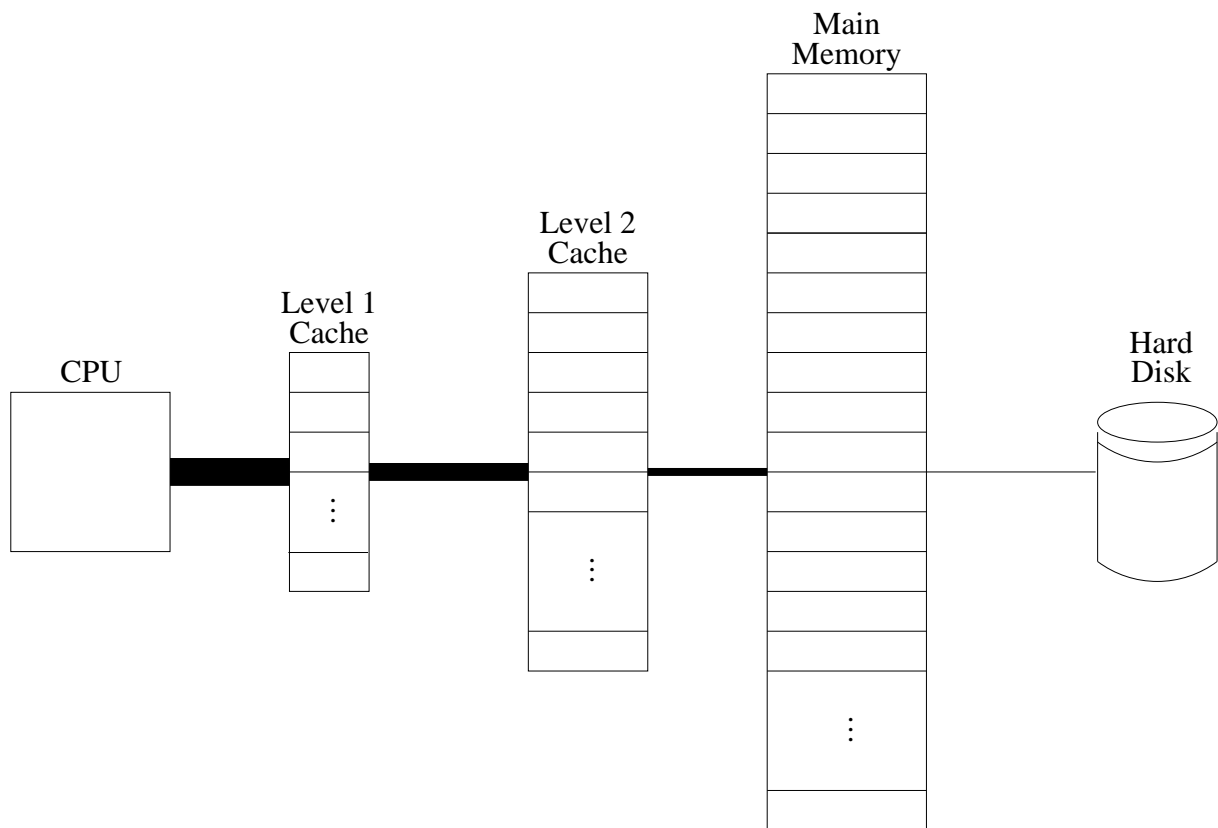
Trend:

- CPUerne bliver ca 55% hurtigere hvert år.
- Memory bliver ca 7% hurtigere hvert år.

Dermed er udvikling fra 1980 til i dag:

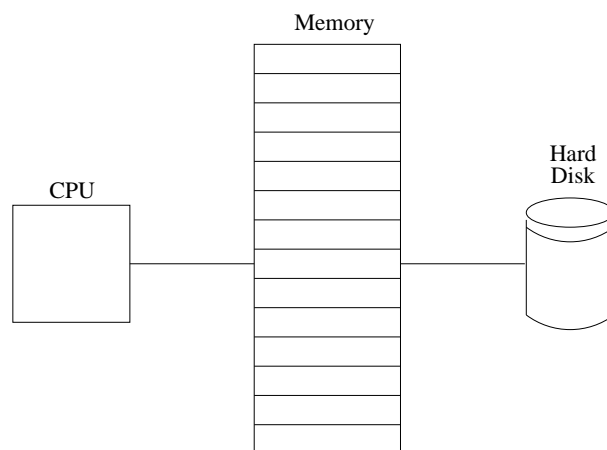
- CPUerne er ca 8000 gange hurtigere.
- Memory er kun ca 4–5 gange hurtigere.

Et typisk hukommelseshierarki



External-memory teorien

- Svar på den meget store latenstid på tilgangen til disk.
- Er bl.a. beskrevet af Floyd i 1972.
- I 1988 laver Aggarvald og Vitter en formaliseret model.
- Beregning af I/O kompleksitet, hvor antallet af læsninger af skrivninger til disken tælles.



Beregning af I/O kompleksitet

Til beregningen af I/O kompleksiteten bruges konstanterne:

N antal elementer der skal håndteres.

M antal elementer der kan være i memory.

B antal elementer der kan være i en blok.

Til beregning af I/O kompleksitet bruges:

scan(N) Det antal I/Os der forekommer ved et sekventielt scan af N elementer = $\Theta\left(\frac{N}{B}\right)$.

sort(N) Det optimale antal I/Os der forekommer ved sortering af N elementer = $\Theta\left(\frac{N}{B} \log_{M/B}\left(\frac{N}{B}\right)\right)$.

Cache-oblivious teorien

Cache-oblivious teorien blive første gang beskrevet af Prokop i 1999.

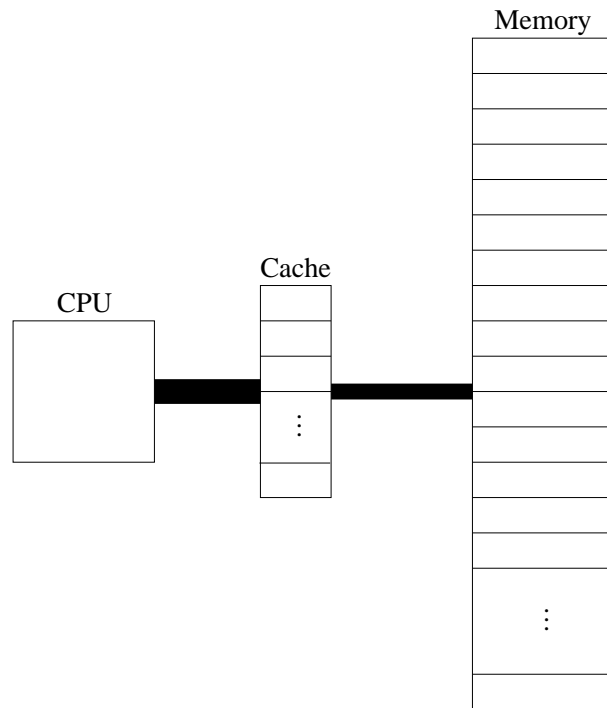
Definition:

“An algorithm is *cache oblivious* if no program variables dependent on hardware configuration parameters, such as cache size (M) and cache-line length (B), need to be tuned to minimize the number of cache misses.”

Mål: At udnytte alle levels optimalt.

Fordele: Gør det nemmere at designe og implementere algoritmer, samt at portere dem.

Ideal-cache modellen



Konceptuelt en uendelig stor *memory* og en *cache* af størrelse M delt ind i cache-linjer af størrelse B (som dog er ukendte).

Som i external-memory bruges også her I/O-komplexitet, f.eks. $sort(N)$ og $scan(N)$.

Fire antagelser om ideal-cache modellen

1. *“Høj” cache: $M = \Omega(B^2)$*

Bruges i analysen af flere algoritmer. Er realistisk i forhold til moderne computere (konstanten svinger mellem 2–1024).

2. *Optimal udskiftningsstrategi*

Ikke mulig i praksis. Sleator & Tarjan viser dog, at for en dobbelt så stor M vil LRU maksimalt foresage dobbelt så mange misses. Asymptotisk er dette ækvivalent (i praksis tilnærmes LRU dog ofte.)

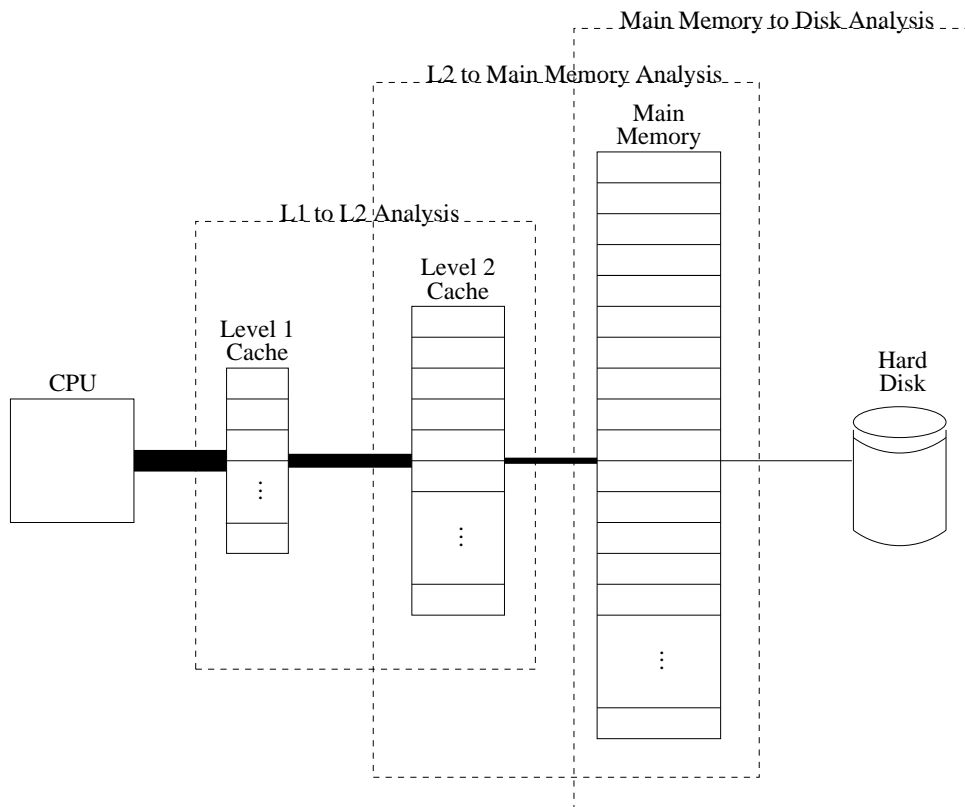
3. *Automatisk udskiftning*

Dette sker allerede i hardware, samt i virtual-memory og er derfor realistisk.

4. *Fuld associativitet*

Ikke realistisk i praksis, men bekvem.

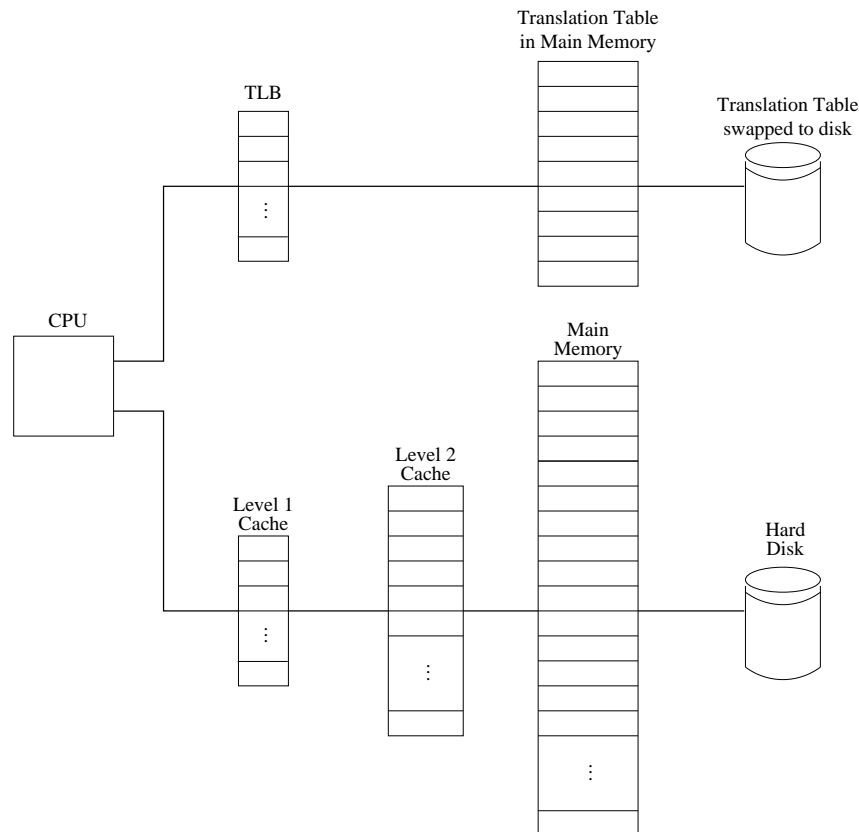
Generalisering til flere niveauer



- Ideal-cache modellen kender ikke størrelser
- Hver cache-level er *inclusive*

Virtual-memory

Det almindelige og det virtuelle hukommelseshierarki kan anskues separat ved følgende abstraktion:



$$\begin{aligned} \text{Tid} &= \text{total omkostning i hukommelseshierakiet} \\ &+ \text{total omkostning i virtual-memory} \end{aligned}$$

TLB kan ses som en cache for *translation table* med $B = 1$ page-translation pr. indgang.

Cache-oblivious teknikker

Pt. er følgende teknikker blevet brugt:

- *Sekventiel datatilgang*

F.eks. scanning af et array = $\Theta\left(\frac{N}{B}\right)$

- *Divide-and-conquer*

På et tidspunkt bliver problemet mindre end M og yderligere neddelinger er “gratis” i henhold til I/O.

- *Rekursiv layout (“van Emde Boas”)*

En statisk datastruktur, f.eks. et binært træ, kan placeres i hukommelsen således at almindelige søgninger udnytter cache optimalt.

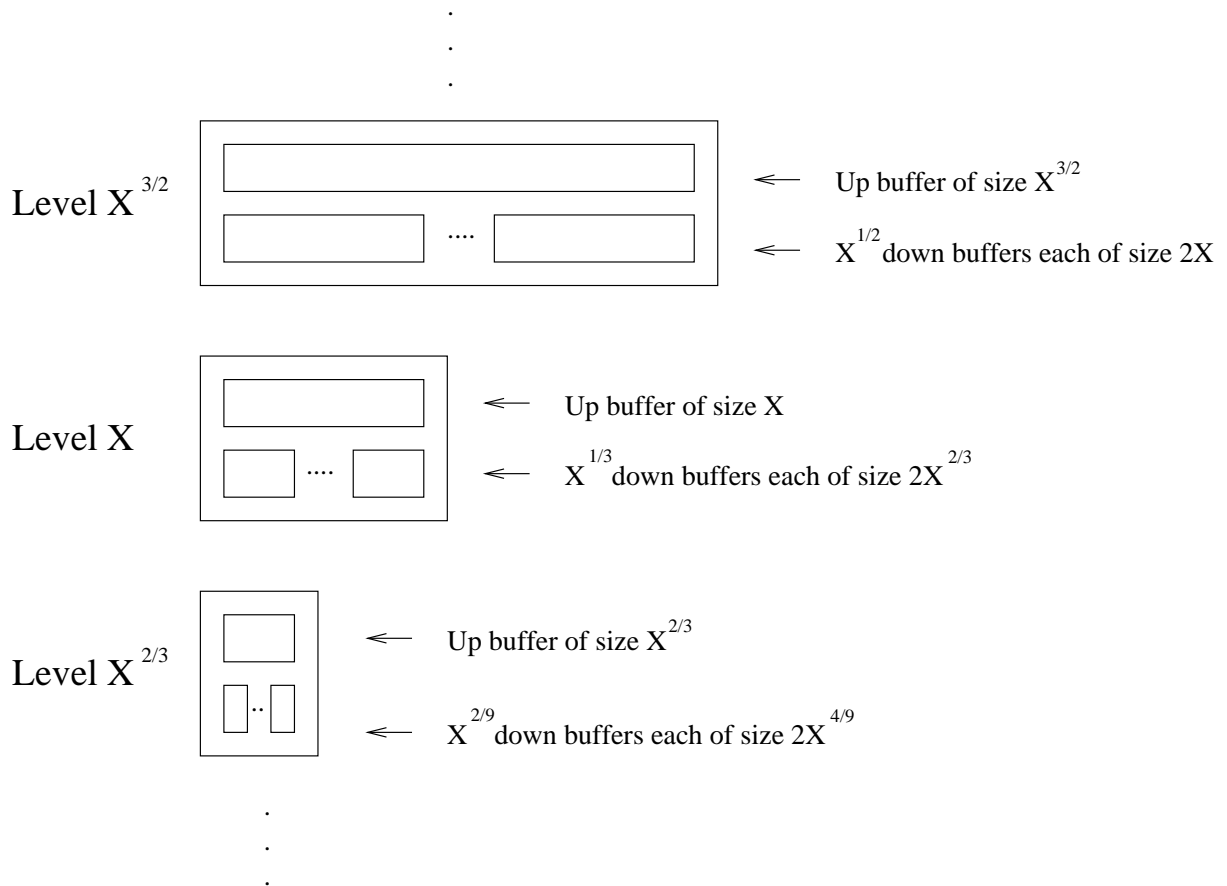
- *“Lazy” evaluering via buffere*

Ved brug af buffere kan man opnå øget sekventiel datatilgang ved kun at tømme buffere, når de er fyldte. Ved at lade bufferne vokse, vil de undervejs passe i forskellige størrelse af M .

Distribution Heap

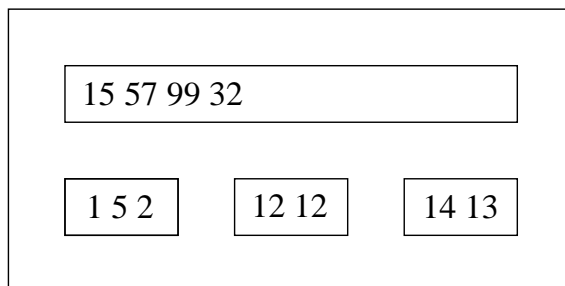
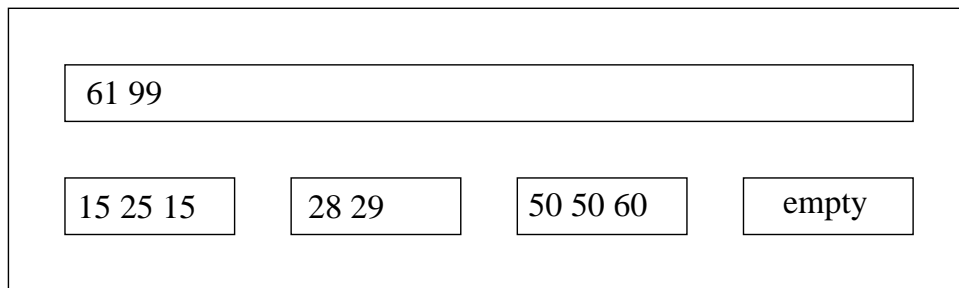
- Cache-Oblivious data-struktur beskrevet af Arge et al. i 2002.
- Består af en række “levels”, hvor den største er af størrelse N .
- De resterende levels størrelser er defineret rekursivt, sådan at det næste level er af størrelse $N^{2/3}$.
- Samlet betyder det at de forskellige levels har størrelserne:
$$N, N^{2/3}, \dots, X^{3/2}, X, X^{2/3}, X^{4/9}, \dots$$

Struktur



Der bruges to operationer: Push og Pull.

Eksempel



Global rebuilding

Der foretages en global rebuilding efter hver $N/2$ operationer.

Metode:

- Alle elementer tages ud.
- Den gamle Distribution Heap nedlægges.
- En ny Distribution Heap oprettes.
- Alle elementer indsættes igen.

Kompleksitet

En Distribution Heap har optimal amortiseret I/O-, arbejde- og plads-kompleksitet:

$$\mathbf{I/O} \quad \Theta\left(\frac{N}{B} \log_{M/B}\left(\frac{N}{B}\right)\right) = \text{sort}(N)$$

$$\mathbf{Arbejde} \quad O(N \log N)$$

$$\mathbf{Plads} \quad O(N)$$

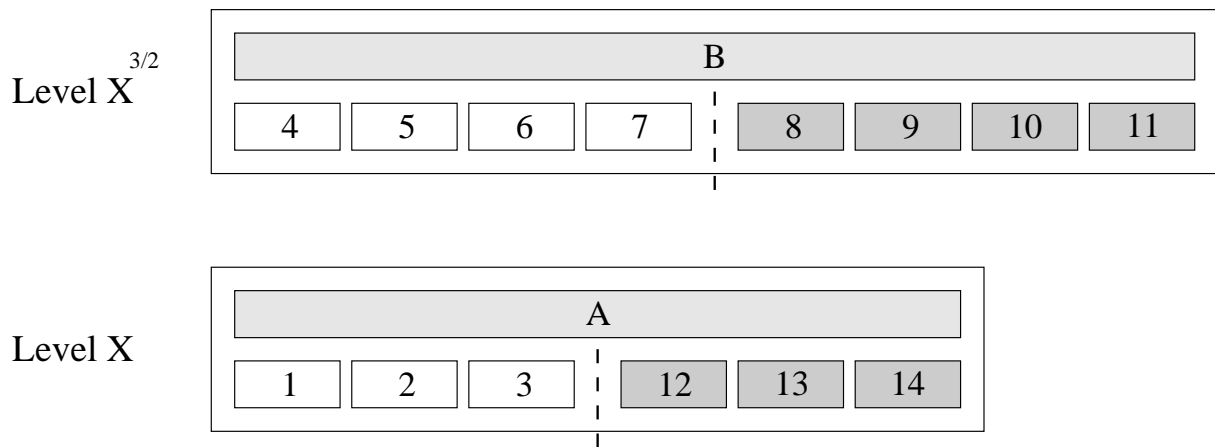
DIPD

Prioritets Deque er en data-struktur der understøtter at man kan udtage både det mindste og det største element fra den same struktur.

Inspireret af den førnævnte Distribution Heap og en prioritets deque kaldet Interval Heap, har vi udviklet en cache oblivious prioritets deque.

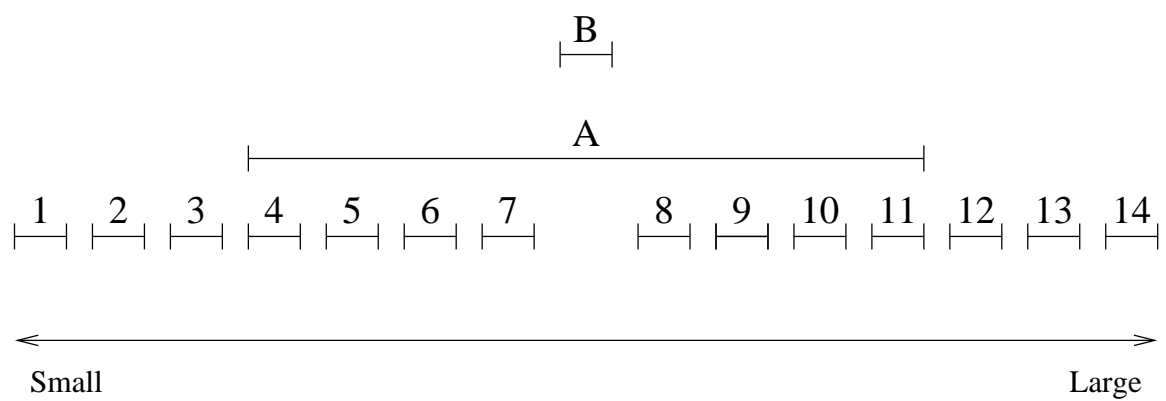
Vi kalder den: Distribution Interval Priority Deque (eller: DIPD).

DIPD Struktur



- Antallet af downbuffere på hvert levels fordobles.
- Halvdelen er *minimum* downbuffere, og halvdelen er *maximum* downbuffere.

Intervaller



Kompleksitet

Den amortiserede kompleksitet af en DIPD er:

$$\mathbf{I/O} \quad \Theta \left(\frac{N}{B} \log_{M/B} \left(\frac{N}{B} \right) \right) = \text{sort}(N)$$

$$\mathbf{Arbejde} \quad O(N \log N)$$

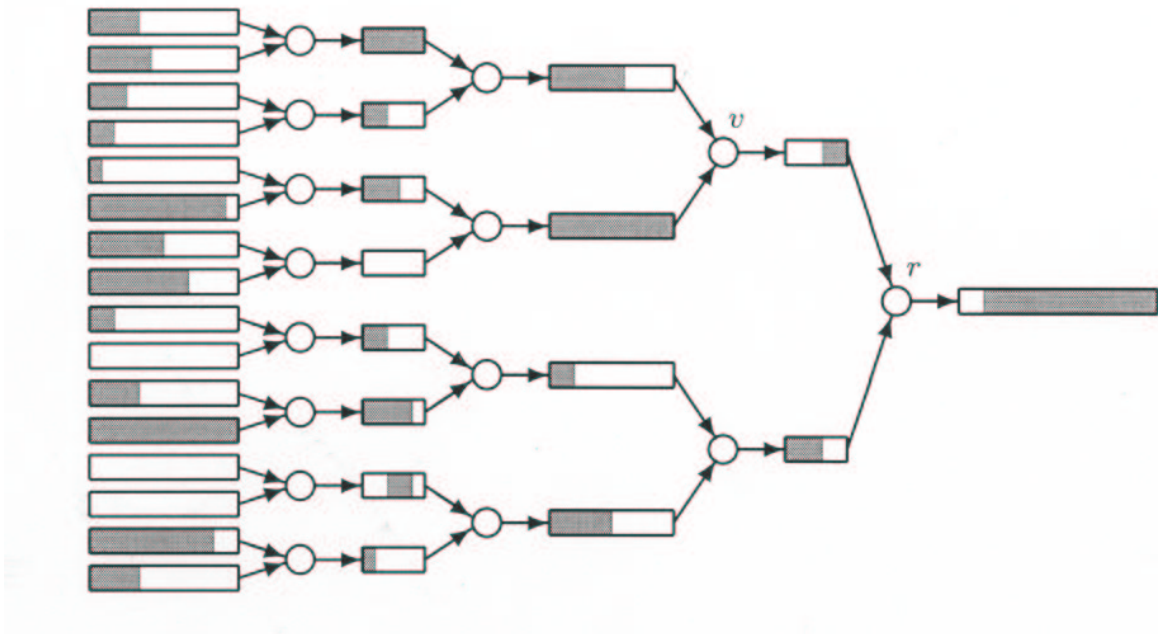
$$\mathbf{Plads} \quad O(N)$$

Dette er optimalt for en prioritets deque.

Funnel-heap

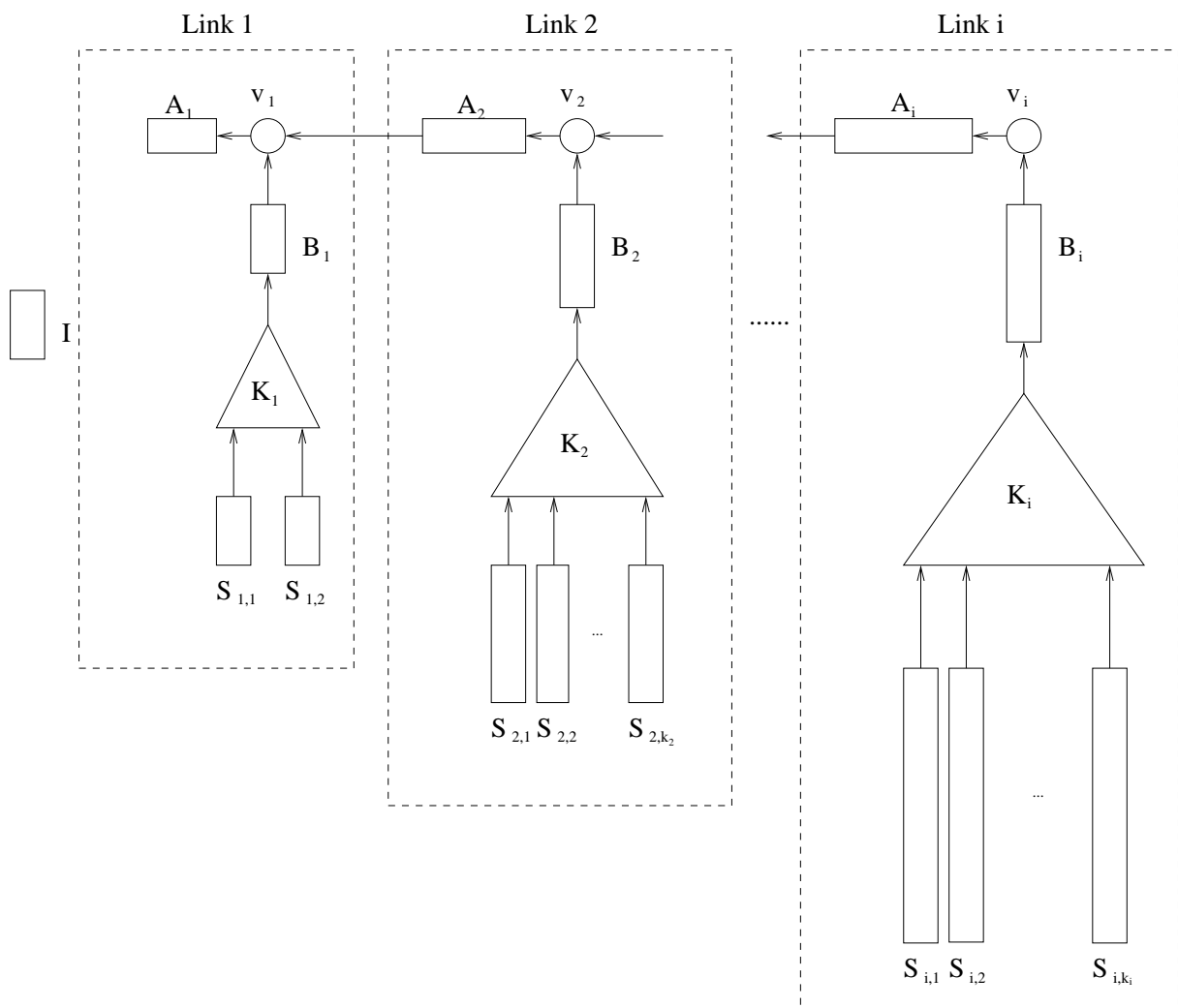
Introduceret af Brodal & Fagerberg i 2002.

En *funnel* består af *buffere* og *binære mergere*:



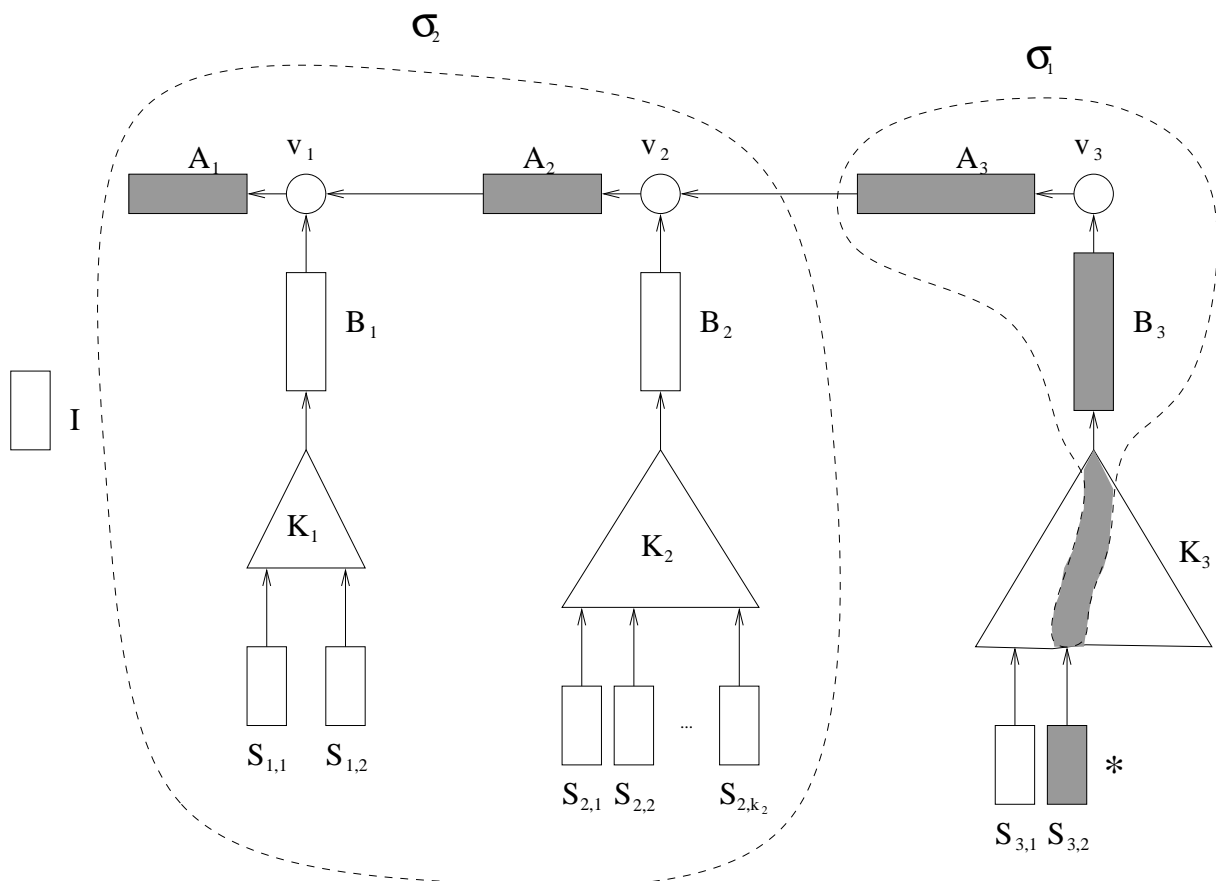
Denne kan bruges til f.eks. cache-oblivious sortering.

En Funnel-Heap er et (ubalanceret) træ af *buffere*, *funnels* og *binære mergere* sat sammen i *links*:



Operationer

- Extract & Insert
- “Sweep” *merger* elementer og flytter dem til et større link:
 1. Vælg den første tomme S-buffer
 2. Lav σ_1 og σ_2 som *merges* til σ .
 3. Indsæt elementer på stien fra roden til den tomme S-buffer



Kompleksitet

$$\mathbf{I/O} \quad \Theta \left(\frac{N}{B} \log_{M/B} \left(\frac{N}{B} \right) \right) = \mathit{sort}(N)$$

$$\mathbf{Arbejde} \quad O(N \log N)$$

$$\mathbf{Plads} \quad O(N)$$

I den publicerede artikel bruger datastrukturen mere end lineær plads. Vi modificerer den til, på bekostning af sekventiel allokering, at allokere S-buffere efter behov. Derved opnås $O(N)$.

Begrænset adresserum

De fleste computere der bruges i dag er 32-bit computere.

- Det betyder at man “kun” kan adressere op til 2^{32} bytes, hvilket svarer til 4GB.
- Disse 4GB skal dog bruges til alt det en proces skal have adgang til, det indbefatter altså også programkoden, kernen m.v.
- På en 32-bit computer med Linux betyder det at selve data er begrænset til ca. 2 GB.

Begrænset adresserum i cache-oblivious algoritmer

Cache-oblivious algoritmer bruger hukommeshierarkiet, og allokerer *hele* datamængden i adresserummet.

Fordele

- Programmøren skal ikke bekymre sig om at flytte data mellem forskellige hierarkier.
- Implementationerne kan uden ændringer flyttes mellem forskellige computere.

Ulempen er at cache-oblivious algoritmer dermed er begrænset af adresserummets størrelse!

Eksempel: En Distribution Heap

Det værste adresserums-forbrug forekommer under global rebuilding af en voksende Distribution Heap.

En Distribution Heap kan maksimalt bruge $1/4$ af den allokerede plads.

På en 32-bit computer med Linux betyder dette, at man kun kan arbejde med op til 512MB data i en Distribution Heap.

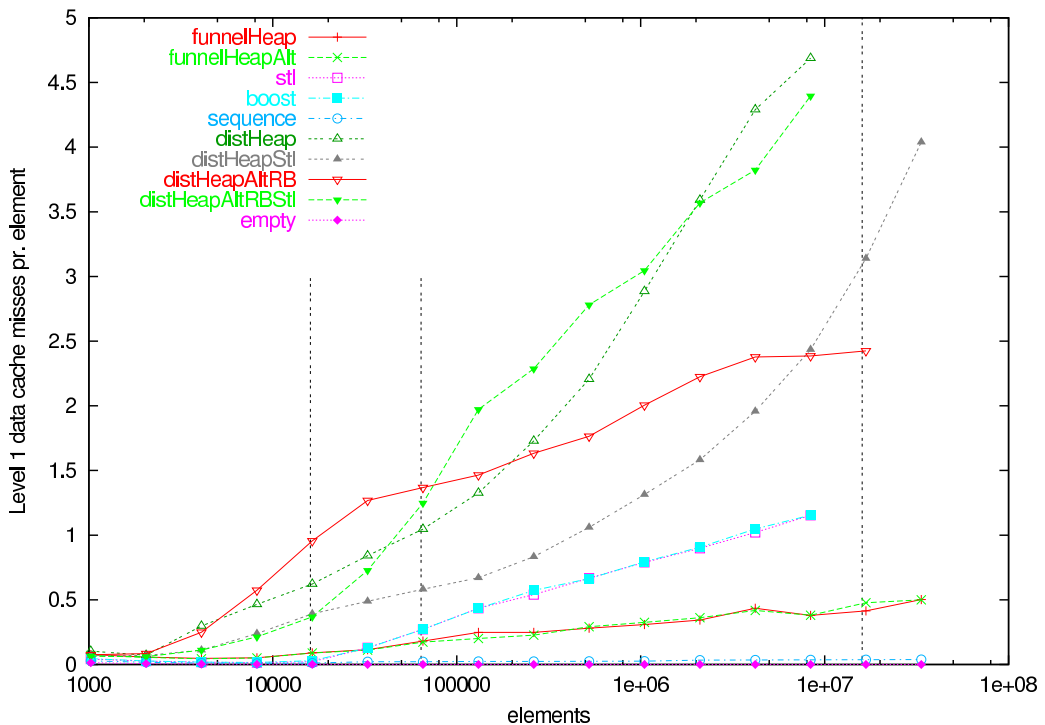
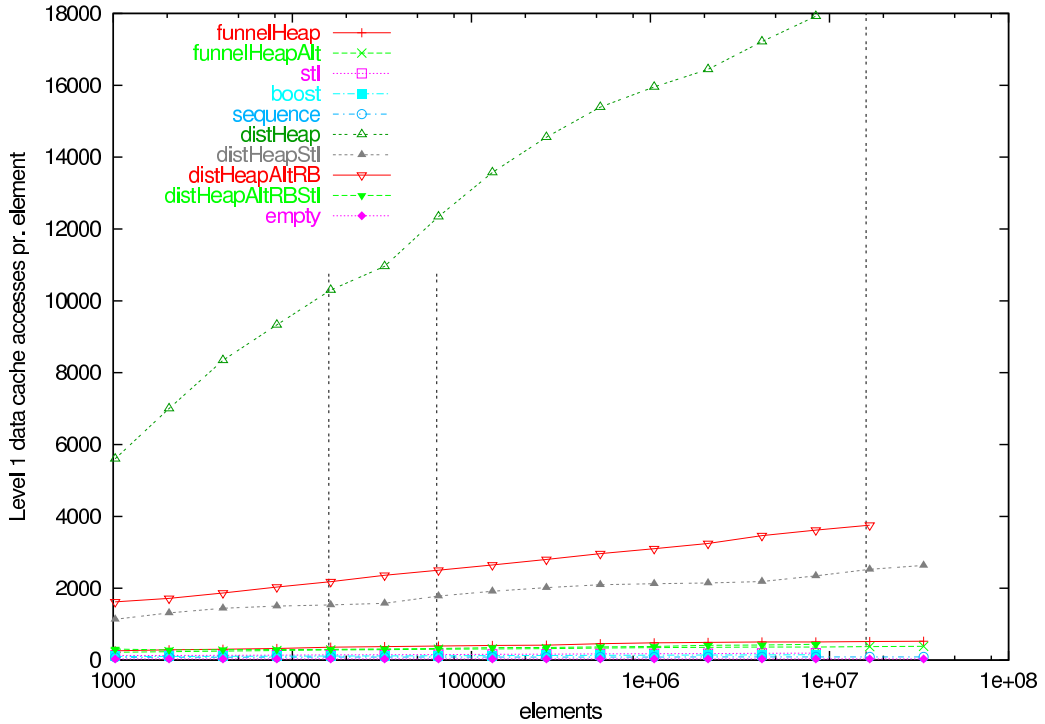
Benchmarks

- Hver heap er implementeret med forskellige sortingsmetoder og andre modifikationer.
- Konkurrenter: RAM, external-memory og cache-aware
- Tid, PAPI, page-faults

Er cache-oblivious ideen så konkurrencedygtig i praksis?

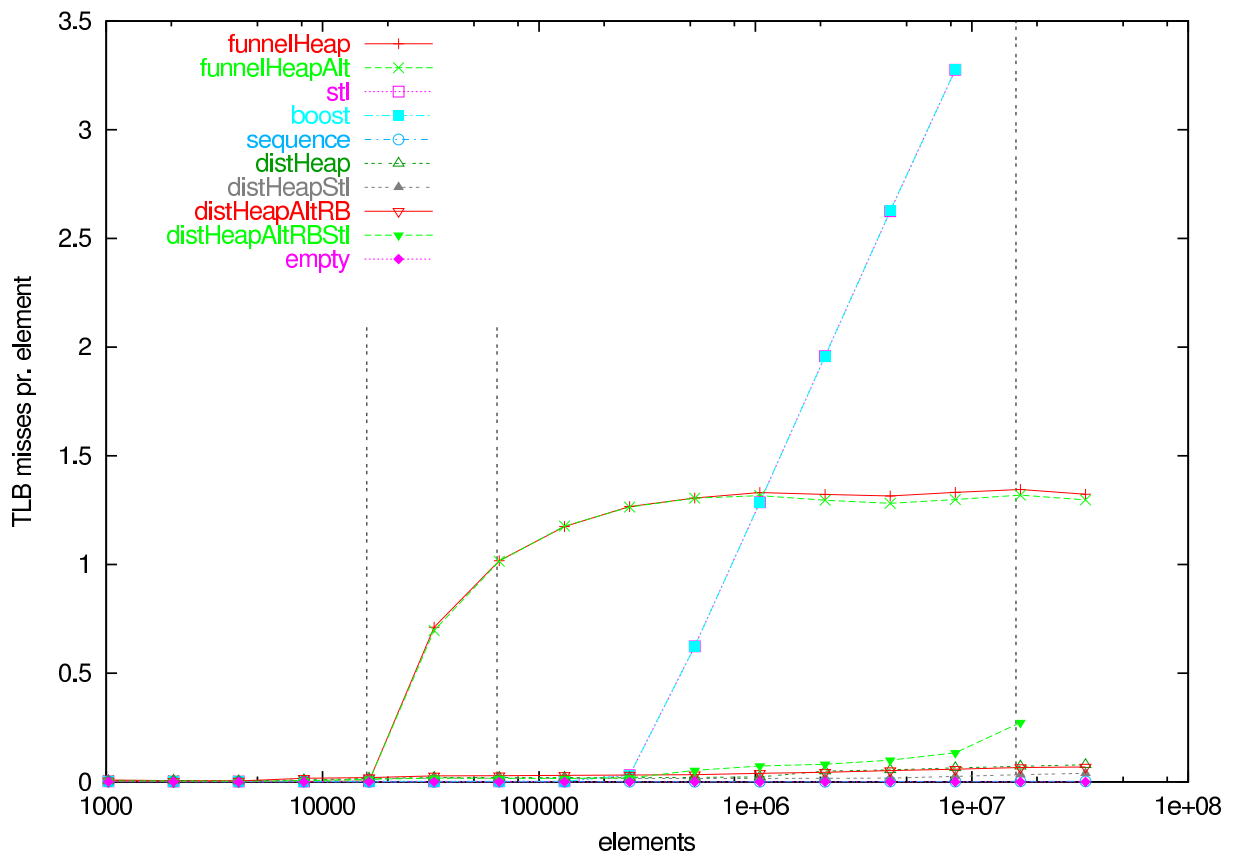
Arbejde vs. cache misses

L1 accesses vs. L1 misses for lang sekvens



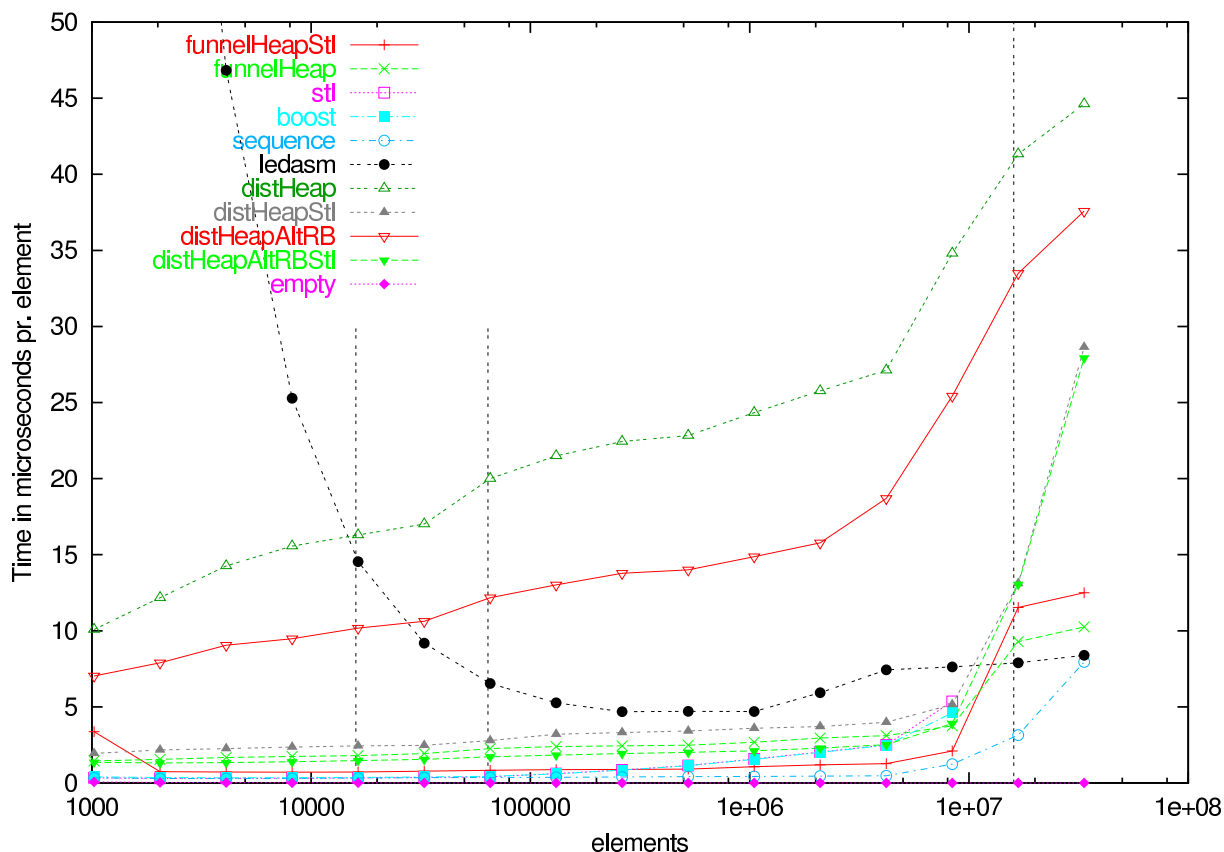
Virtual-memory

TLB misses for monotont voksende sekvens:



De to RAM algoritmer foresager mange TLB misses, mens algoritmer der optimerer til cache (sekventiel datatilgang) er klart bedre.

Tid for prioritetskø-sortering



Cache-oblivious kan følger med, når datamængden er større end main memory, men external-memory og cache-aware er bedre.

Opsummering af benchmarks

For de to cache-oblivious algoritmer, som vi har kigget på, er det ekstra arbejde større end gevinsten – ihvertfald på de små niveauer i hukommelseshierarkiet. Cache-aware vinder pga. mindre overhead.

Konklusion

I dette speciale:

- I. Diskuterer vi sammenhængen mellem antagelserne i ideal-cache modellen og virkelighedens computere.
- II. Viser vi at optimal cache-oblivious algoritmer også anvender virtuel-memory systemet optimalt.
- III. Behandler og løser vi praktiske problemer i forbindelse med implementeringen af de to prioritetskø algoritmer.
- IV. Udvikler vi en cache-oblivious optimal prioritets deque.
- V. Beskriver vi problemet med begrænset adresserum.
- VI. Sammenligner vi performance af forskellige prioritetskøer.

Resultater

- Vi viser at det meste af teorien kan overføres til praksis.
- Vi påpeger, at der er praktiske problemer – især med det begrænsede adresserum.
- Vores benchmarks viser vigtigheden af, at algoritmerne også skal minimere det arbejde der skal udføres af CPUen.

PAUSE