1 Arrays, Verkettete Listen und abgeleitete Datenstrukturen

Bounded Arrays

Eingebaute Datenstruktur.

Größe muss von Anfang an bekannt sein

Unbounded Array

z.B. `std::vector`

pushBack: Element anhängen

popBack: Letztes Element löschen

Idee: verdopple wenn der Platz ausgeht

halbiere wenn Platz verschwendet wird

Wenn man das richtig macht, brauchen

n pushBack/popBack Operationen Zeit $O(n)$

Algorithmese: pushBack/popBack haben konstante amortisierte Komplexität Was kann man falsch machen?

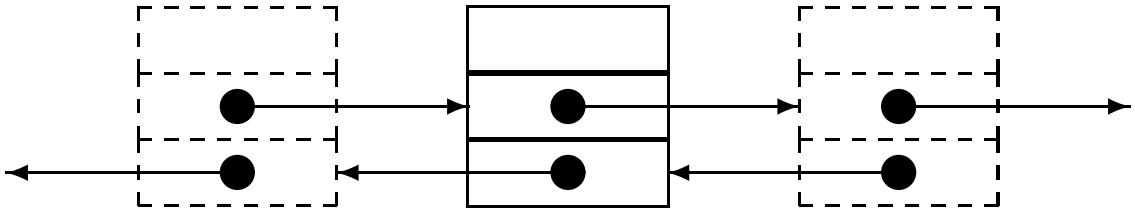
Doppelt verkettete Listen

Class Item **of** Element // one link in a doubly linked list

e : Element

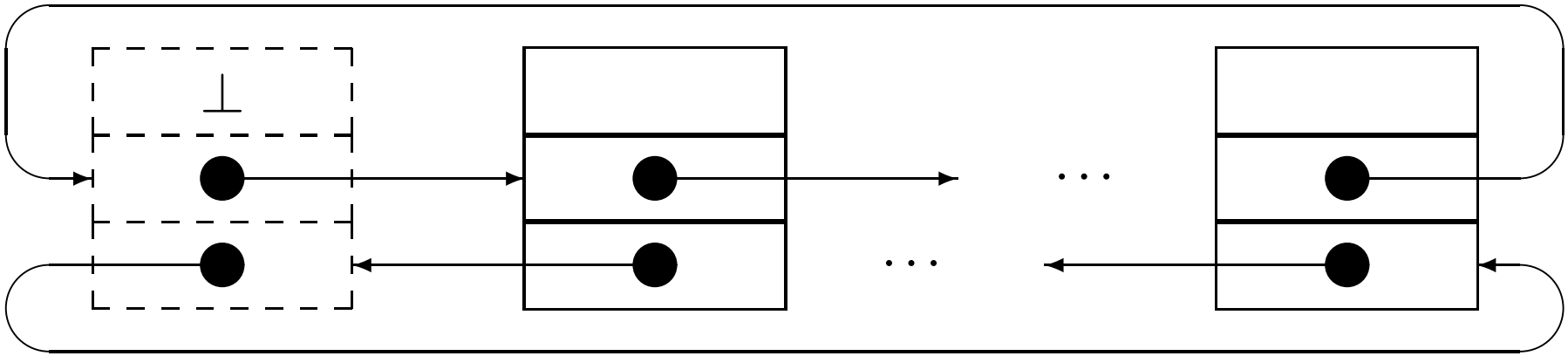
next : Handle //

prev : Handle



invariant next→prev=prev→next=**this**

Trick: Use a dummy header



Procedure splice(a,b,t : Handle)

assert b is not before $a \wedge t \notin \langle a,\ldots,b \rangle$

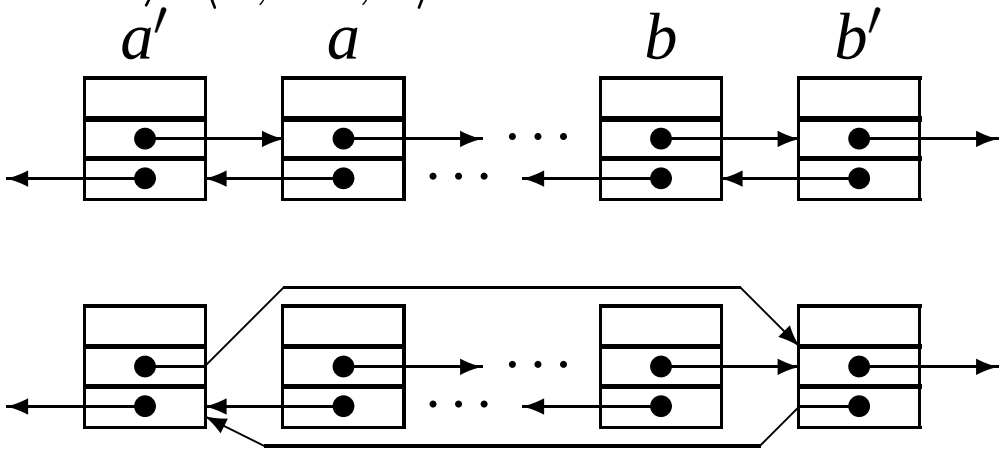
//Cut out $\langle a,\ldots,b \rangle$

a' := a→prev

b' := b→next

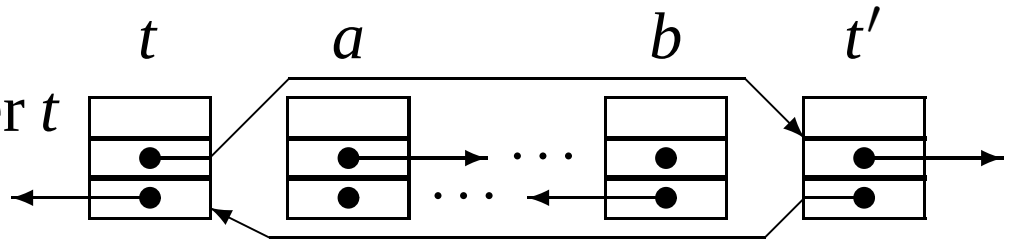
a' →next := b' //

b' →prev := a' //



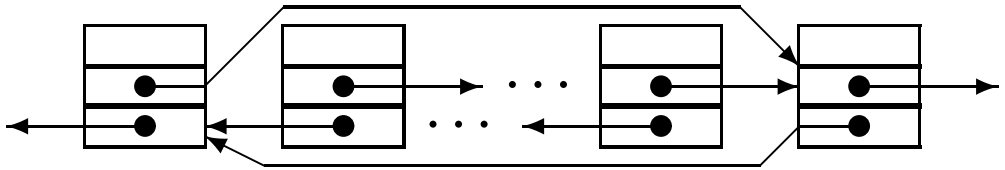
//insert $\langle a,\ldots,b \rangle$ after t

t' := t→next //



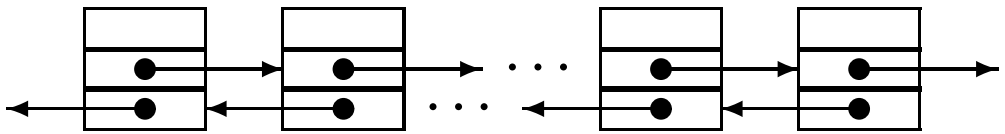
b→next := t' //

a→prev := t //

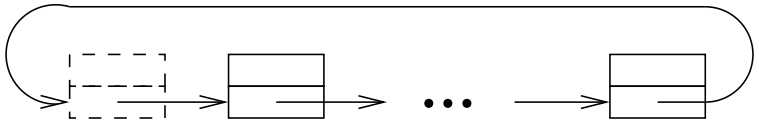


t →next := a //

t' →prev := b //



Einfach verkettete Listen



Vergleich mit doppelt verketteten Listen

- Weniger Speicherplatz
- Platz ist oft auch Zeit
- Eingeschränkter z.B. kein delete
- Merkwürdige Benutzerschnittstelle, z.B. deleteAfter

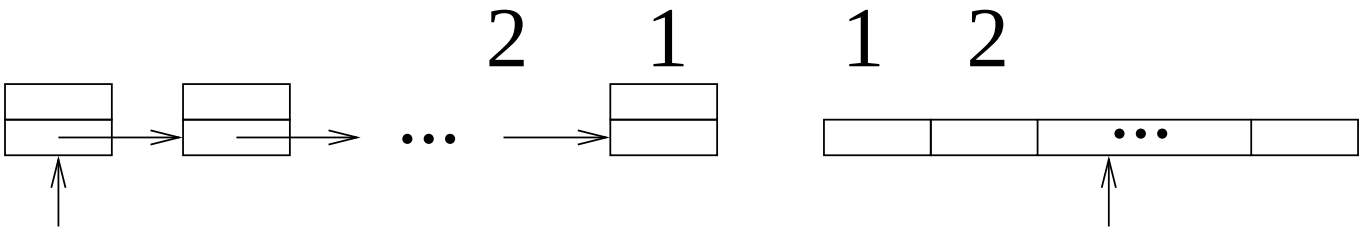
Speicherverwaltung für Listen

- kann leicht 90 % der Zeit kosten!
- Lieber Elemente zwischen (Free)lists herschieben als echte `mallocs`
- Alloziere viele Items gleichzeitig
- Am Ende alles freigeben?
- Speichere „parasitär“. z.B. Graphen:
  Knotenarray. Jeder Knoten speichert ein ListItem
  $\rightsquigarrow$ Partition der Knoten kann als verkettete Listen gespeichert werden
  $\rightsquigarrow$ MST, shortest Path

Challenge: garbage collection, viele Datentypen

$\rightsquigarrow$ auch ein Software Engineering Problem hier nicht

Beispiel: Stack

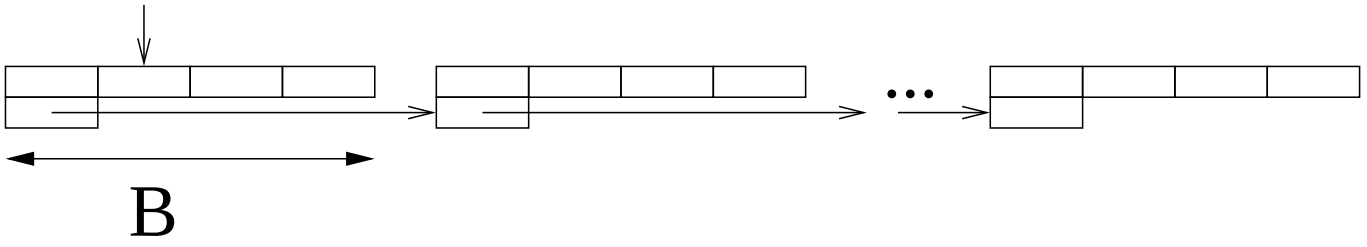


	SList	B-Array	U-Array
dynamisch	+	–	+
Platzverschwendung	pointer	zu groß?	zu groß?
	freigeben?		
Zeitverschwendung	cache miss	+	umkopieren
worst case time	(+)	+	–

Wars das?

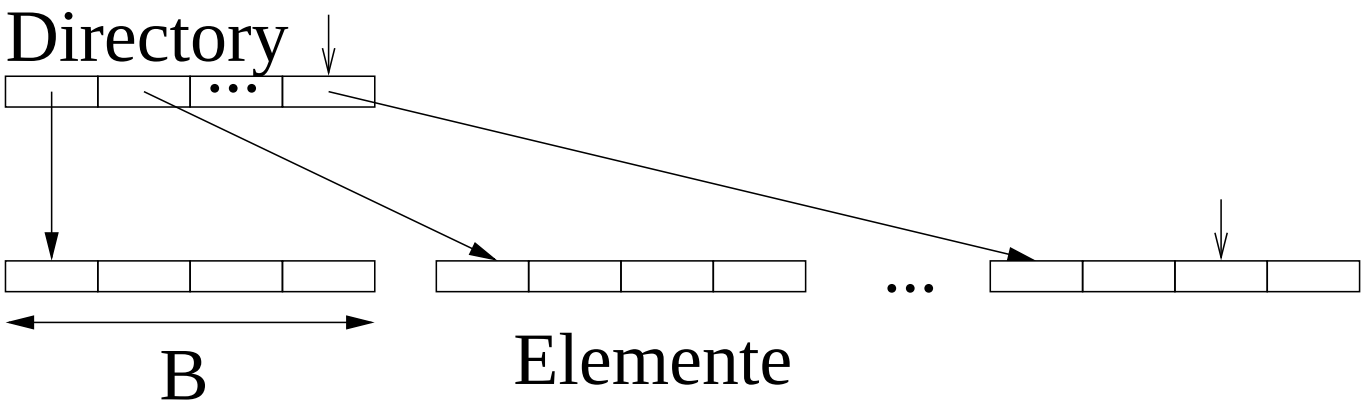
Hat jede Implementierung gravierende Schwächen?

The Best From Both Worlds



	hybrid
dynamisch	+
Platzverschwendung	$n/B + B$
Zeitverschwendung	+
worst case time	+

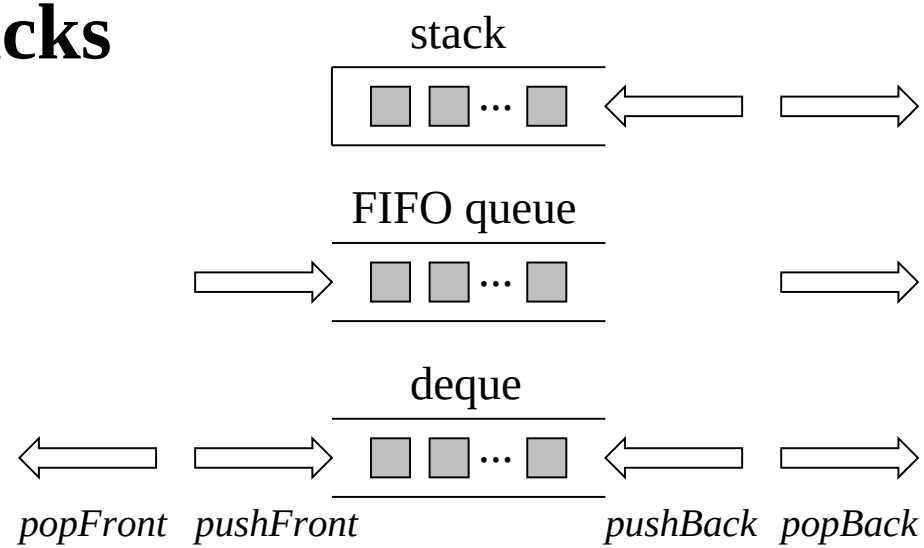
Eine Variante

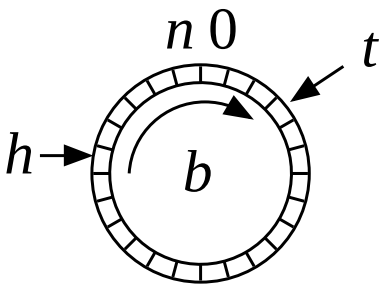


– Reallozierung im top level $\rightsquigarrow$ nicht worst case konstante Zeit

\+ Indizierter Zugrif auf $S[i]$ in konstanter Zeit

Beyond Stacks





FIFO: BArray $\longrightarrow$ cyclic array

Aufgabe: Ein Array, das “[i]” in konstanter Zeit und einfügen/löschen von Elementen in Zeit $O(\sqrt{n})$ unterstützt

Aufgabe: Ein externer Stack, der n push/pop Operationen mit $O(n/B)$ I/Os unterstützt

Aufgabe: Tabelle für hybride Datenstrukturen vervollständigen

Operation	List	SList	UArray	CArray	explanation of '*'
$[\cdot]$	n	n	1	1	
$\lvert\cdot\rvert$	1^*	1^*	1	1	not with inter-list splice
first	1	1	1	1	
last	1	1	1	1	
insert	1	1^*	n	n	insertAfter only
remove	1	1^*	n	n	removeAfter only
pushBack	1	1	1^*	1^*	amortized
pushFront	1	1	n	1^*	amortized
popBack	1	n	1^*	1^*	amortized
popFront	1	1	n	1^*	amortized
concat	1	1	n	n	
splice	1	1	n	n	
findNext,...	n	n	n^*	n^*	cache efficient

Was fehlt?

Fakten Fakten Fakten

Messungen für

- Verschiedene Implementierungsvarianten
- Verschiedene Architekturen
- Verschiedene Eingabegrößen
- Auswirkungen auf reale Anwendungen
- Kurven dazu
- Interpretation, ggf. Theoriebildung

Aufgabe: Array durchlaufen versus zufällig allozierte verkettete Liste

Quicksort

Function quickSort(s : Sequence **of** Element) : Sequence **of** Element

if $|s| \leq 1$ **then return** s // base case

pick $p \in s$ uniformly at random // pivot key

a := $\langle e \in s : e < p \rangle$

b := $\langle e \in s : e = p \rangle$

c := $\langle e \in s : e > p \rangle$

return concatenate(quickSort(a), b, quickSort(c))

Engineering Quicksort

- array
- 2-Wege-Vergleiche
- sentinels für innere Schleife
- inplace swaps
- Rekursion auf kleinere Teilproblemgröße
  $\rightarrow O(\log n)$ zusätzlichen Platz
- Rekursionsabbruch für kleine (20–100) Eingaben, insertion sort
  (nicht ein großer insertion sort)

Procedure qSort(a : **Array of** Element; $\ell, r : \mathbb{N}$) // Sort $a[\ell..r]$
  while $r - \ell \geq n_0$ **do** // Use divide-and-conquer
    j := pickPivotPos(a, l, r)
    swap$(a[\ell], a[j])$ // Helps to establish the invariant
    $p := a[\ell]$
    $i := \ell$; $j := r$
    repeat // a: ℓ $i\rightarrow$ $\leftarrow j$ r
      while $a[i] < p$ **do** i++ // Scan over elements (A)
      while $a[j] > p$ **do** j−− // on the correct side (B)
      if $i \leq j$ **then** swap$(a[i], a[j])$; i++ ; j−−
    until $i > j$ // Done partitioning
    if $i < \frac{l+r}{2}$ **then** qSort(a,ℓ,j); $\ell := j + 1$
    else qSort(a,i,r) ; $r := i - 1$
  insertionSort$(a[l..r])$ // faster for small $r - l$

Picking Pivots Painstakingly — Theory

probabilistically: Expected $1.4n \log n$ element comparisons

median of three: Expected $1.2n \log n$ element comparisons

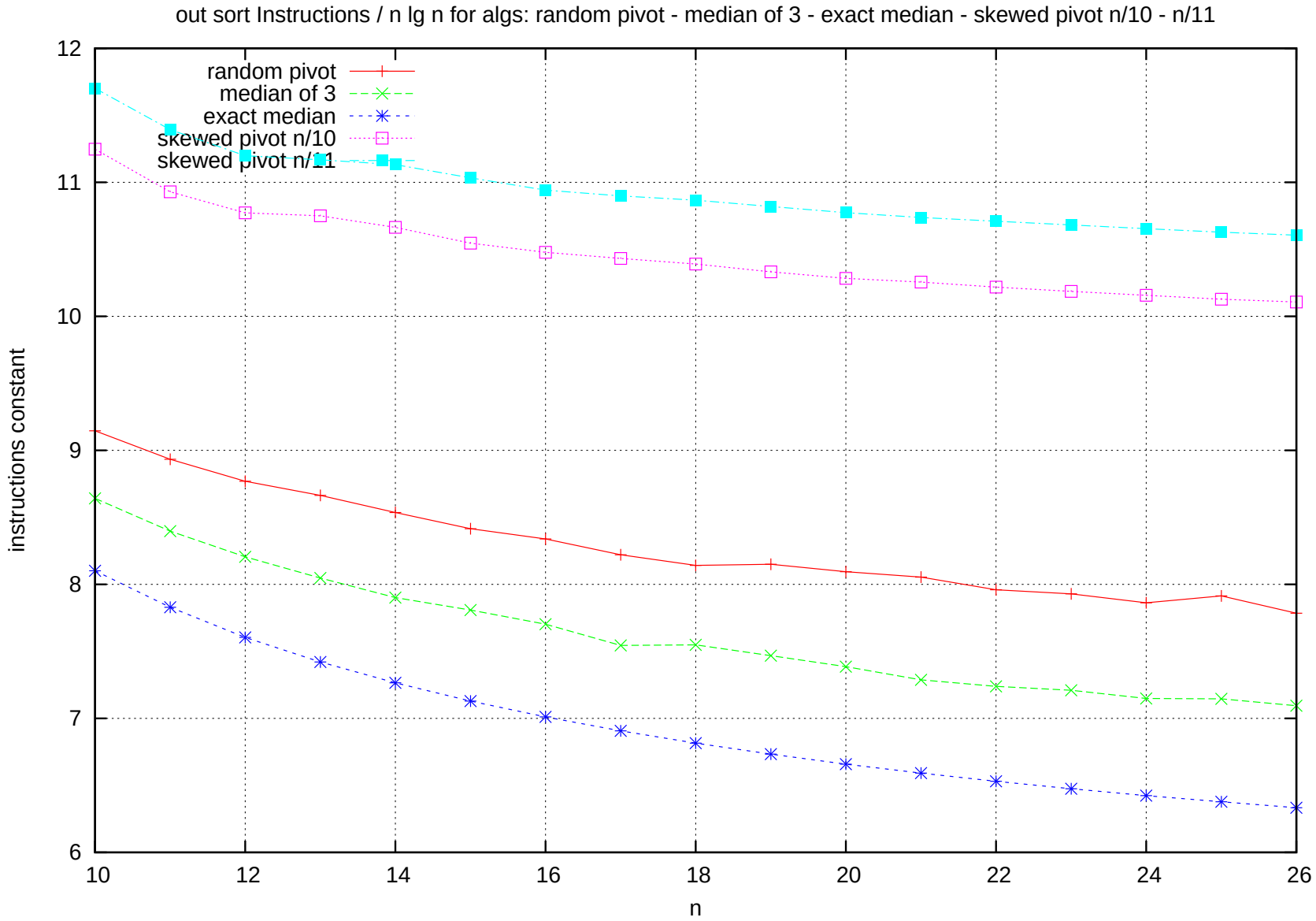
perfect: $\longrightarrow n \log n$ element comparisons

(approximate using large samples)

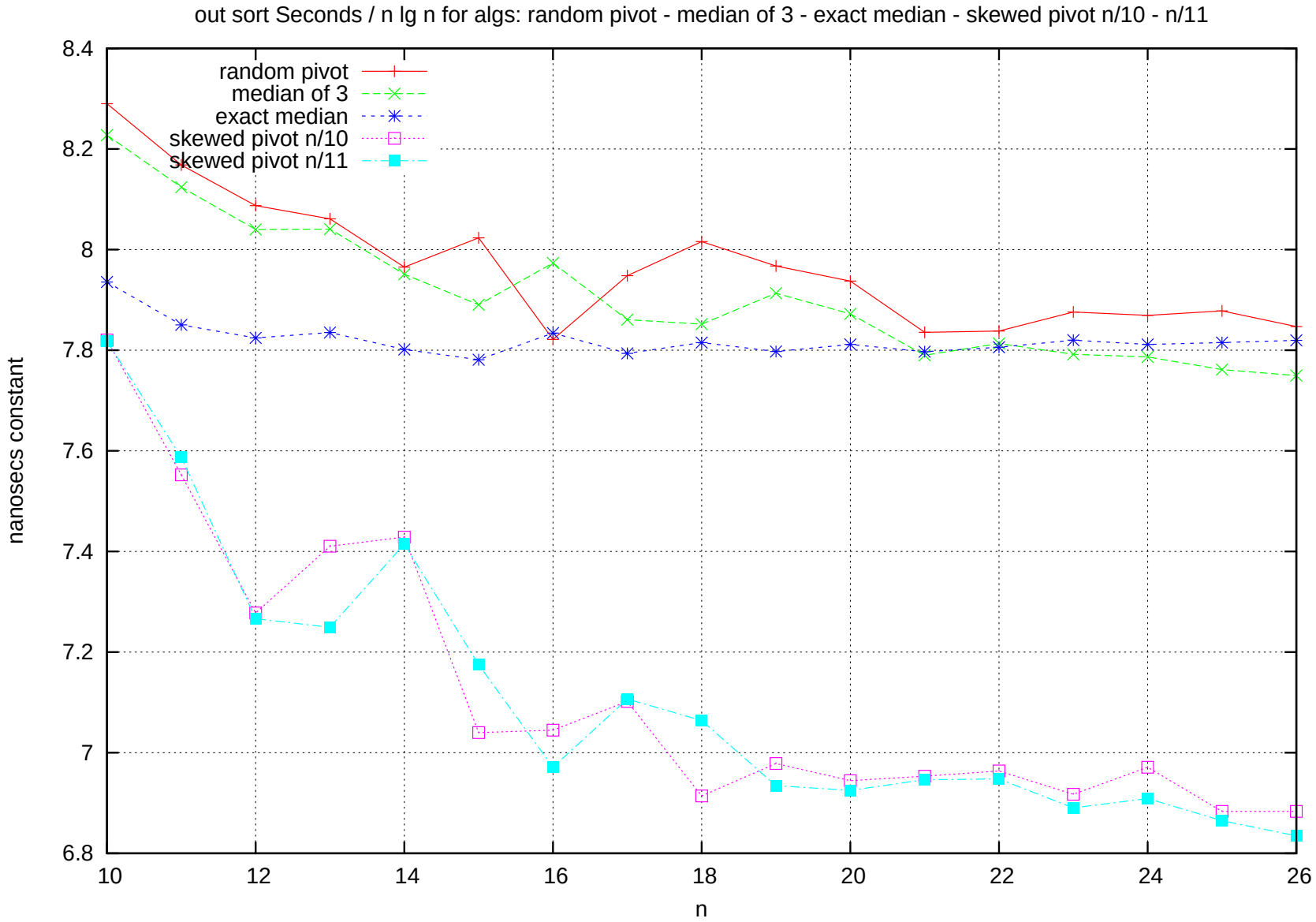
Practice

3GHz Pentium 4 Prescott, g++

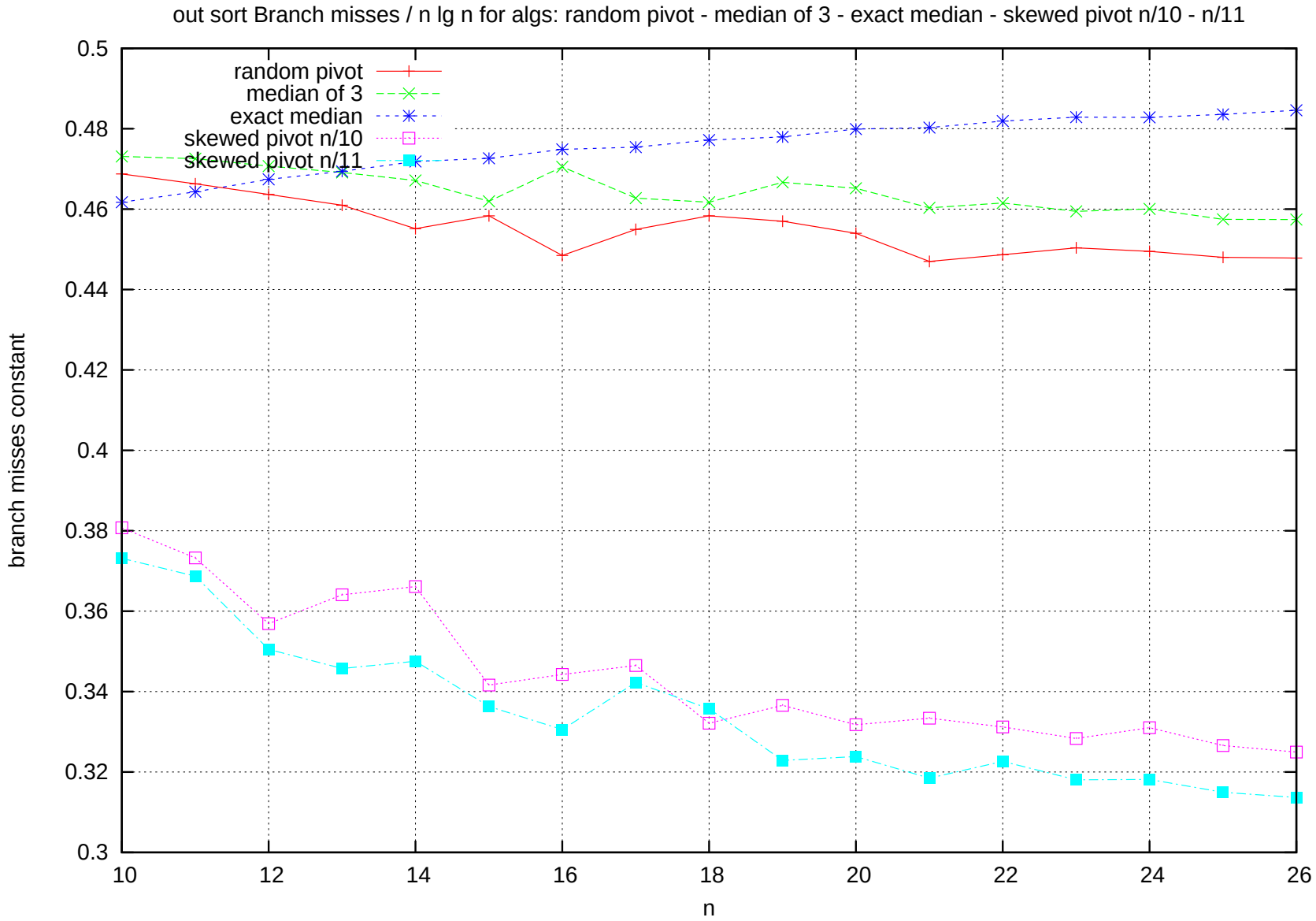
Picking Pivots Painstakingly — Instructions



Picking Pivots Painstakingly — Time



Picking Pivots Painstakingly — Branch Misses



Can We Do Better? Previous Work

Integer Keys

+ Can be $2-3$ times faster than quicksort
– Naive ones are cache inefficient and slower than quicksort
– Simple ones are distribution dependent.

Cache efficient sorting

k-ary merge sort

[Nyberg et al. 94, Arge et al. 04, Ranade et al. 00, Brodal et al. 04]

+ Faktor $\log k$ less cache faults
– Only ≈ 20 % speedup, and only for laaarge inputs

Sample Sort

Function sampleSort$(e = \langle e_1, \ldots, e_n \rangle, k)$

if n/k is "small" **then return** smallSort(e)

let $S = \langle S_1, \ldots, S_{ak-1} \rangle$ denote a random sample of e

sort S

$\langle s_0, s_1, s_2, \ldots, s_{k-1}, s_k \rangle :=$

$\langle -\infty, S_a, S_{2a}, \ldots, S_{(k-1)a}, \infty \rangle$

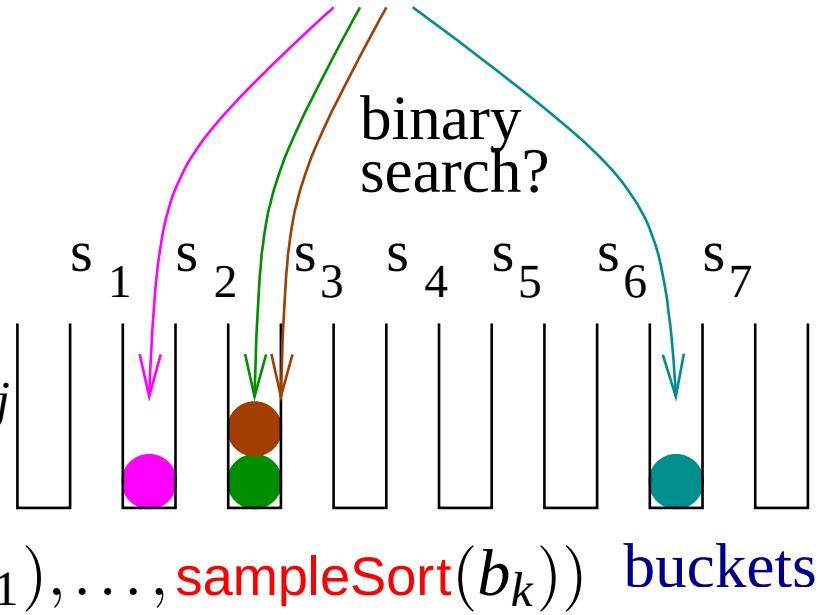
for $i := 1$ **to** n **do**

find $j \in \{1, \ldots, k\}$

such that $s_{j-1} < e_i \leq s_j$

place e_i in bucket b_j

return concatenate$(\text{sampleSort}(b_1), \ldots, \text{sampleSort}(b_k))$



Why Sample Sort?

- traditionally: parallelizable on coarse grained machines
- \+ Cache efficient $\approx$ merge sort
- – Binary search not much faster than merging
- – complicated memory management

Super Scalar Sample Sort

- Binary search $\longrightarrow$ implicit search tree
- Eliminate all conditional branches
- $\rightsquigarrow$ Exploit instruction parallelism
- $\rightsquigarrow$ Cache efficiency comes to bear
- “steal” memory management from radix sort

Classifying Elements

$t := \langle s_{k/2}, s_{k/4}, s_{3k/4}, s_{k/8}, s_{3k/8}, s_{5k/8}, s_{7k/8}, \ldots \rangle$

for $i := 1$ **to** n **do**

$j := 1$

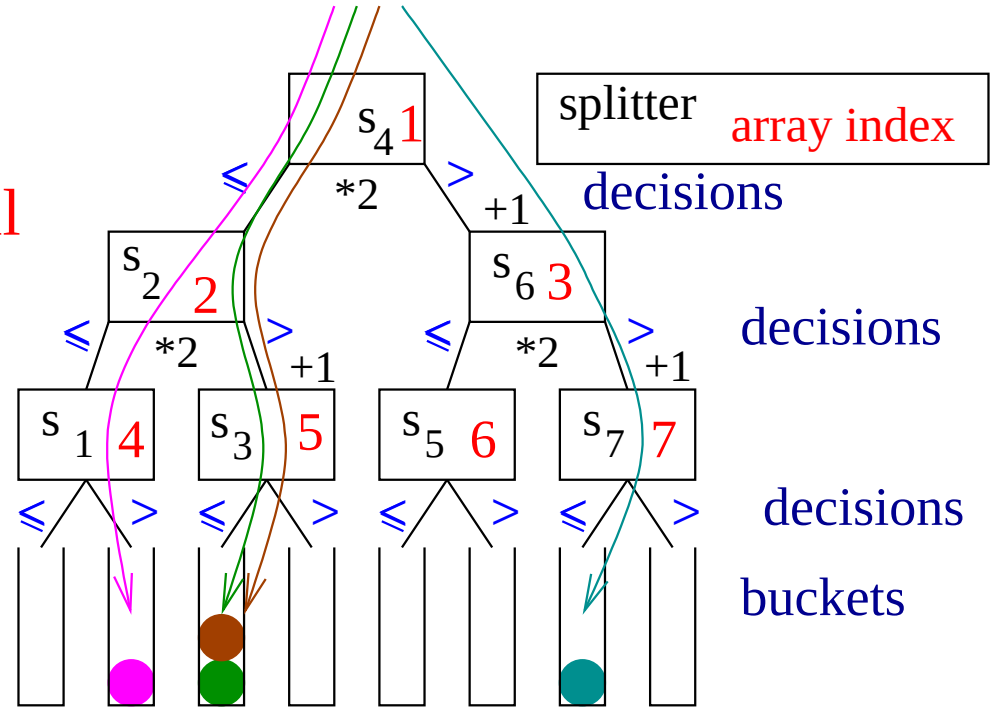
repeat $\log k$ times // unroll

$j := 2j + (a_i > t_j)$

$j := j - k + 1$

$|b_j|{+}{+}$

o(i):= j // oracle



Now the compiler should:

- use predicated instructions
- interleave for loop iterations (unrolling ∨ software pipelining)


```
template <class T>
void findOraclesAndCount(const T* const a,
    const int n, const int k, const T* const s,
    Oracle* const oracle, int* const bucket) {
{ for (int i = 0;  i < n;  i++)
    int j = 1;
    while (j < k) {
      j = j*2 + (a[i] > s[j]);
    }
    int b = j-k;
    bucket[b]++;
    oracle[i] = b;
  }
}
```

Predication

Hardware mechanism that allows instructions to be conditionally executed

- Boolean predicate registers (1–64) hold condition codes
- predicate registers p are additional inputs of predicated instructions I
- At runtime, I is executed if and only if p is true
- \+ Avoids branch misprediction penalty
- \+ More flexible instruction scheduling
- – Switched off instructions still take time
- – Longer opcodes
- – Complicated hardware design

Example (IA-64)

Translation of: if (r1 > 2) r3 := r3 + 4

With a conditional branch:

```
      cmp.gt p6,p7=r1,r2
(p7)  br.cond .label
      add r3=4,r3
.label:
                    (...)
```

Via predication:

```
      cmp.gt p6,p7=r1,r2
(p6)  add r3=4,r3
```

Other Current Architectures:

Conditional moves only

Unrolling ($k = 16$)

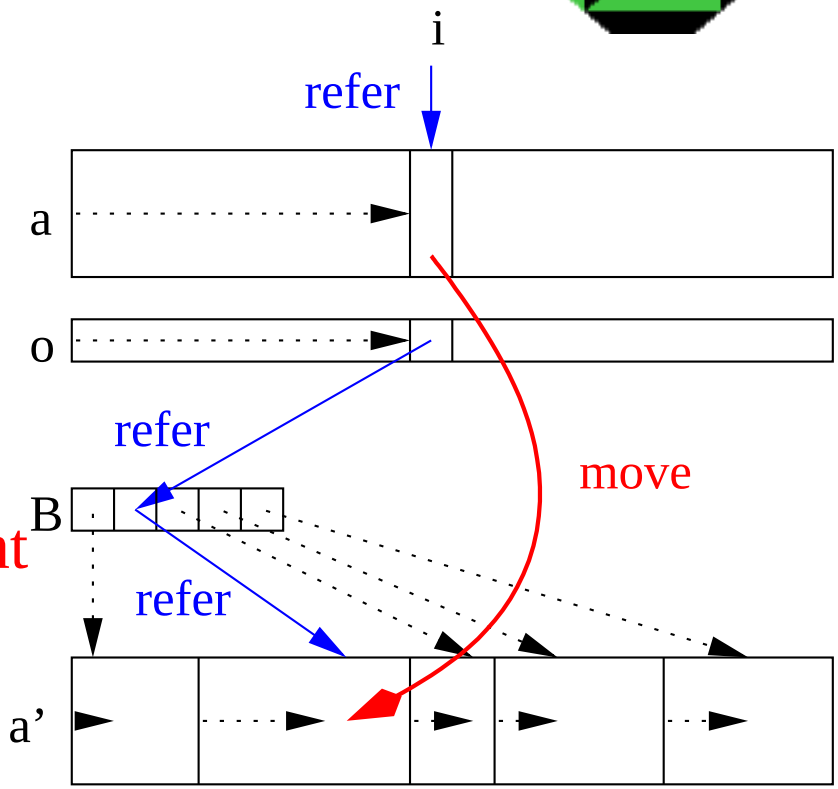
```
template <class T>
void findOraclesAndCountUnrolled([...]){
  for (int i = 0;  i < n;  i++)
    int j = 1;
    j = j*2 + (a[i] > s[j]);
    j = j*2 + (a[i] > s[j]);
    j = j*2 + (a[i] > s[j]);
    j = j*2 + (a[i] > s[j]);
    int b = j-k;
    bucket[b]++;
    oracle[i] = b;
} }
```

More Unrolling $k = 16$, n even

```
template <class T>
void findOraclesAndCountUnrolled2([...]){
   for (int i = n & 1;  i < n;  i+=2) {\
      int j0 = 1;                int j1 = 1;
      T  ai0 = a[i];             T ai1  = a[i+1];
      j0=j0*2+(ai0>s[j0]);  j1=j1*2+(ai1>s[j1]);
      j0=j0*2+(ai0>s[j0]);  j1=j1*2+(ai1>s[j1]);
      j0=j0*2+(ai0>s[j0]);  j1=j1*2+(ai1>s[j1]);
      j0=j0*2+(ai0>s[j0]);  j1=j1*2+(ai1>s[j1]);
      int b0 = j0-k;             int b1 = j1-k;
      bucket[b0]++;              bucket[b1]++;
      oracle[i] = b0;            oracle[i+1] = b1;
} }
```

Distributing Elements

for $i := 1$ **to** n **do** $a'_{B[o(i)]++} := a_i$



Why Oracles?

- simplifies memory management
- no overflow tests or re-copying
- simplifies software pipelining
- separates computation and memory access aspects
- small (n bytes)
- sequential, predictable memory access
- can be hidden using prefetching / write buffering

Distributing Elements

```
template <class T> void distribute(
  const T* const a0, T* const a1,
  const int n, const int k,
  const Oracle* const oracle, int* const bucket)
{ for (int i = 0, sum = 0;  i <= k;  i++) {
    int t = bucket[i];  bucket[i] = sum;  sum += t;
  }
  for (int i = 0; i < n;  i++) {
    a1[bucket[oracle[i]]++] = a0[i];
  }
}
```

Aufgabe

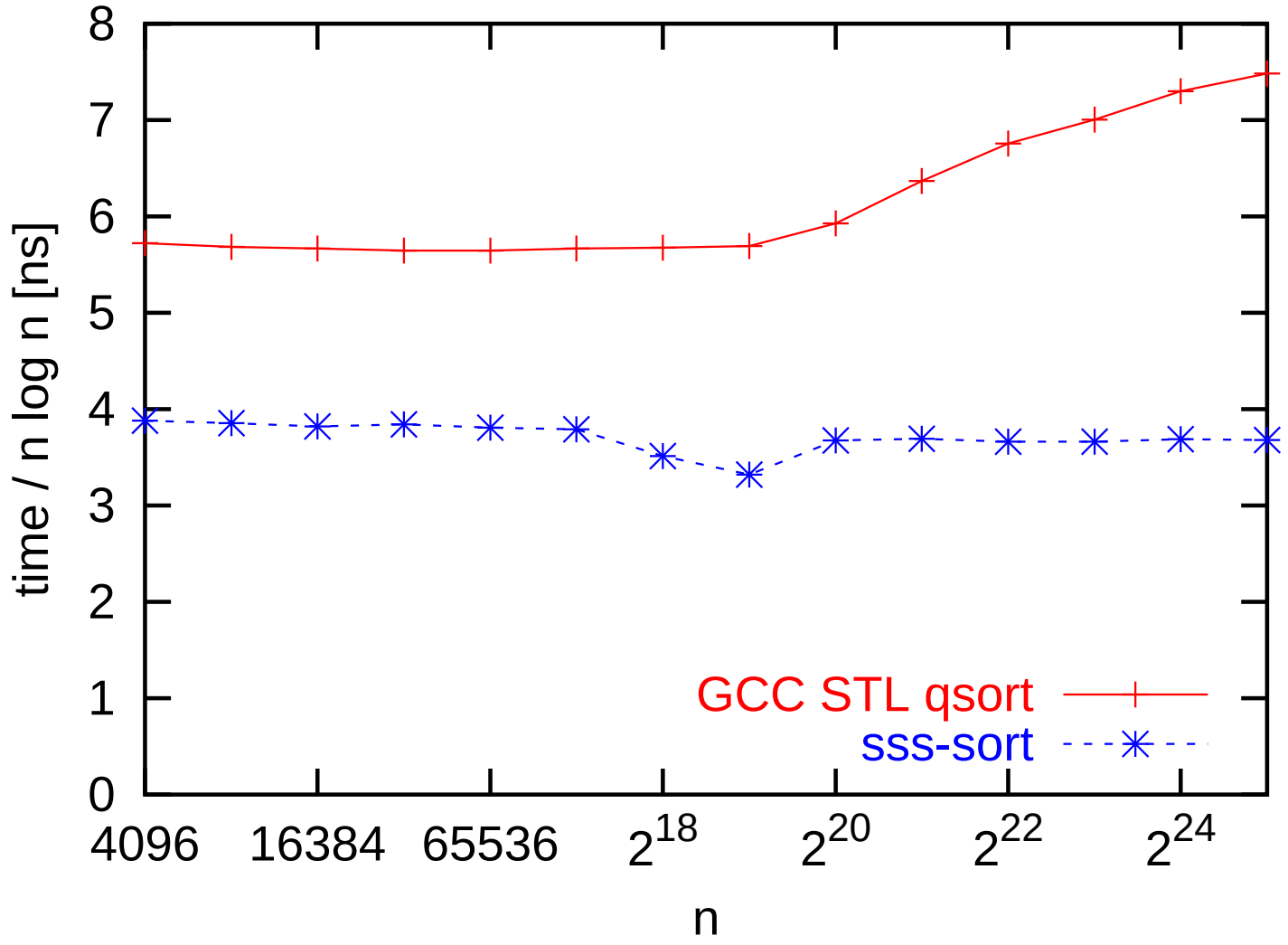
Wie implementiert man 4-Wege Mischen (8-Wege-Mischen) mit

- $2n$ Speicherzugriffen und
- $2n$ ($3n$) Vergleichen
- auf einer Maschine mit 8 (16) Registern?
- Ist eine naive Implementierung mit $3n$ ($7n$) leichter vorhersagbaren Vergleichen am Ende schneller?

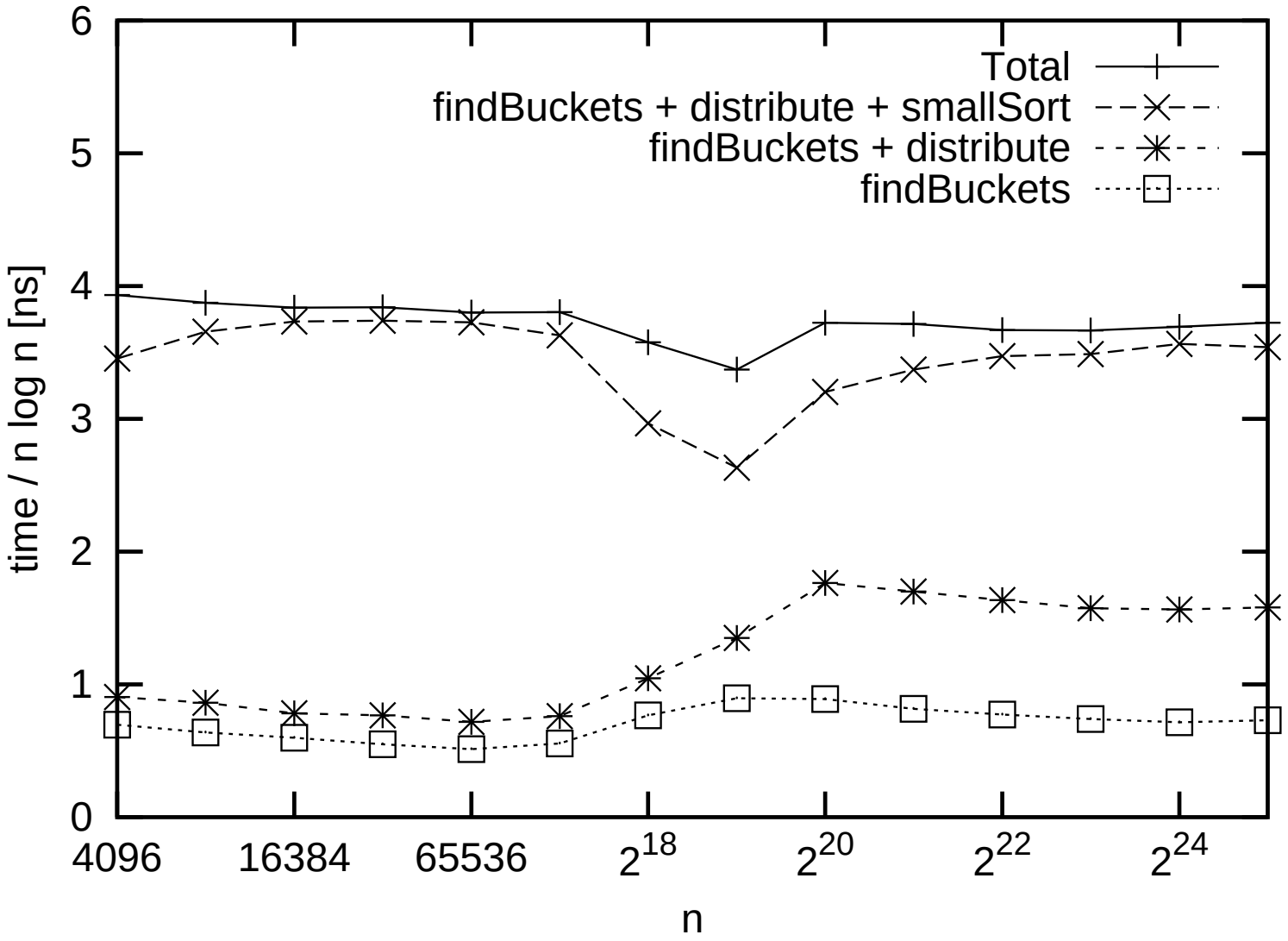
Experiments: 1.4 GHz Itanium 2

- `restrict` keyword from ANSI/ISO C99
  to indicate nonaliasing
- Intel's C++ compiler v8.0 uses predicated instructions
  automatically
- Profiling gives 9% speedup
- $k = 256$ splitters
- Use `stl:sort` from g++ ($n \leq 1000$)!
- insertion sort for $n \leq 100$
- Random 32 bit integers in $[0, 10^9]$

Comparison with Quicksort



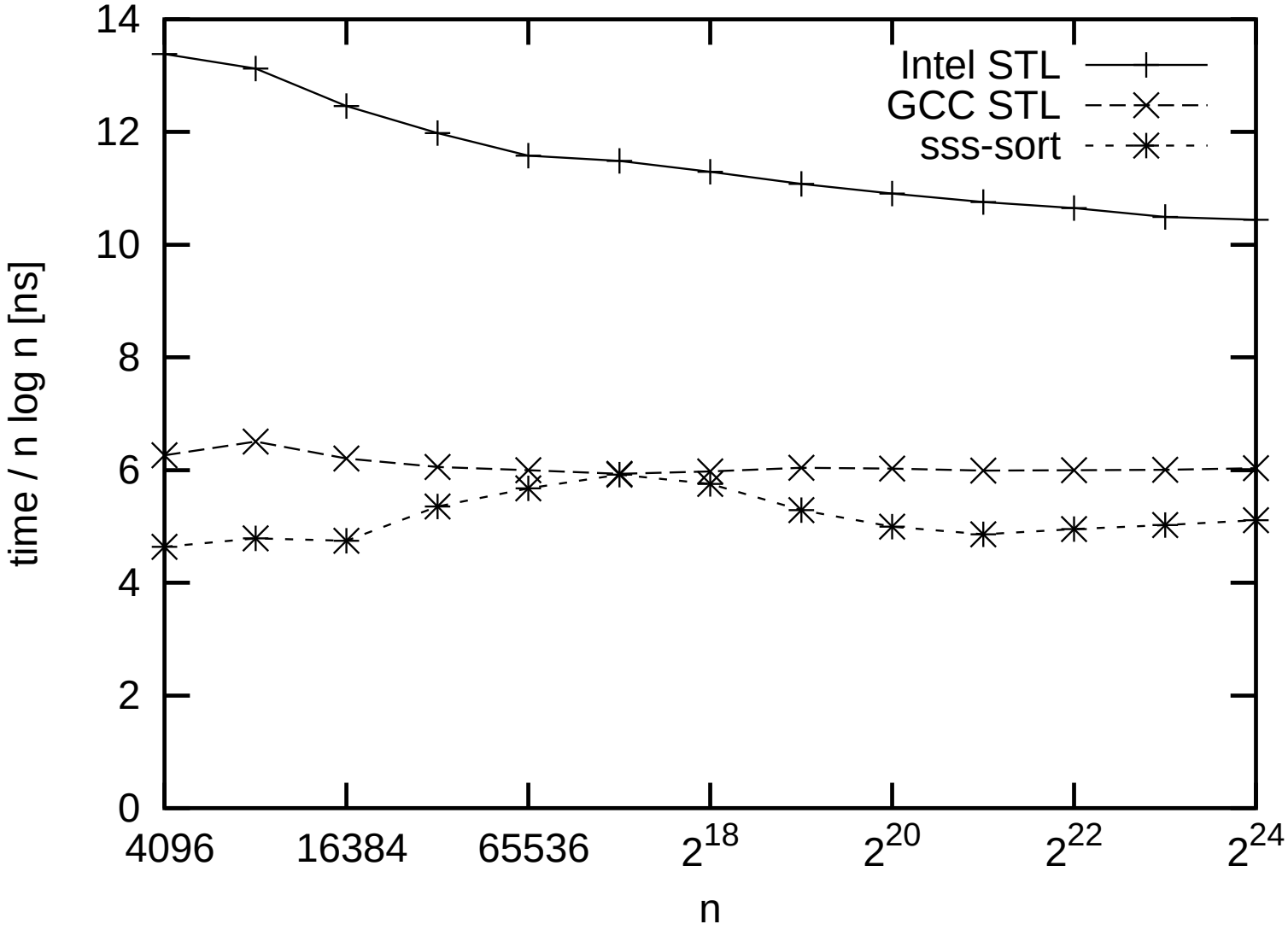
Breakdown of Execution Time



A More Detailed View

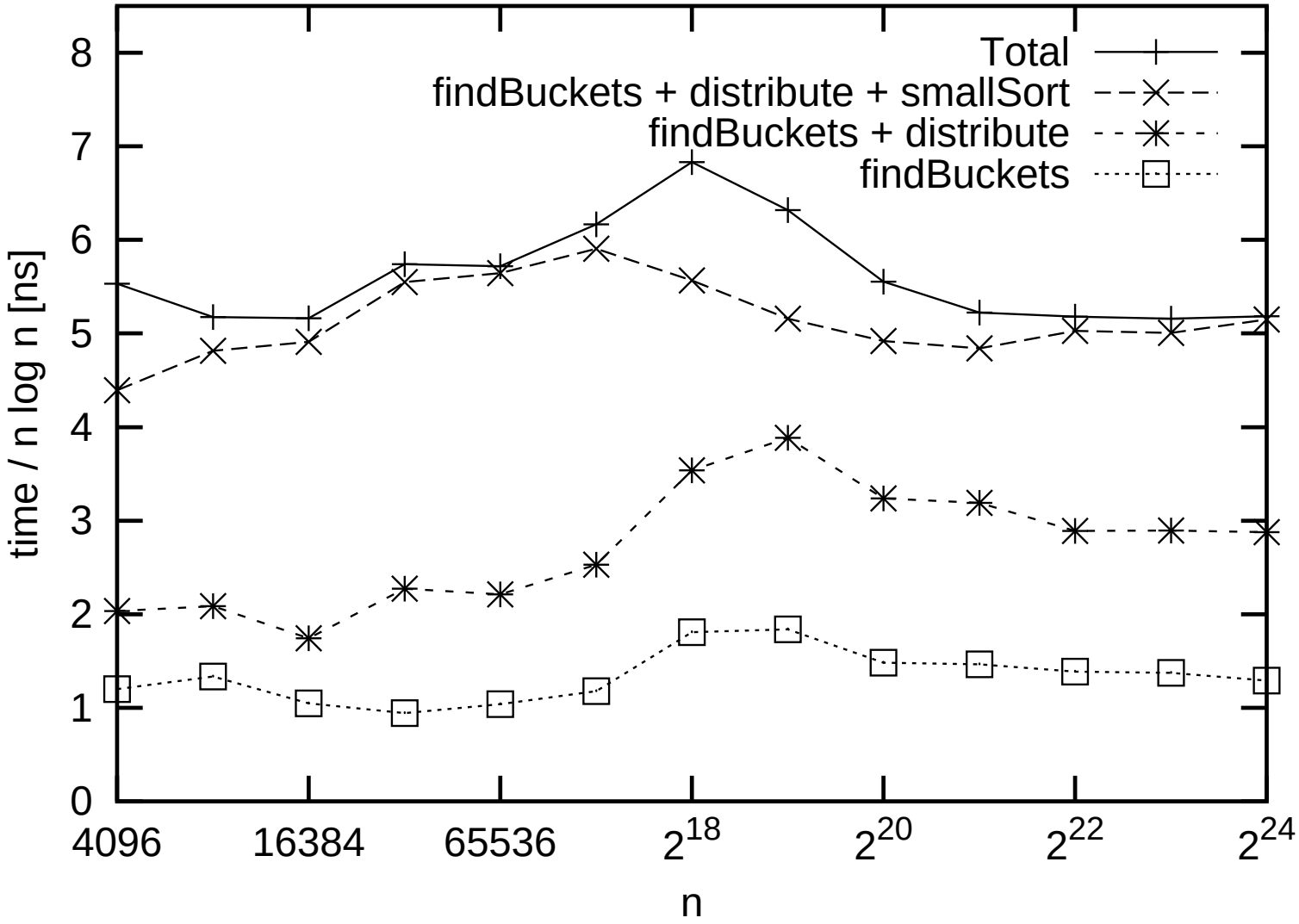
	instr.	cycles	dynamic IPC small n	dynamic IPC $n = 2^{25}$
findBuckets, 1× outer loop	63	11	5.4	4.5
distribute, one element	14	4	3.5	0.8

Comparison with Quicksort Pentium 4



Problems: few registers, one condition code only, compiler needs “help”

Breakdown of Execution Time Pentium 4



Analysis

	mem. acc.	branches	data dep.	I/Os	registers	instructions
k-way distribution:						
sss-sort	$n \log k$	$O(1)$	$O(n)$	$3.5n/B$	$3\times$unroll	$O(\log k)$
quicksort $\log k$ lvls.	$2n \log k$	$n \log k$	$O(n \log k)$	$2\frac{n}{B} \log k$	4	$O(1)$
k-way merging:						
memory	$n \log k$	$n \log k$	$O(n \log k)$	$2n/B$	7	$O(\log k)$
register	$2n$	$n \log k$	$O(n \log k)$	$2n/B$	k	$O(k)$
funnel $k'^{\log_{k'} k}$	$2n \log_{k'} k$	$n \log k$	$O(n \log k)$	$2n/B$	$2k'+2$	$O(k')$

Conclusions

- sss-sort up to twice as fast as quicksort on Itanium
- comparisons $\neq$ conditional branches
- algorithm analysis is not just instructions and caches

Kritik I

Warum nur zufällige Schlüssel?

Antwort I

Sample sort ist kaum von der Verteilung der Eingabe abhängig

Kritik I´

Aber was wenn es viele gleiche Schlüssel gibt?

Die landen alle in einem Bucket

Antwort I´

Its not a bug its a feature:

Sei $s_i = s_{i+1} = \cdots = s_j$ Indikator für häufigen Schlüssel!

Setze $s_i := \max\{x \in Key : x < s_i\}$,

(optional: streiche $s_{i+2}, \ldots s_j$)

Nun muss bucket $i+1$ nicht mehr sortiert werden!

todo: implementieren

Kritik II

Quicksort ist inplace

Antwort II

Benutze hybride Listen-Array Repräsentation von Folgen

braucht $O\left(\sqrt{kn}\right)$ extra Platz für k-way sample sort

Kritik II´

Ich will aber Arrays für Ein- und Ausgabe

Antwort II´

Inplace Konvertierung

Eingabe: einfach

Ausgabe: tricky. Aufgabe: grobe Idee entwickeln. Hinweis: Permutation von Blöcken. Jede Permutation besteht aus einem Produkt zyklischer Permutationen. Inplace zyklische Permutation ist einfach

Future Work

- high level fine-tuning, e.g., clever choice of k
- other architectures, e.g., Opteron, PowerPC
- almost in-place implementation
- multilevel cache-aware or cache-oblivious generalization (oracles help)

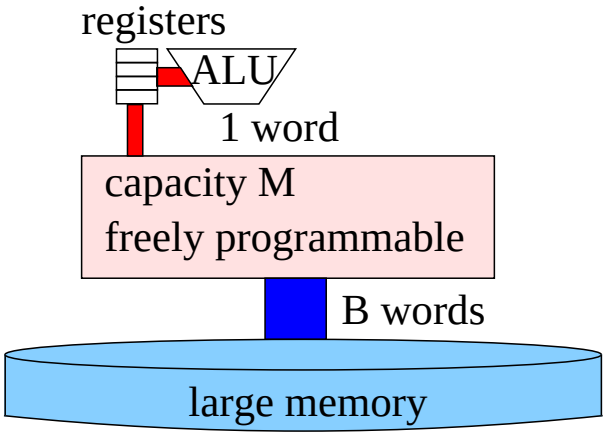
Studienarbeit, (Diplomarbeit), Hiwi-Job?

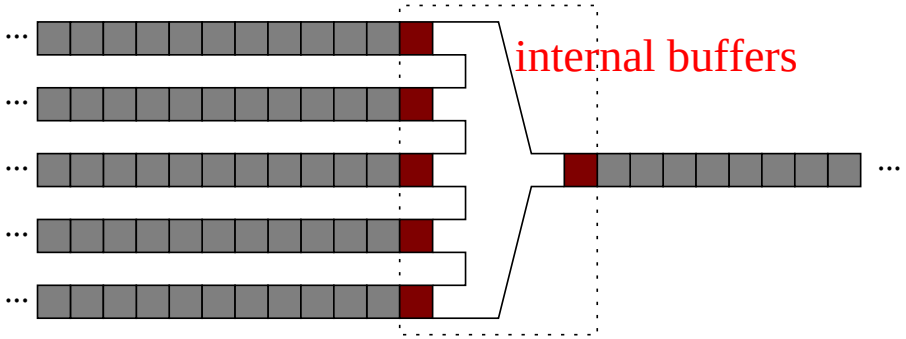
Sorting by Multiway Merging

- Sort $\lceil n/M \rceil$ runs with M elements each — $2n/B$ I/Os
- Merge M/B runs at a time — $2n/B$ I/Os
- until only one run is left — $\times \left\lceil \log_{M/B} \frac{n}{M} \right\rceil$ merge phases

In total

$$\text{sort}(n) := \frac{2n}{B}\left(1 + \left\lceil \log_{M/B} \frac{n}{M} \right\rceil\right) \text{ I/Os}$$

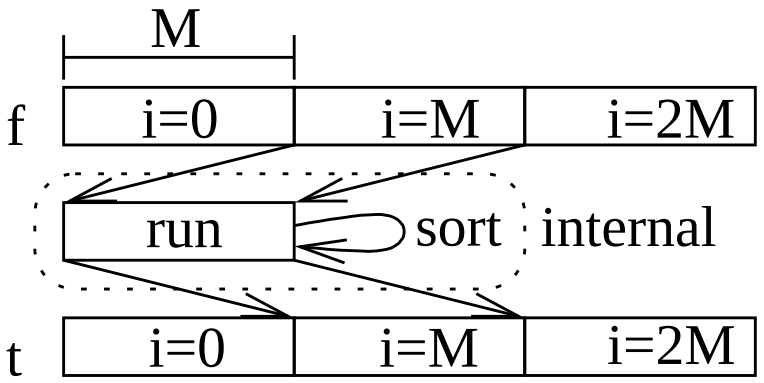




Procedure twoPassSort(M : $\mathbb{N}$; a : external **Array** $[0..n-1]$ **of** Element)
    b : external **Array** $[0..n-1]$ **of** Element // auxiliary storage
    formRuns(M, a, b)
    mergeRuns(M, b, a)

// Sort runs of size M from f writing sorted runs to t

Procedure formRuns(M : $\mathbb{N}$; f, t : external **Array** $[0..n-1]$ **of** Element)
    for $i := 0$ **to** $n-1$ **step** M **do**
        run := $f[i..i+M-1]$
        sort(run)
        $t[i..i+M-1]$:= run



// Merge n elements from f to t where f stores sorted runs of size L

Procedure mergeRuns($L : \mathbb{N}$; f, t : external **Array** $[0..n-1]$ **of** Element)

$k := \lceil n/L \rceil$ // Number of runs

next : PriorityQueue **of** Key $\times \mathbb{N}$

runBuffer := **Array** $[0..k-1][0..B-1]$ **of** Element

for $i := 0$ **to** $k-1$ **do**

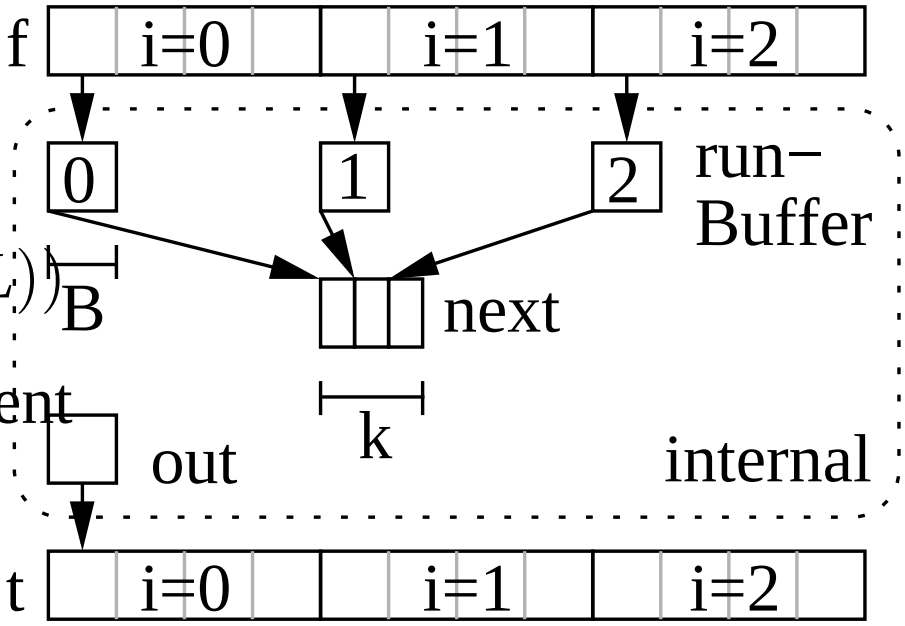
runBuffer$[i]$:=

$f[iM..iM+B-1]$

next.insert(

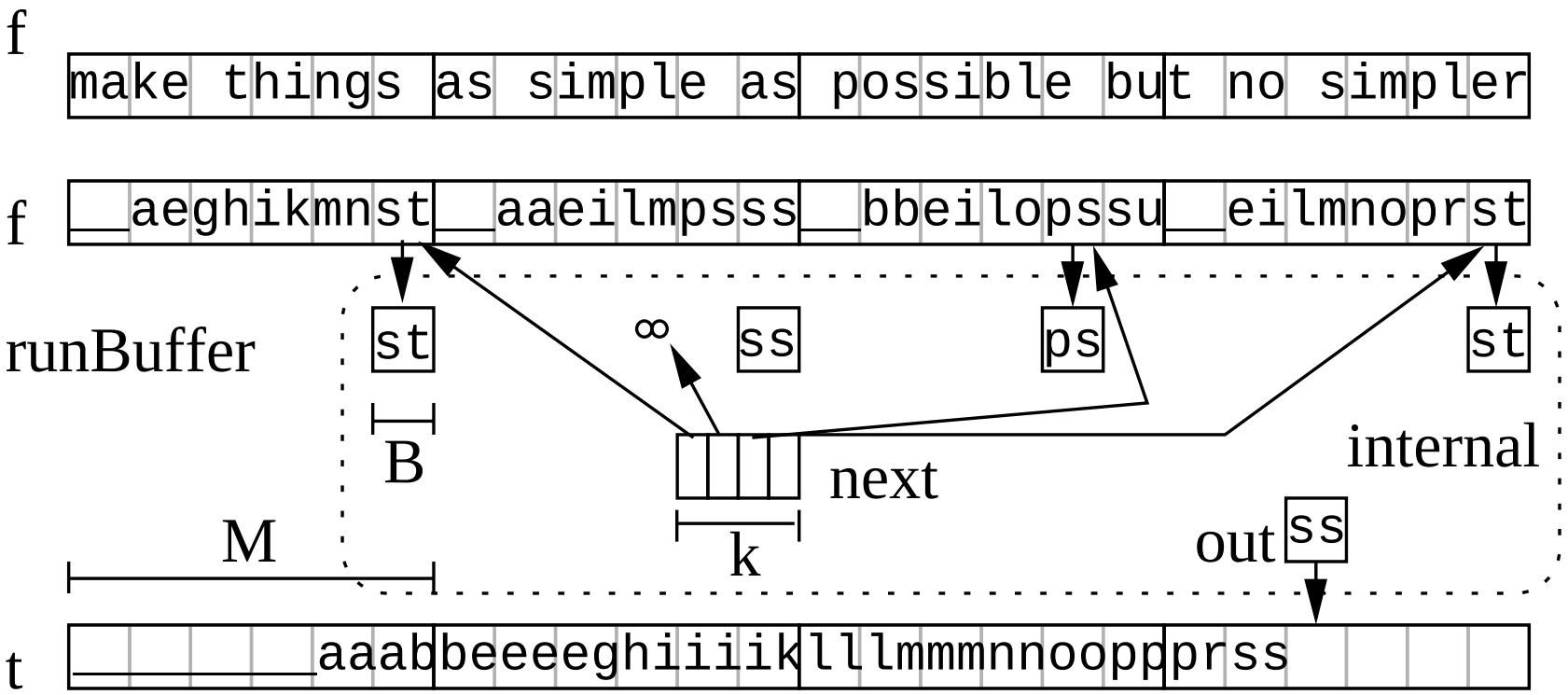
key(runBuffer$[i][0]$), iL)

out : **Array** $[0..B-1]$ **of** Element




```
//k-way merging
    for i := 0 to n − 1 step B do
        for j := 0 to B − 1 do
            (x, ℓ) := next.deleteMin
            out[j] := runBuffer[ℓ div L][ℓ mod B]
            ℓ++
            if ℓ mod B = 0 then                    // New input block
                if ℓ mod L = 0 ∨ ℓ = n then
                    runBuffer[ℓ div L][0] := ∞// sentinel for exhausted ru
                else runBuffer[ℓ div L] := f[ℓ..ℓ + B − 1]// refill buffer
            next.insert((runBuffer[ℓ div L][0], ℓ))
        write out to t[i..i + B − 1]                // One output step
```

Example, $B = 2$, run size $= 6$



2 Super Scalar Sample Sort

Comparison Based Sorting

Large data sets (thousands to millions)

Theory: Lots of algorithms with $n \log n + O(n)$ comparisons

Practice: Quicksort

- \+ $n \log n + O(n)$ expected comparisons
  using sample based pivot seletction
- \+ sufficiently cache efficient
  $O\left(\frac{n}{B} \log n\right)$ cache misses on all levels
- \+ Simple
- \+ Compiler writers are aware of it

Can we nevertheless beat quicksort?

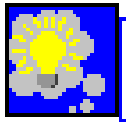
3 Sorting with Parallel Disks

I/O Step := Access to a single physical block per disk

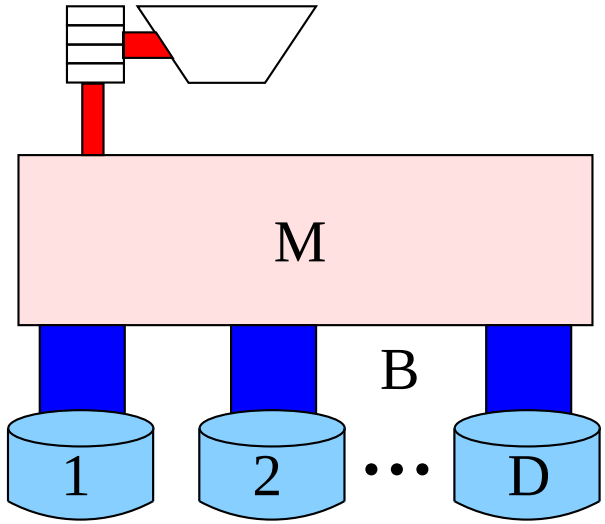
Theory: Balance Sort [Nodine Vitter 93].

Deterministic, complex

asymptotically optimal



$O(\cdot)$



independent disks

Multiway merging

"Usually" factor 10? less I/Os.

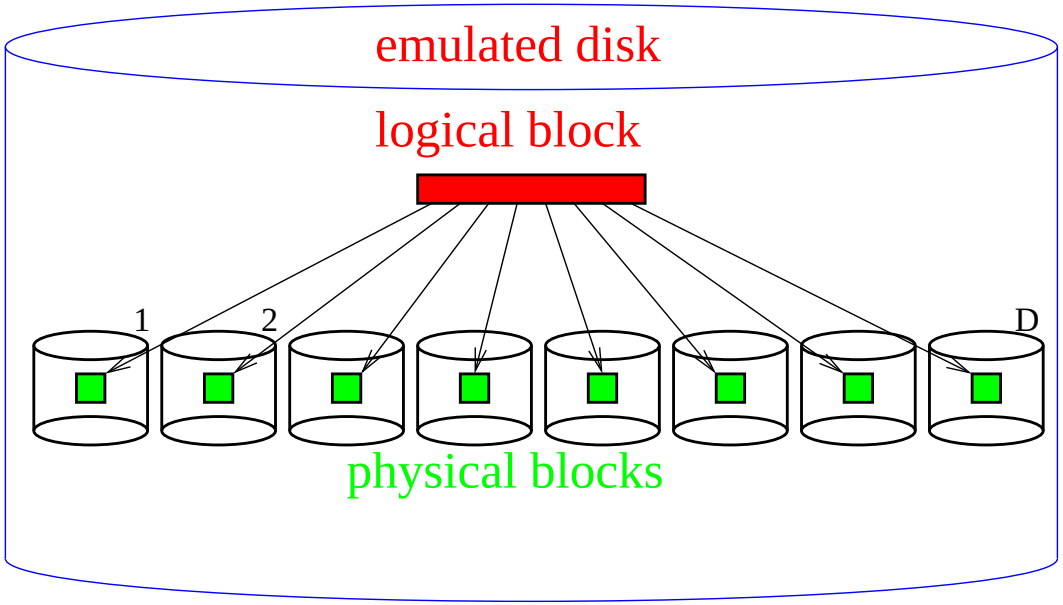
Not asymptotically optimal.

42%

[Vitter Shriver 94]

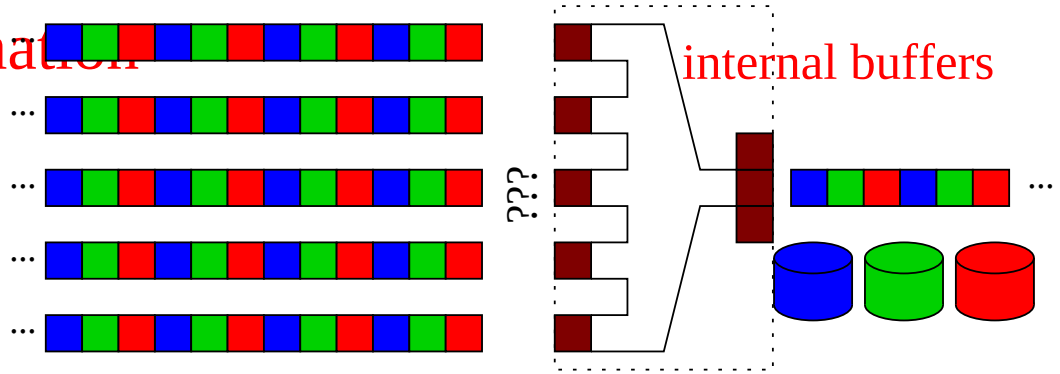
Basic Approach: Improve Multiway Merging

Striping



That takes care of run formation

and writing the output

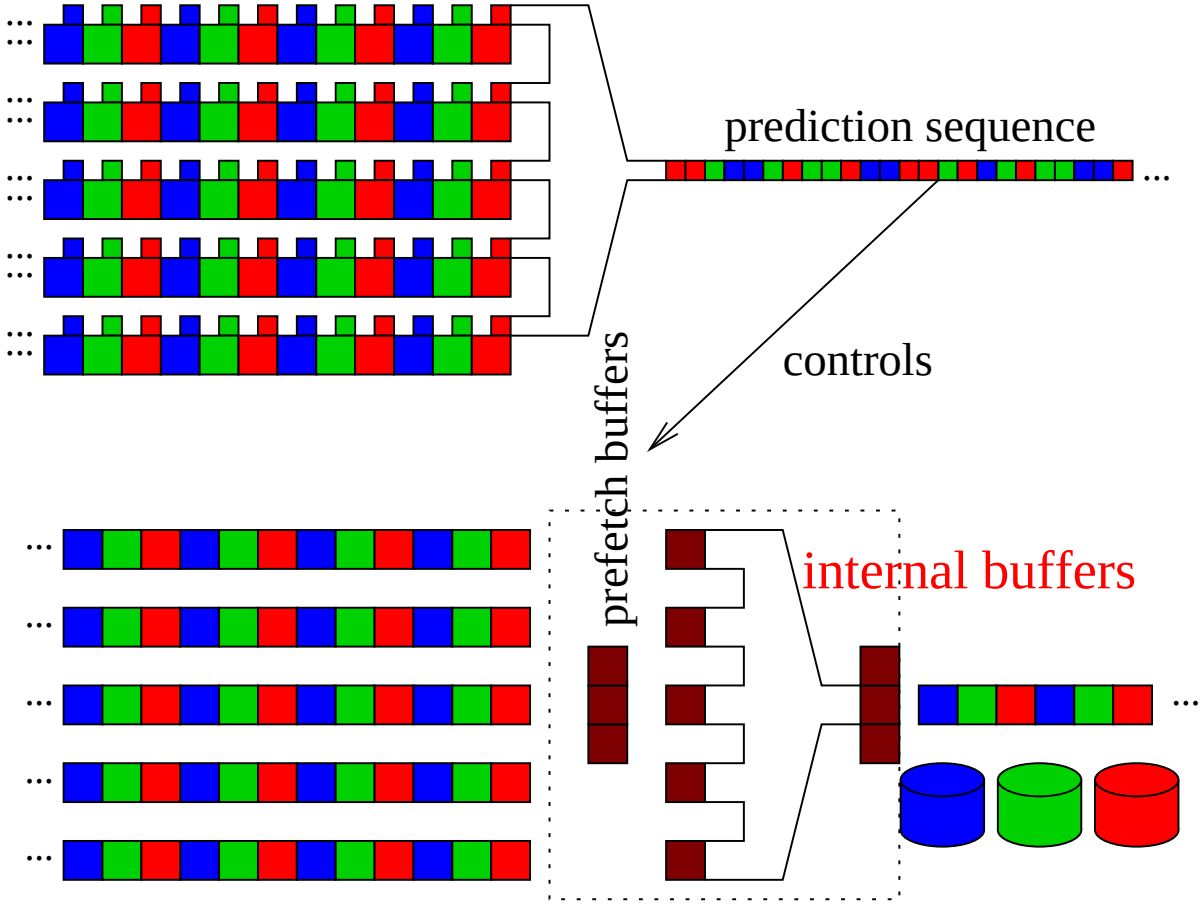


But what about merging?

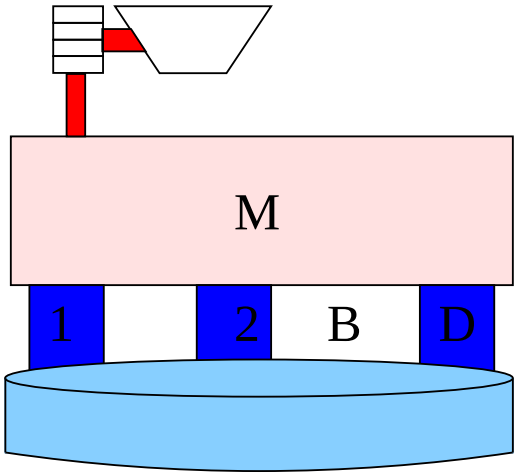
Prediction

[Folklore, Knuth]
Smallest Element
of each block
triggers fetch.

Prefetch buffers
allow parallel access
of next blocks

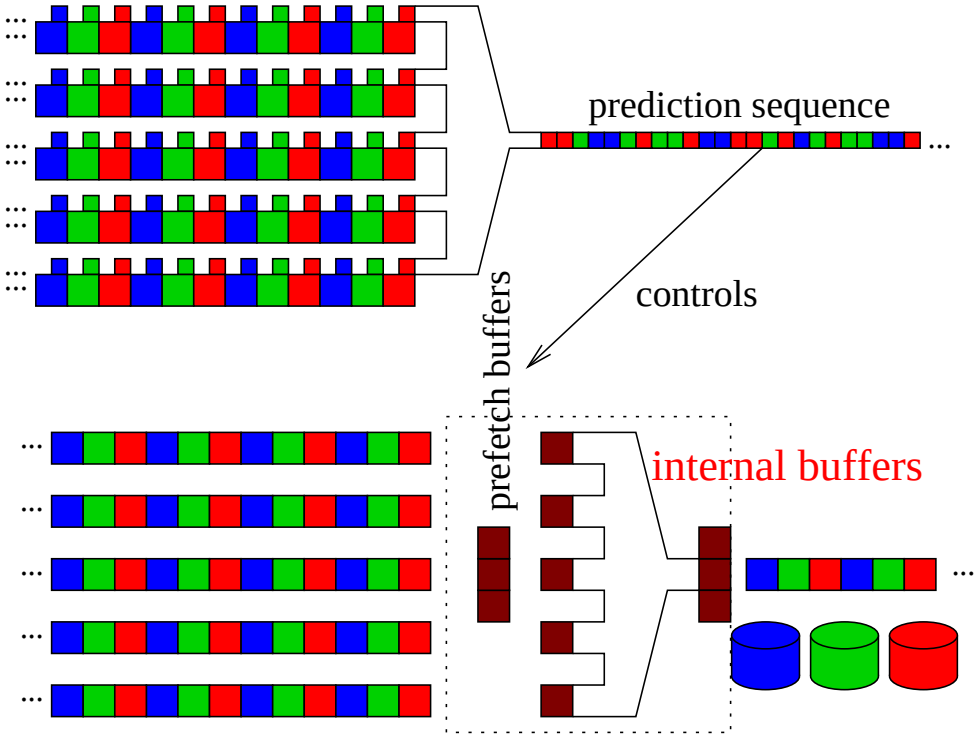


Warmup: Multihead Model



Multihead Model

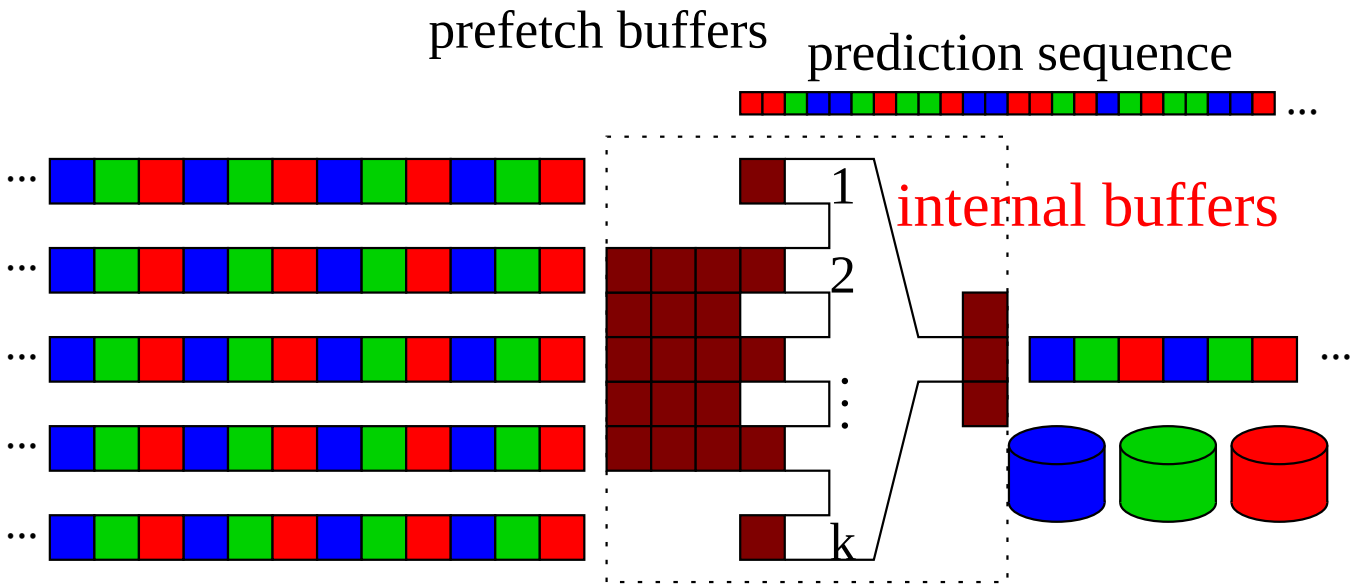
[Aggarwal Vitter 88]



D prefetch buffers yield an optimal algorithm

$$\text{sort}(n) := \frac{2n}{DB}\left(1+\left\lceil \log_{M/B}\frac{n}{M}\right\rceil\right) \text{ I/Os}$$

Bigger Prefetch Buffer



$Dk \rightsquigarrow$ good deterministic performance

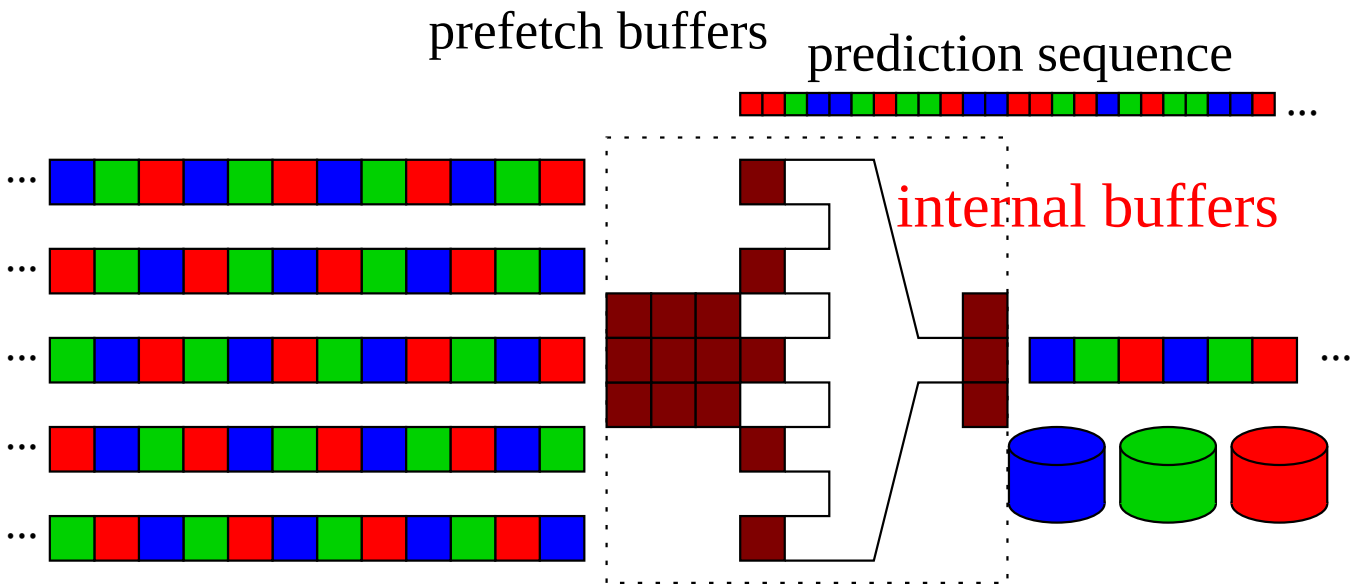
$O(D)$ would yield an optimal algoritihm.

Possible?

Randomized Cycling

[Vitter Hutchinson 01]

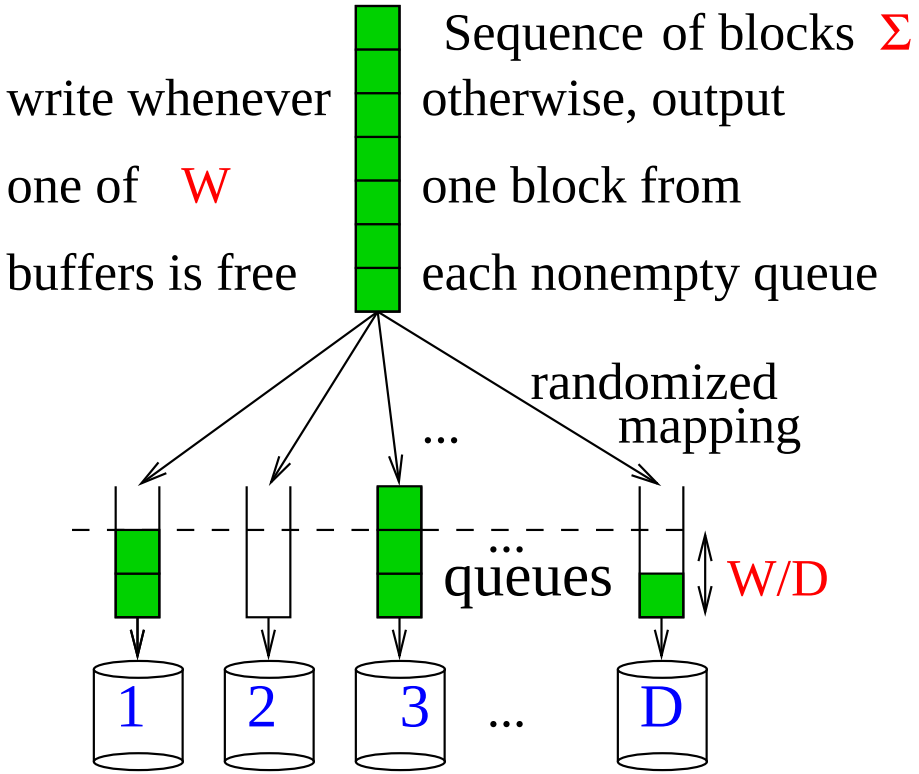
Block i of stripe j goes to disk $\pi_j(i)$ for a random permutation π_j



Good for naive prefetching and $\Omega(D \log D)$ buffers

Buffered Writing

[Sanders-Egner-Korst SODA00, Hutchinson Sanders Vitter ESA 01]



Theorem:
Buffered Writing
is optimal
...
But
how good is optimal?

Theorem: Randomized cycling achieves efficiency $1 - O(D/W)$.

Analysis: negative association of random variables, application of queueing theory to a “throttled” Alg.

Optimal Offline Prefetching

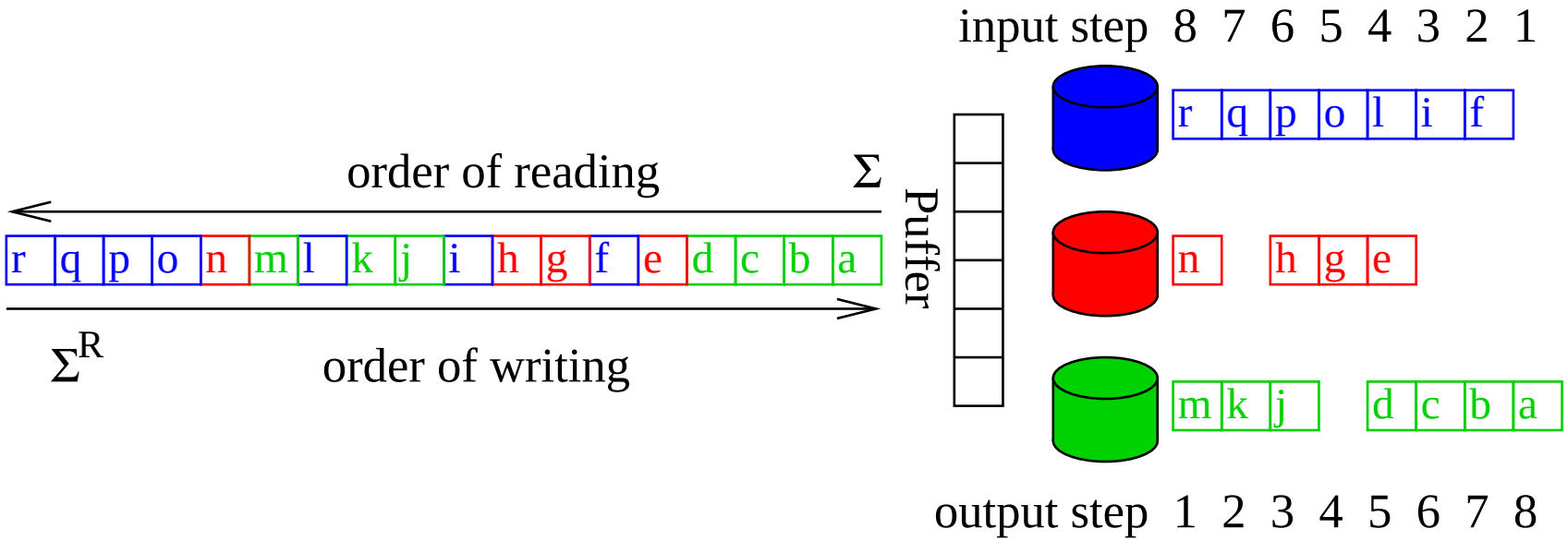
Theorem:

For buffer size W:

$\exists$ (offline) prefetching schedule for Σ with T input steps

$\Leftrightarrow$

$\exists$ (online) write schedule for Σ^R with T output steps



Optimal Offline Prefetching

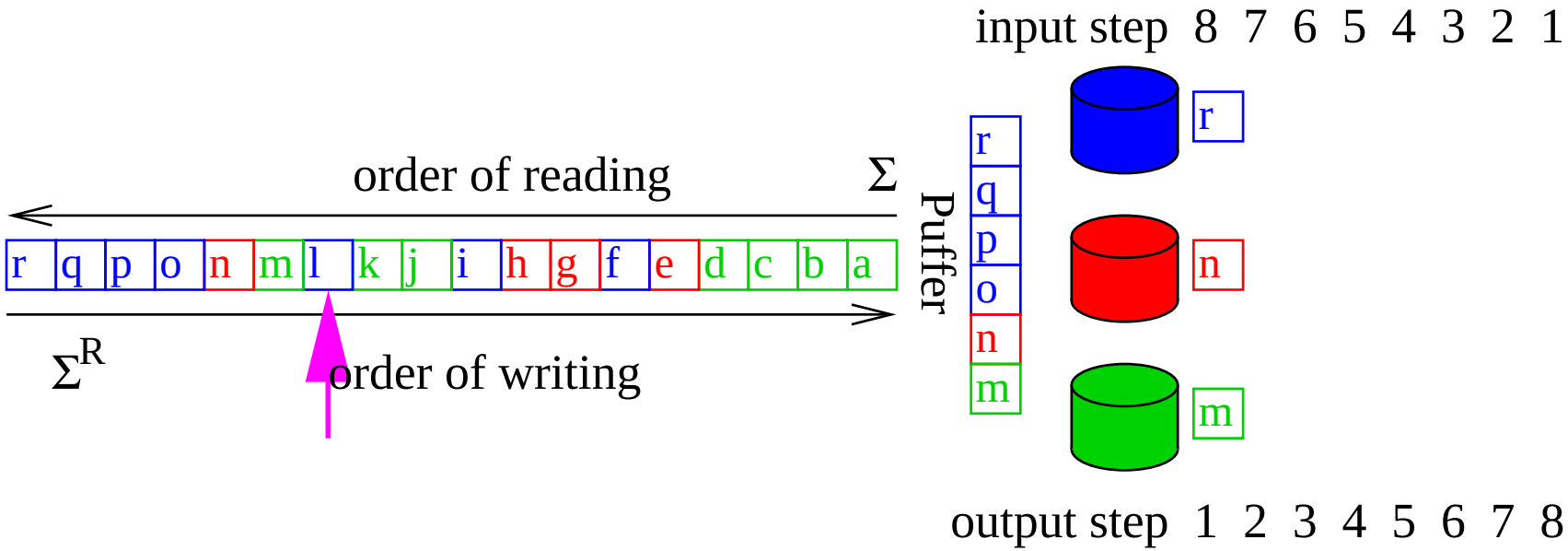
Theorem:

For buffer size W:

$\exists$ (offline) prefetching schedule for Σ with T input steps

$\Leftrightarrow$

$\exists$ (online) write schedule for Σ^R with T output steps



Optimal Offline Prefetching

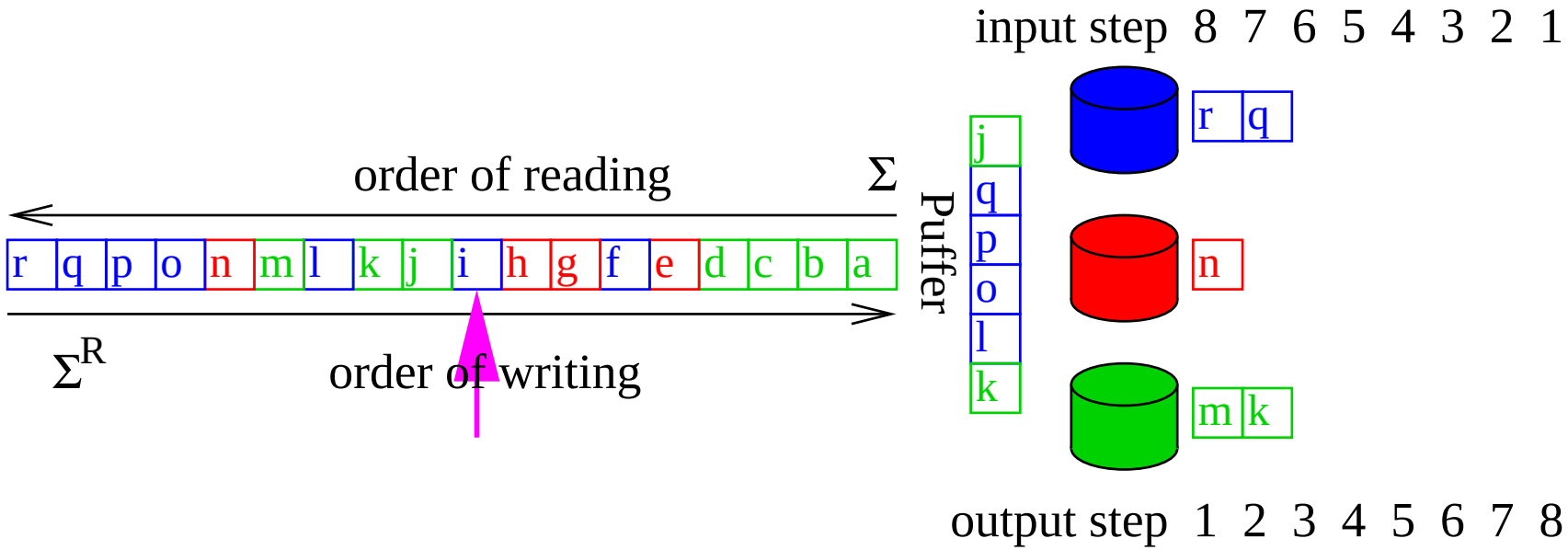
Theorem:

For buffer size W:

$\exists$ (offline) prefetching schedule for Σ with T input steps

$\Leftrightarrow$

$\exists$ (online) write schedule for Σ^R with T output steps



Optimal Offline Prefetching

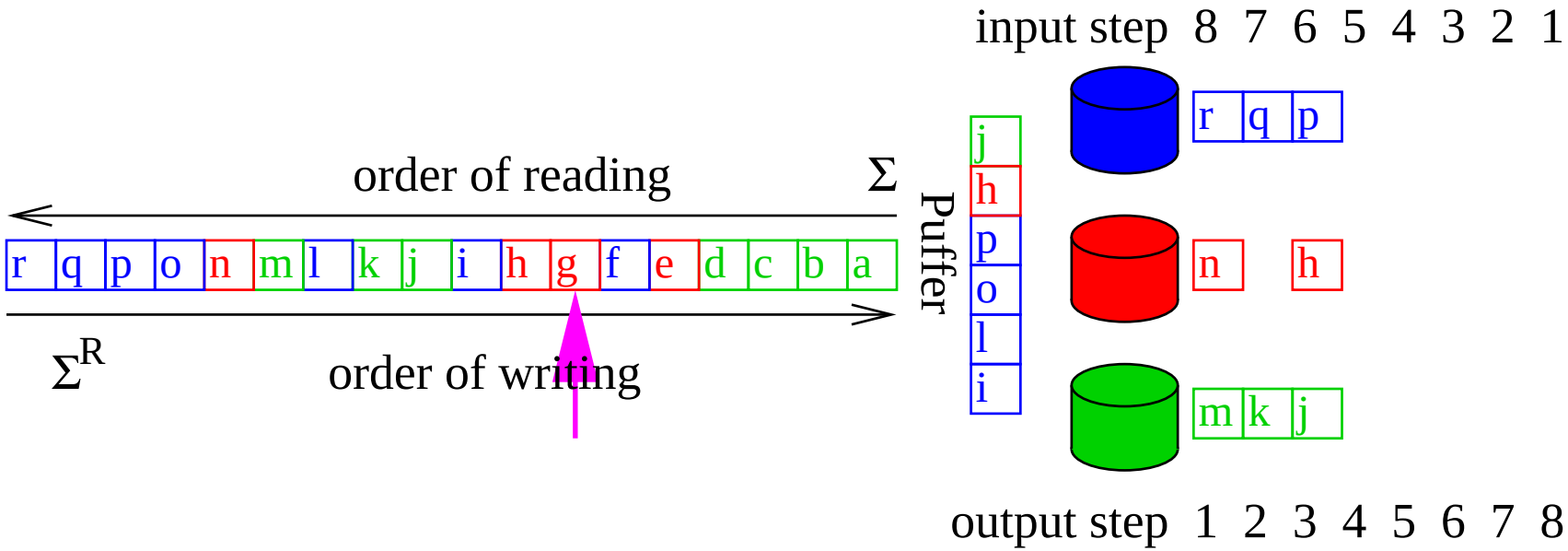
Theorem:

For buffer size W:

$\exists$ (offline) prefetching schedule for Σ with T input steps

$\Leftrightarrow$

$\exists$ (online) write schedule for Σ^R with T output steps



Optimal Offline Prefetching

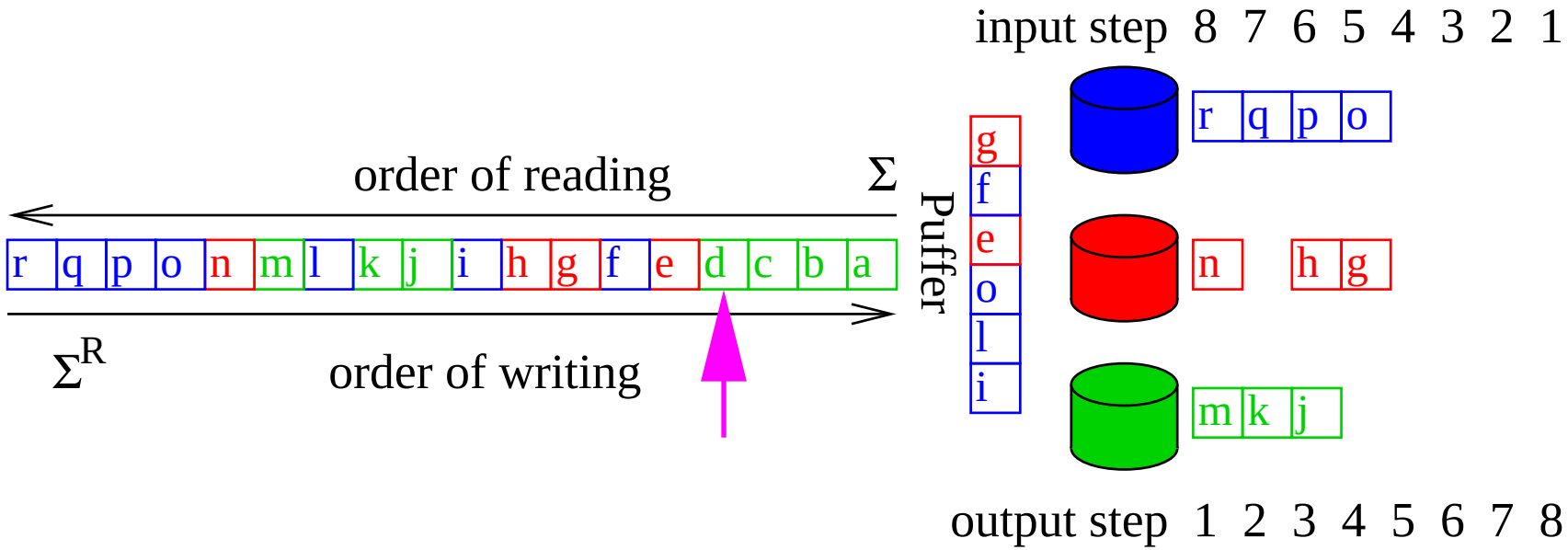
Theorem:

For buffer size W:

$\exists$ (offline) prefetching schedule for Σ with T input steps

$\Leftrightarrow$

$\exists$ (online) write schedule for Σ^R with T output steps



Optimal Offline Prefetching

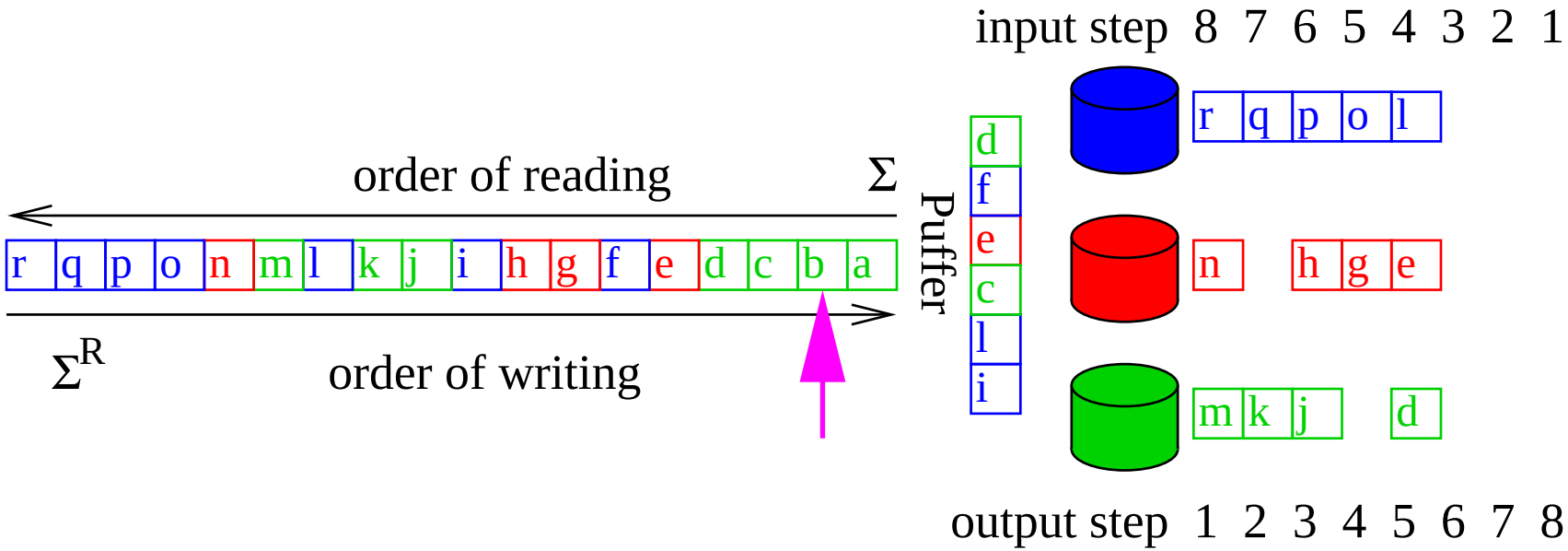
Theorem:

For buffer size W:

$\exists$ (offline) prefetching schedule for Σ with T input steps

$\Leftrightarrow$

$\exists$ (online) write schedule for Σ^R with T output steps



Optimal Offline Prefetching

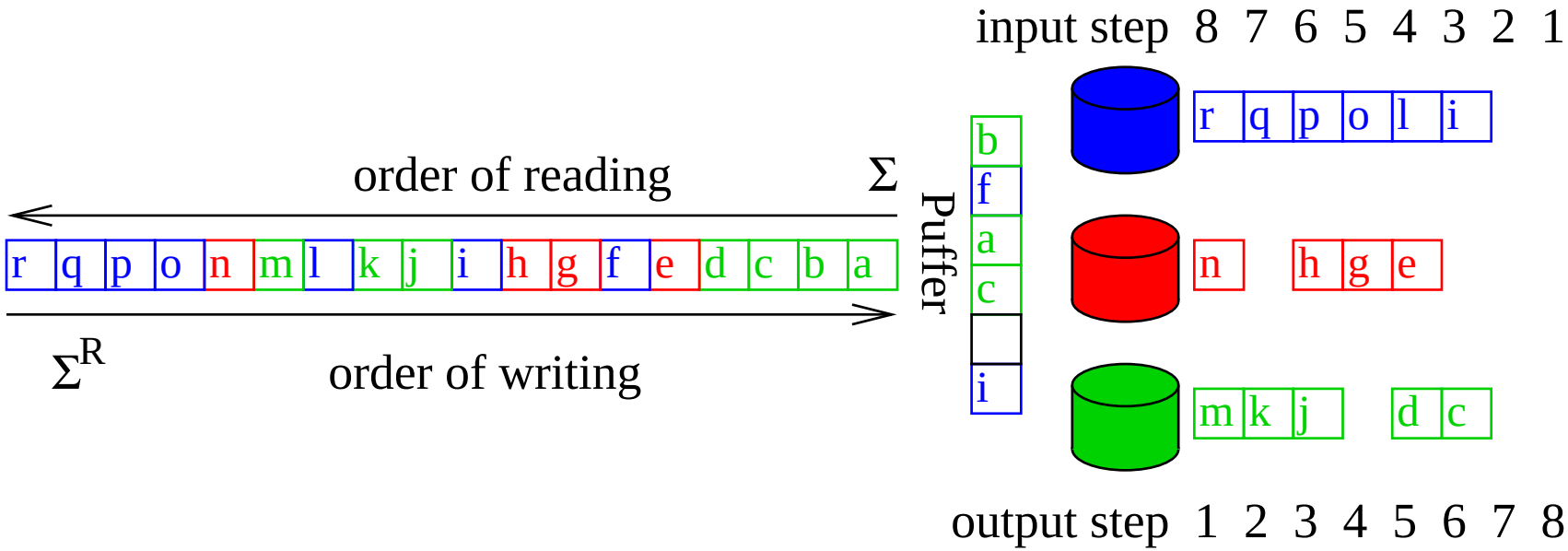
Theorem:

For buffer size W:

$\exists$ (offline) prefetching schedule for Σ with T input steps

$\Leftrightarrow$

$\exists$ (online) write schedule for Σ^R with T output steps



Optimal Offline Prefetching

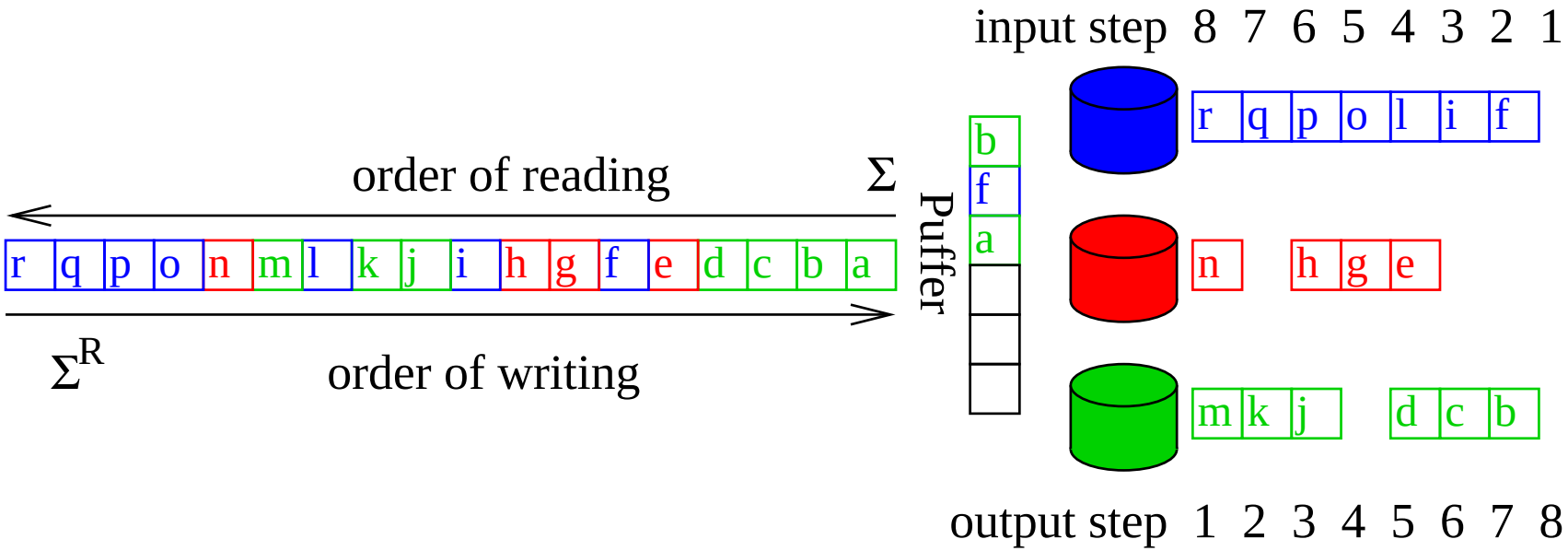
Theorem:

For buffer size W:

$\exists$ (offline) prefetching schedule for Σ with T input steps

$\Leftrightarrow$

$\exists$ (online) write schedule for Σ^R with T output steps



Optimal Offline Prefetching

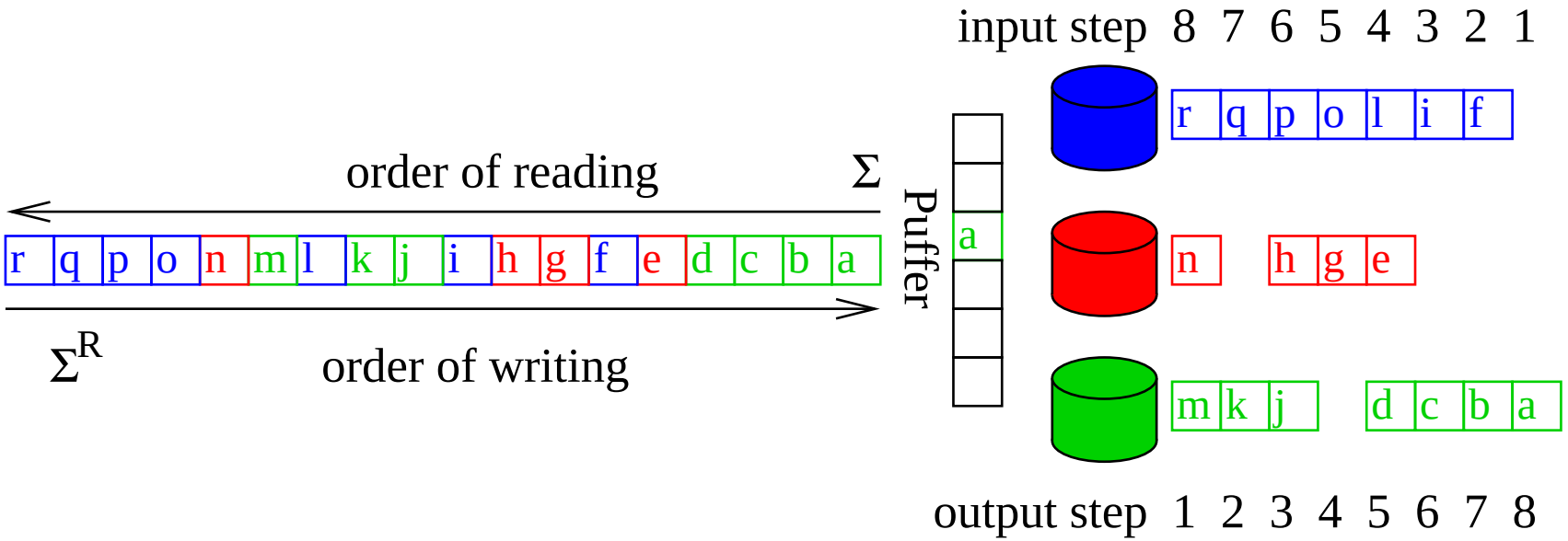
Theorem:

For buffer size W:

$\exists$ (offline) prefetching schedule for Σ with T input steps

$\Leftrightarrow$

$\exists$ (online) write schedule for Σ^R with T output steps



Synthesis

Multiway merging

+ prediction [60s Folklore]

+optimal (randomized) writing [Sanders Egner Korst SODA 2000]

+randomized cycling [Vitter Hutchinson 2001]

+optimal prefetching [Hutchinson Sanders Vitter ESA 2002]

$\rightsquigarrow (1+o(1))\cdot \text{sort}(n)$ I/Os

$\rightsquigarrow$"answers" question in [Knuth 98]; difficulty 48 on a 1..50 scale.

We are not done yet!

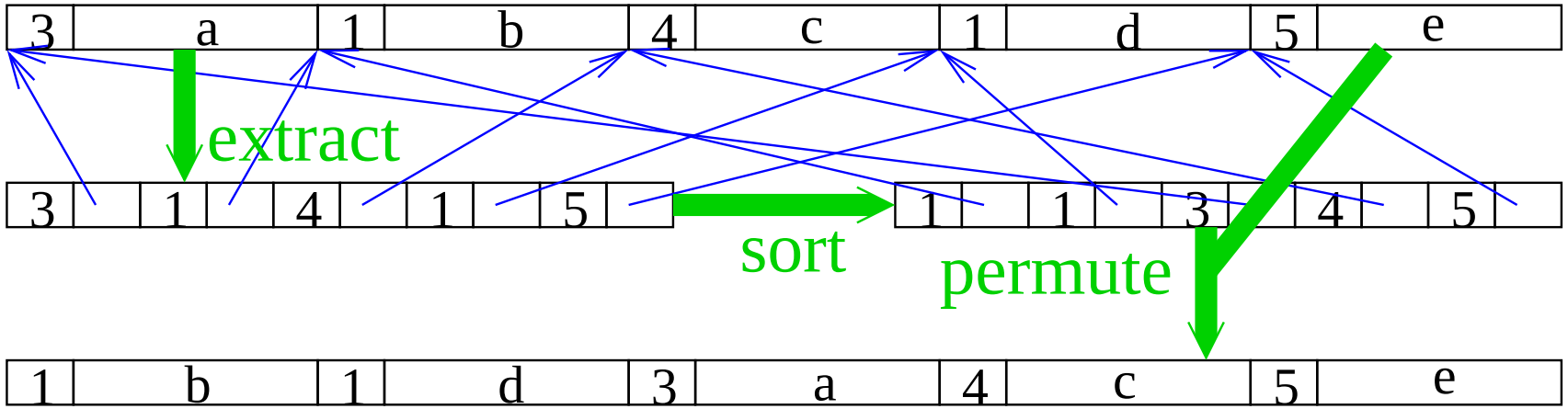
- Internal work
- Overlapping I/O and computation
- Reasonable hardware
- Interfacing with the Operating System
- Parameter Tuning
- Software engineering
- Pipelining

Key Sorting

The I/O bandwidth of our machine is about $1/3$ of its main memory bandwidth

$\rightsquigarrow$ If key size $\ll$ element size

sort key pointer pairs to save memory bandwidth during run formation



Tournament Trees for Multiway Merging

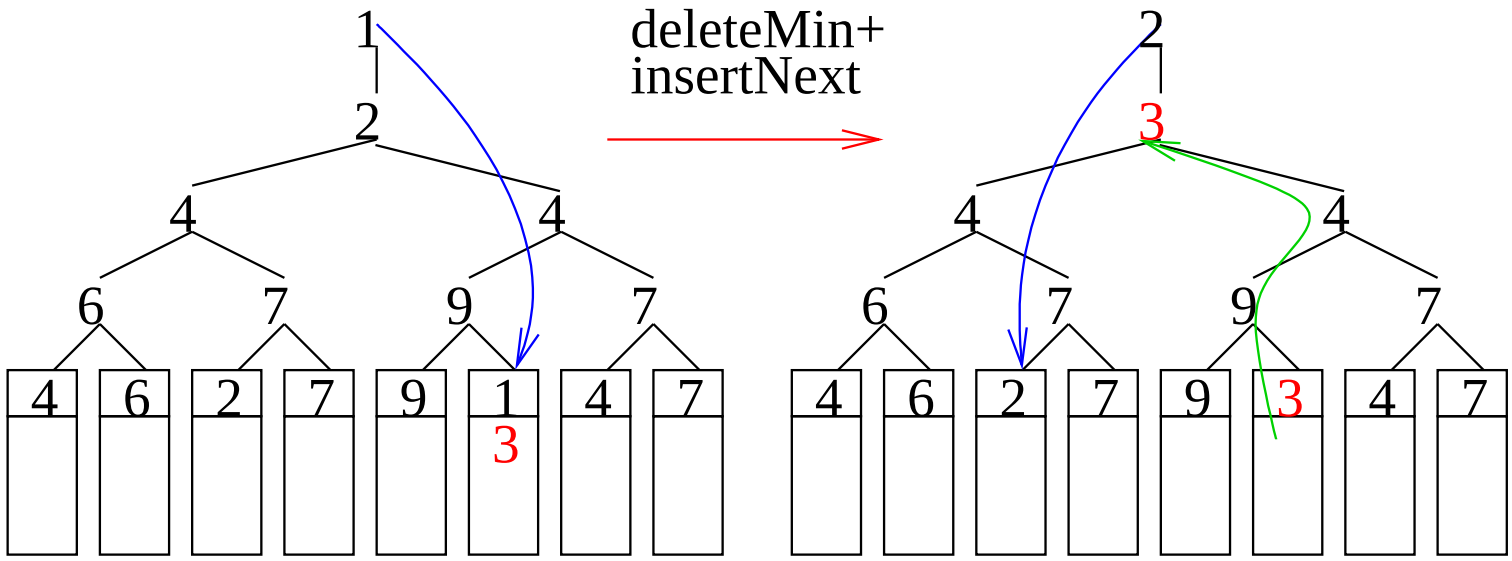
Assume $k = 2^K$ runs

K level complete binary tree

Leaves: smallest current element of each run

Internal nodes: loser of a competition for being smallest

Above root: global winner



Why Tournament Trees

- Exactly $\log k$ element comparisons
- Implicit layout in an array $\rightsquigarrow$ simple index arithmetics (shifts)
- Predictable load instructions and index computations

(Unrollable) inner loop:

```
for (int i=(winnerIndex+kReg)>>1; i>0; i>>=1){
  currentPos = entry + i;
  currentKey = currentPos->key;
  if (currentKey < winnerKey) {
    currentIndex      = currentPos->index;
    currentPos->key   = winnerKey;
    currentPos->index = winnerIndex;
    winnerKey         = currentKey;
    winnerIndex       = currentIndex;}}
```

Overlapping I/O and Computation

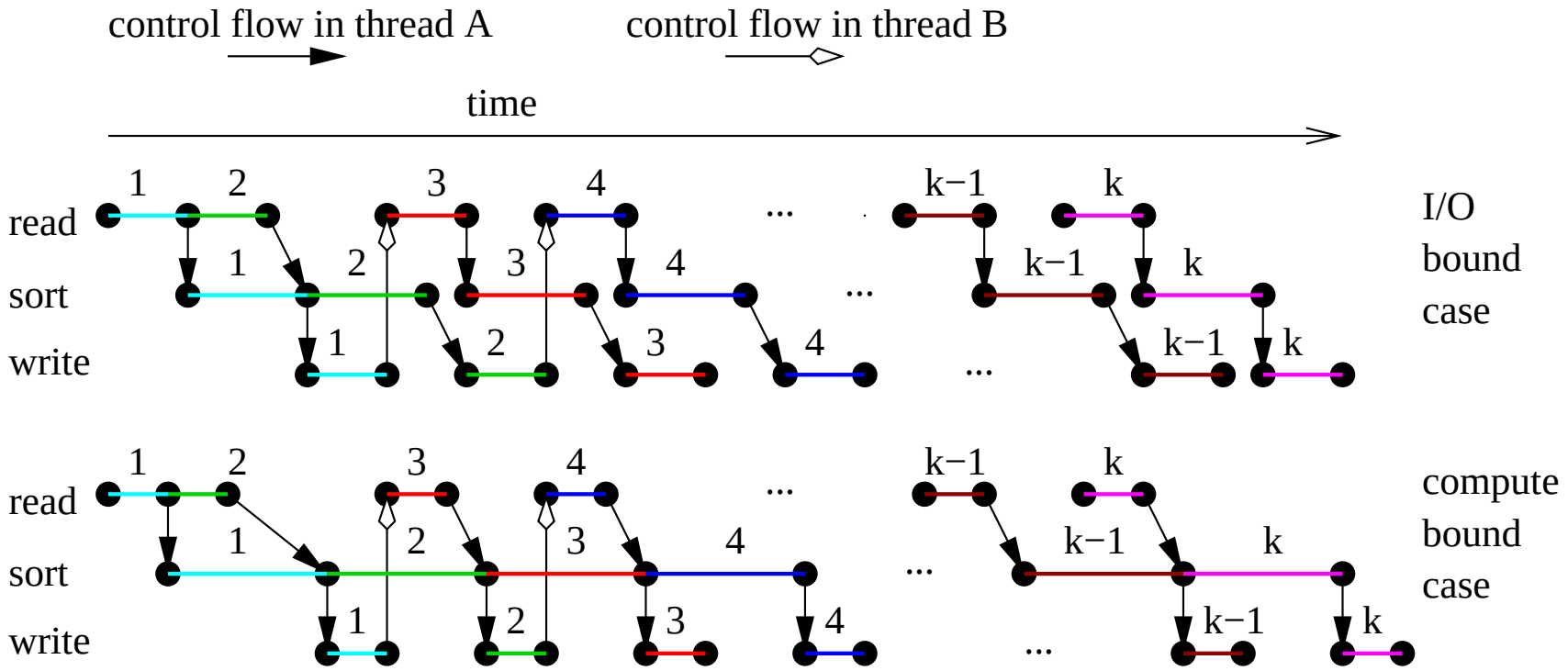
- One thread for each disk (or asynchronous I/O)
- Possibly additional threads
- Blocks filled with elements are passed by references between different buffers

Overlapping During Run Formation

First post read requests for runs 1 and 2

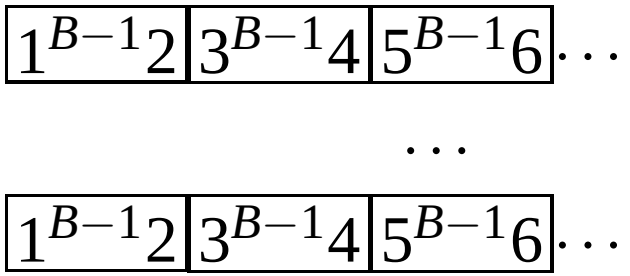
Thread A: Loop { wait-read i; sort i; post-write i};

Thread B: Loop { wait-write i; post-read $i+2$};

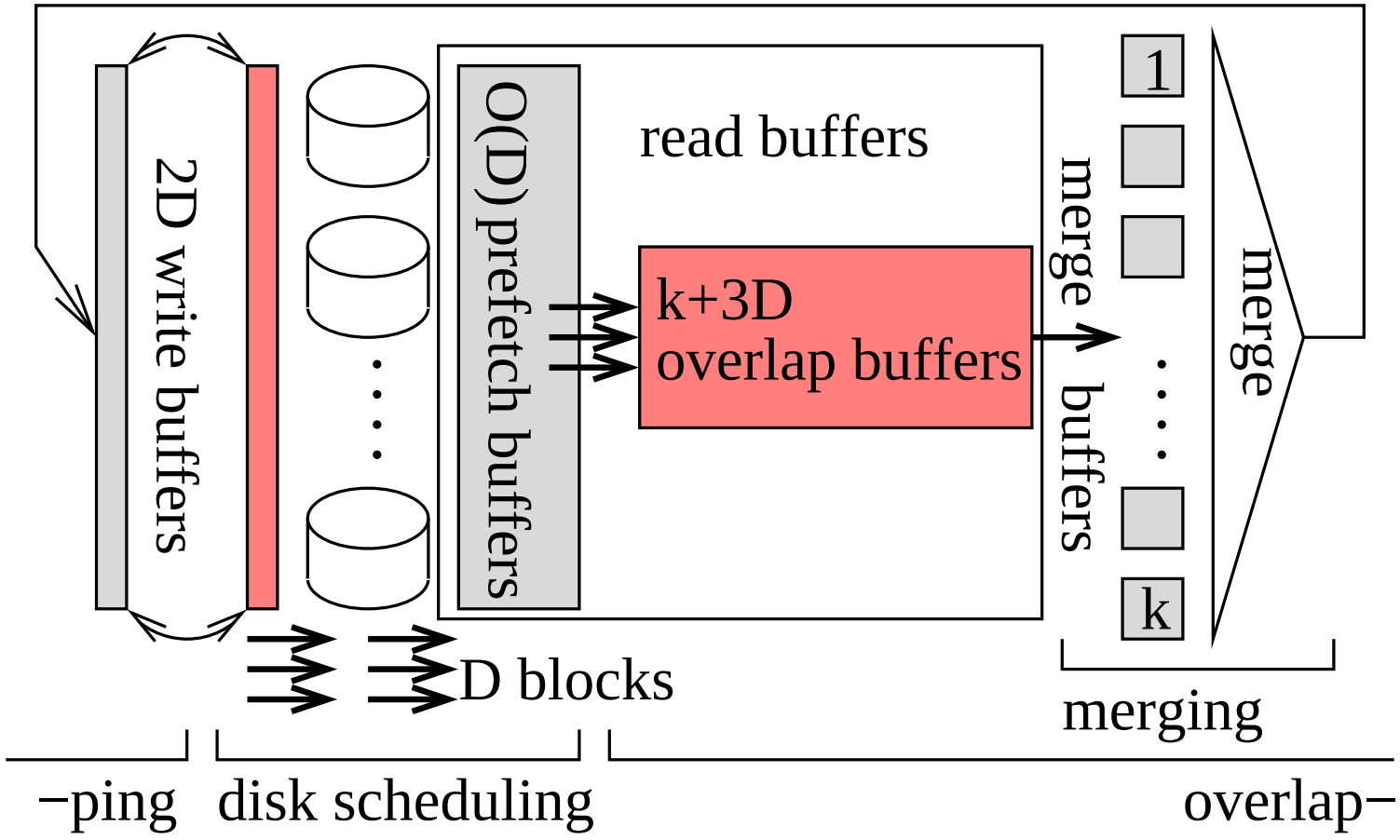


Overlapping During Merging

Bad example:



Overlapping During Merging



I/O Threads: Writing has priority over reading

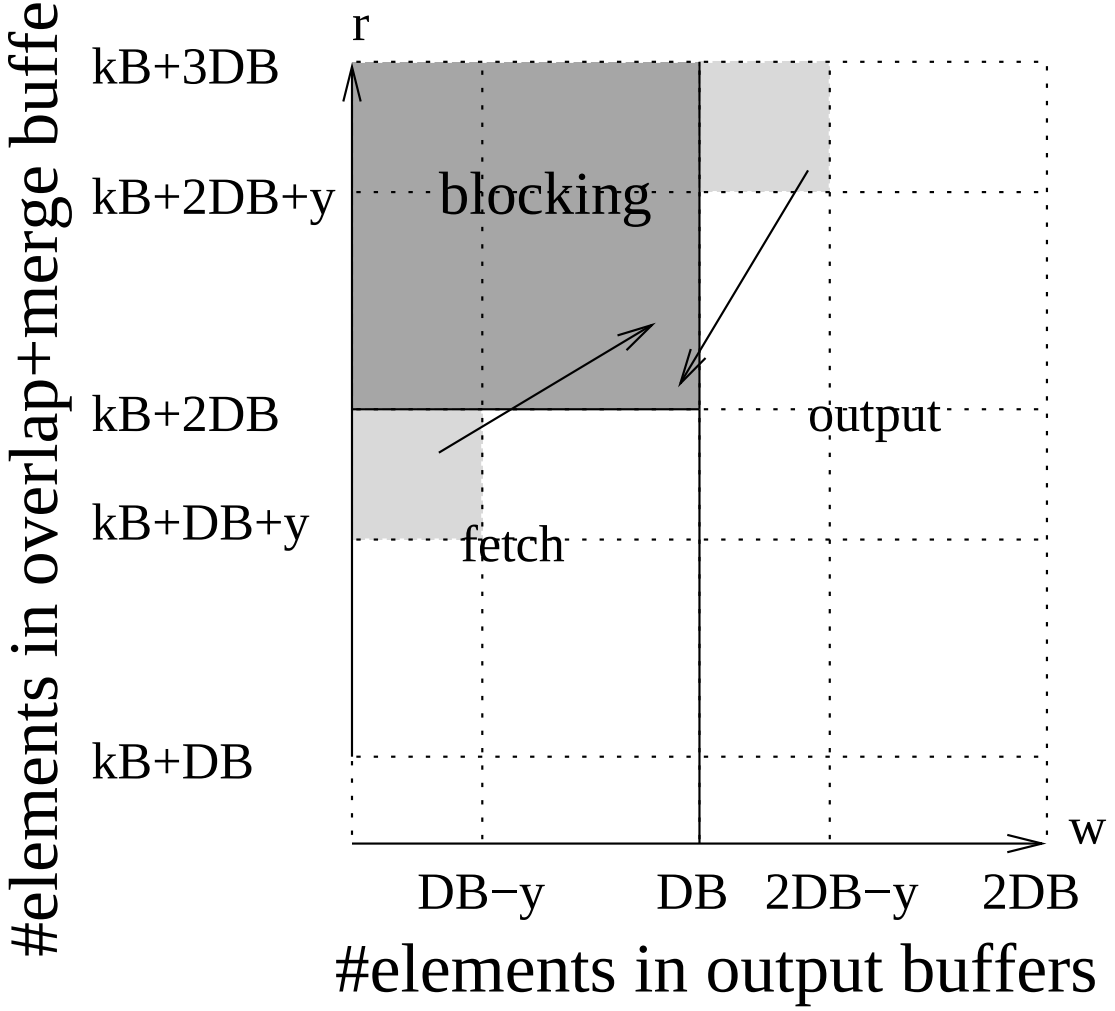
I/O bound case: The I/O thread never blocks

$y =$ # of elements merged during one I/O step.

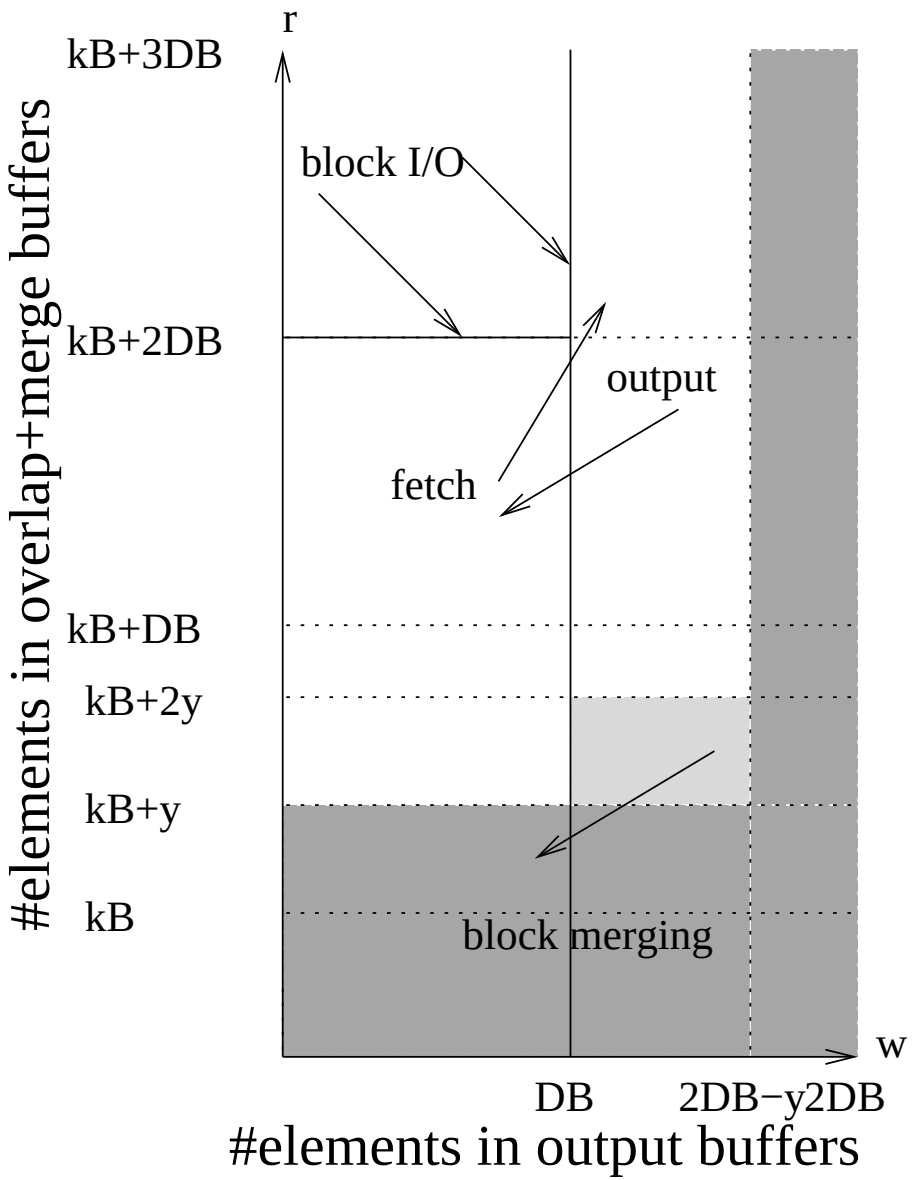
I/O bound $\rightsquigarrow$

$y > \frac{DB}{2}$

$y \leq DB$



Compute bound case: The merging thread never blocks



Hardware (mid 2002)

Linux

(2 × 2GHz Xeon × 2 Threads)

Several 66 MHz PCI-buses

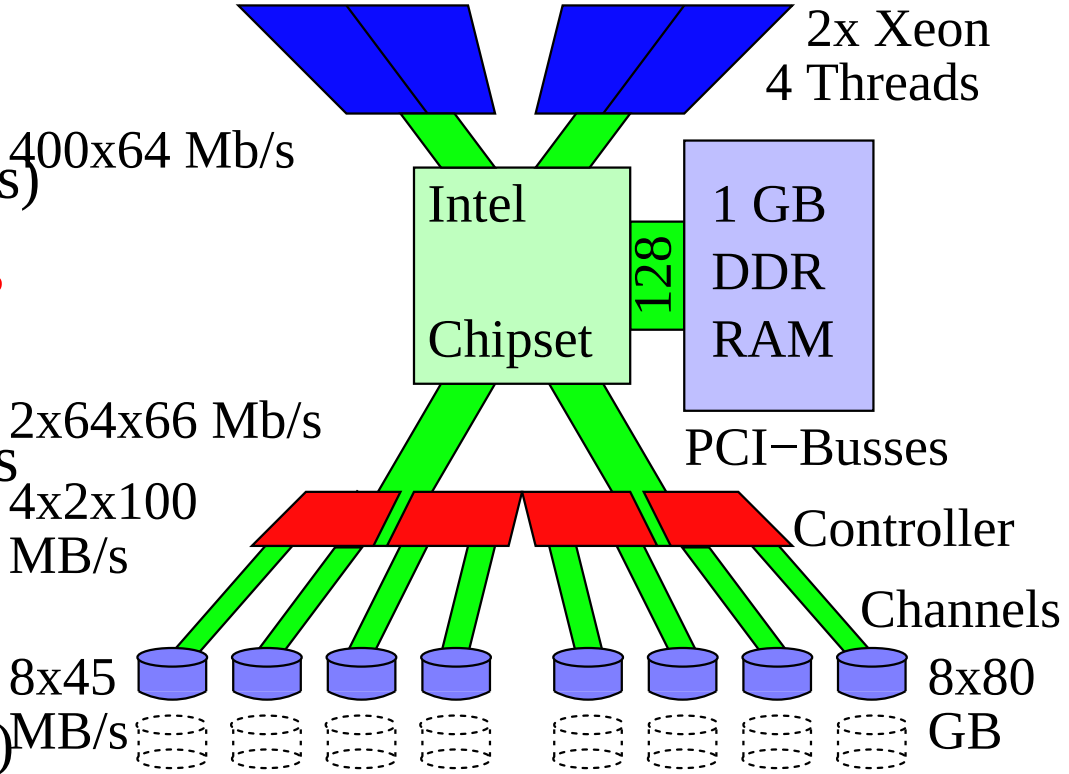
(SuperMicro P4DPE3)

Several fast IDE controllers

(4 × Promise Ultra100 TX2)

Many fast IDE disks

(8× IBM IC35L080AVVA07)

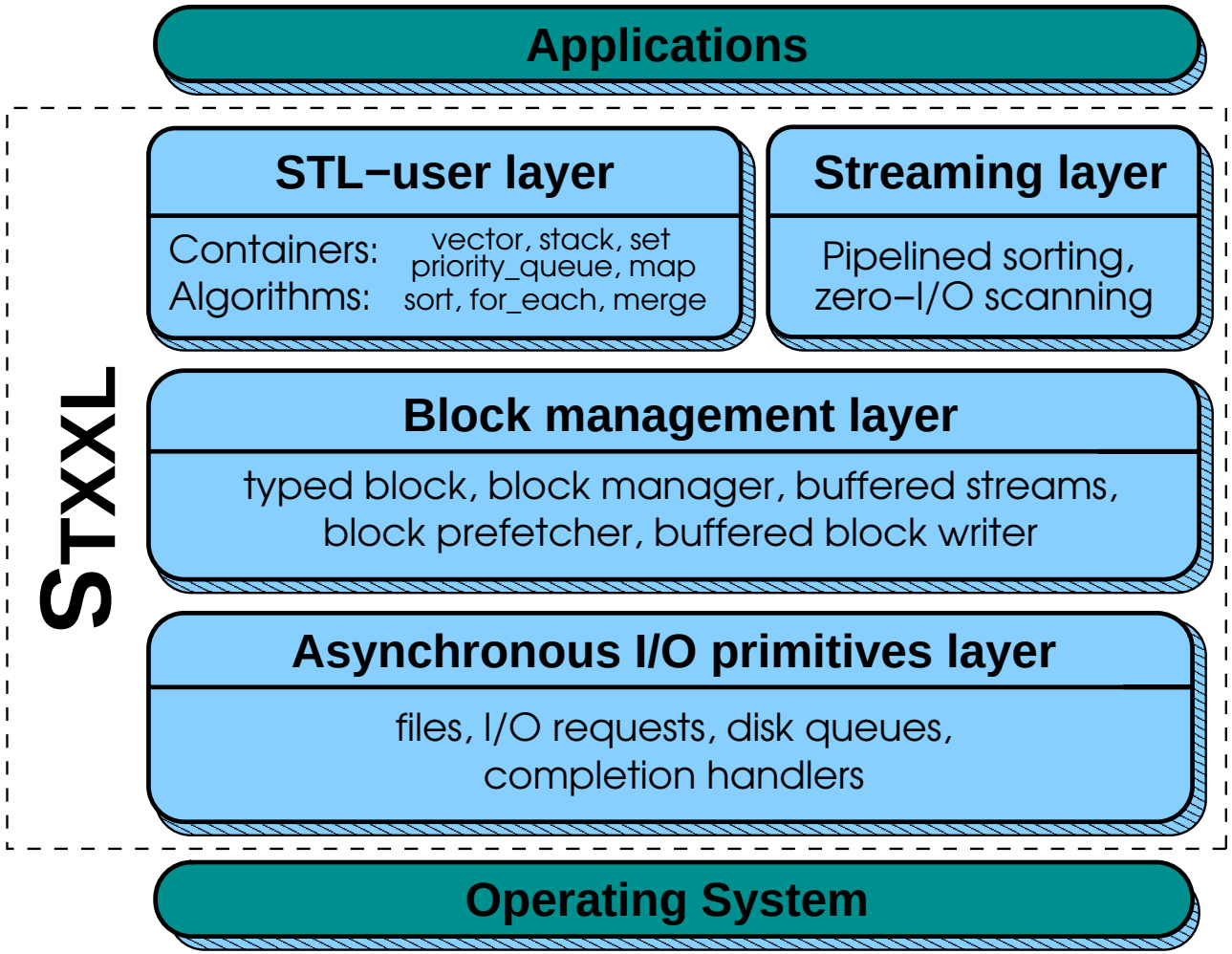


---

cost effective I/O-bandwidth (real 360 MB/s for ≈ 3000 €)

Software Interface

Goals: efficient + simple + compatible



Default Measurement Parameters

$t :=$ number of available buffer blocks

Input Size: 16 GByte

Element Size: 128 Byte

Keys: Random 32 bit integers

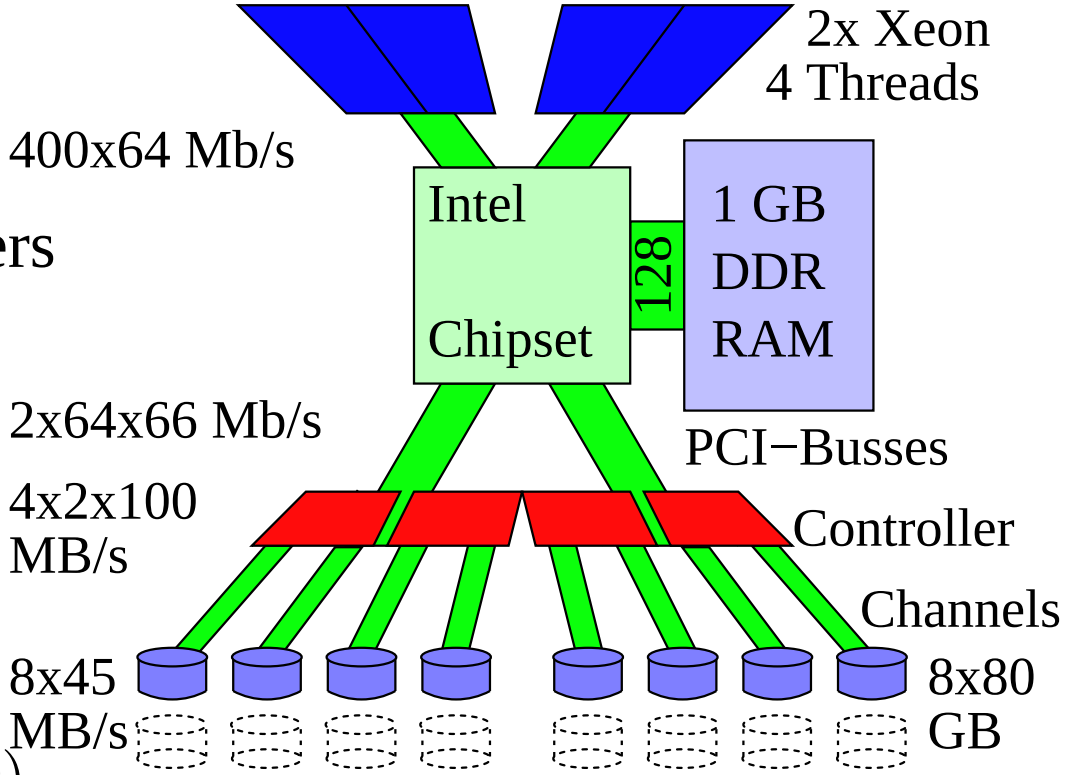
Run Size: 256 MByte

Block size B: 2 MByte

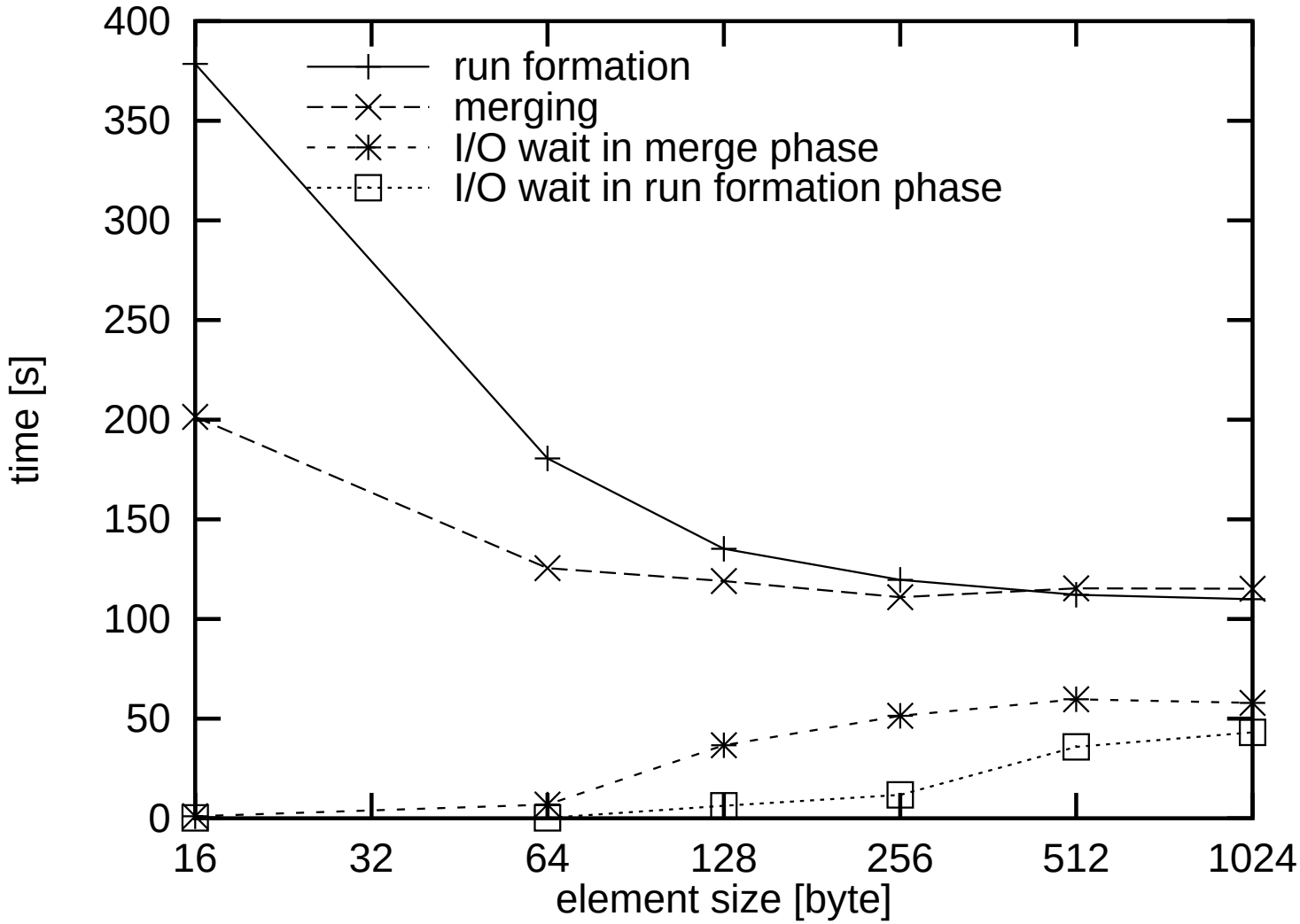
Compiler: g++ 3.2 -O6

Write Buffers: $\max(t/4, 2D)$

Prefetch Buffers: $2D + \frac{3}{10}(t - w - 2D)$



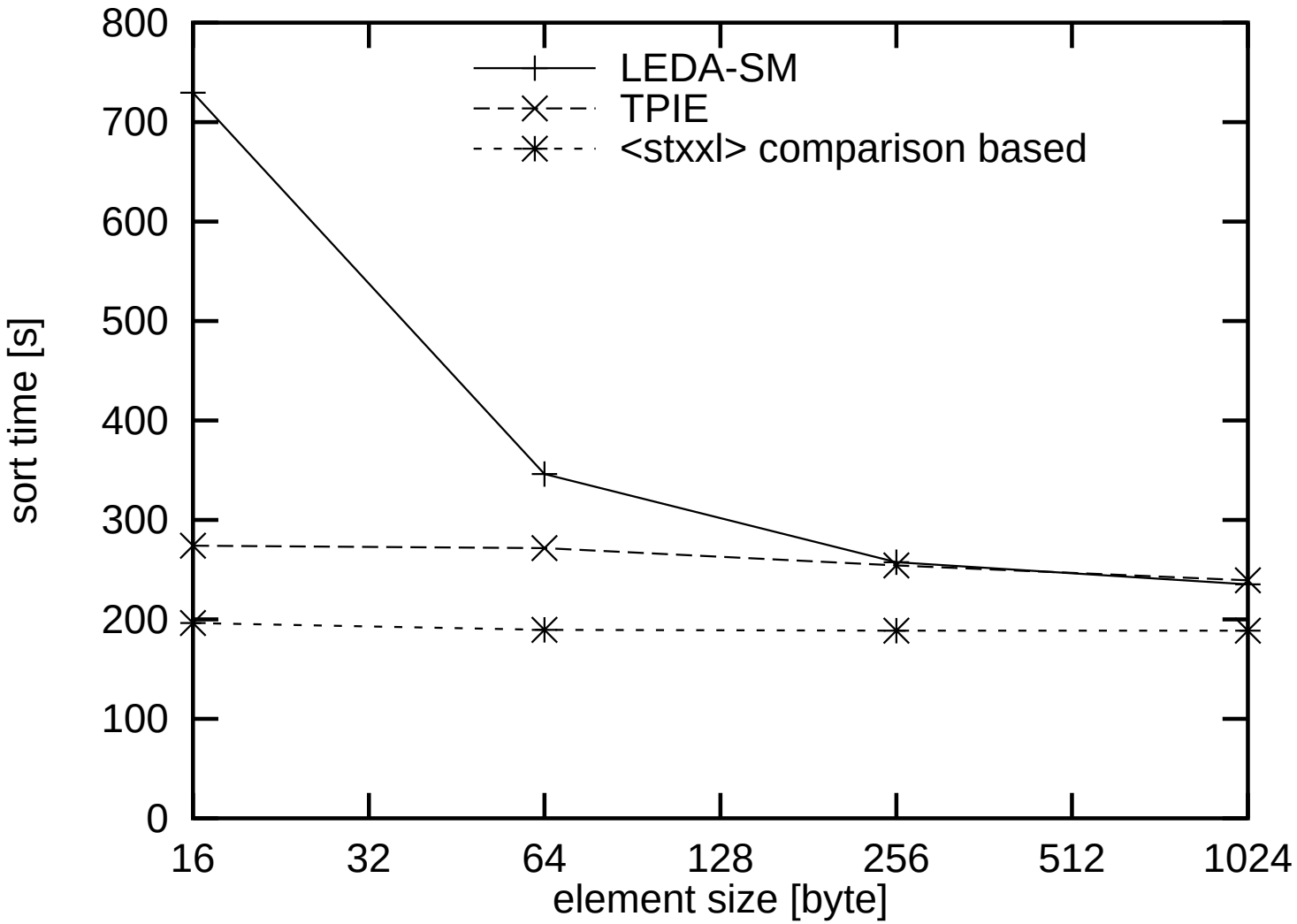
Element sizes (16 GByte, 8 disks)



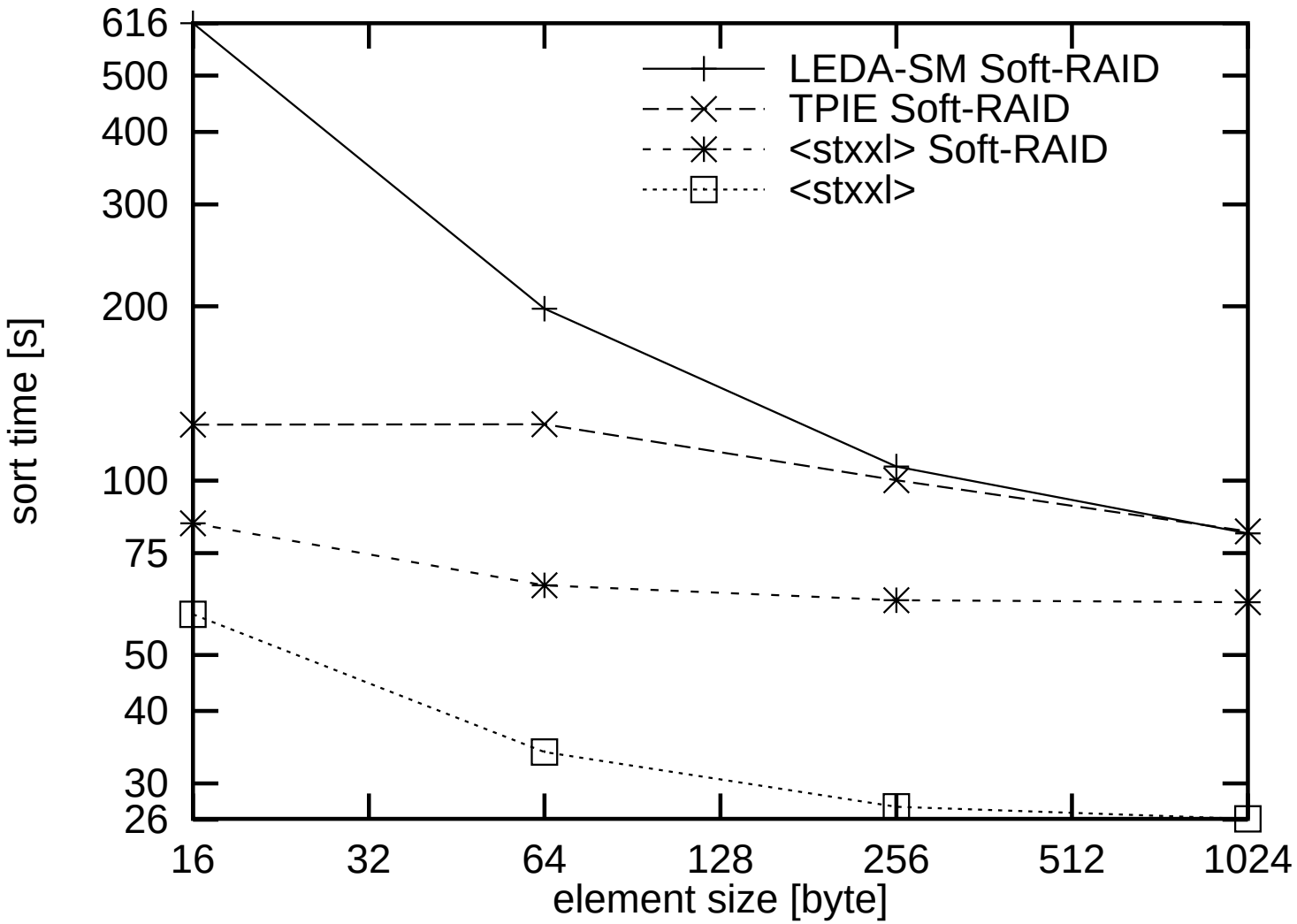
parallel disks $\rightsquigarrow$ bandwidth "for free" $\rightsquigarrow$ internal work, overlapping are relev

Earlier Academic Implementations

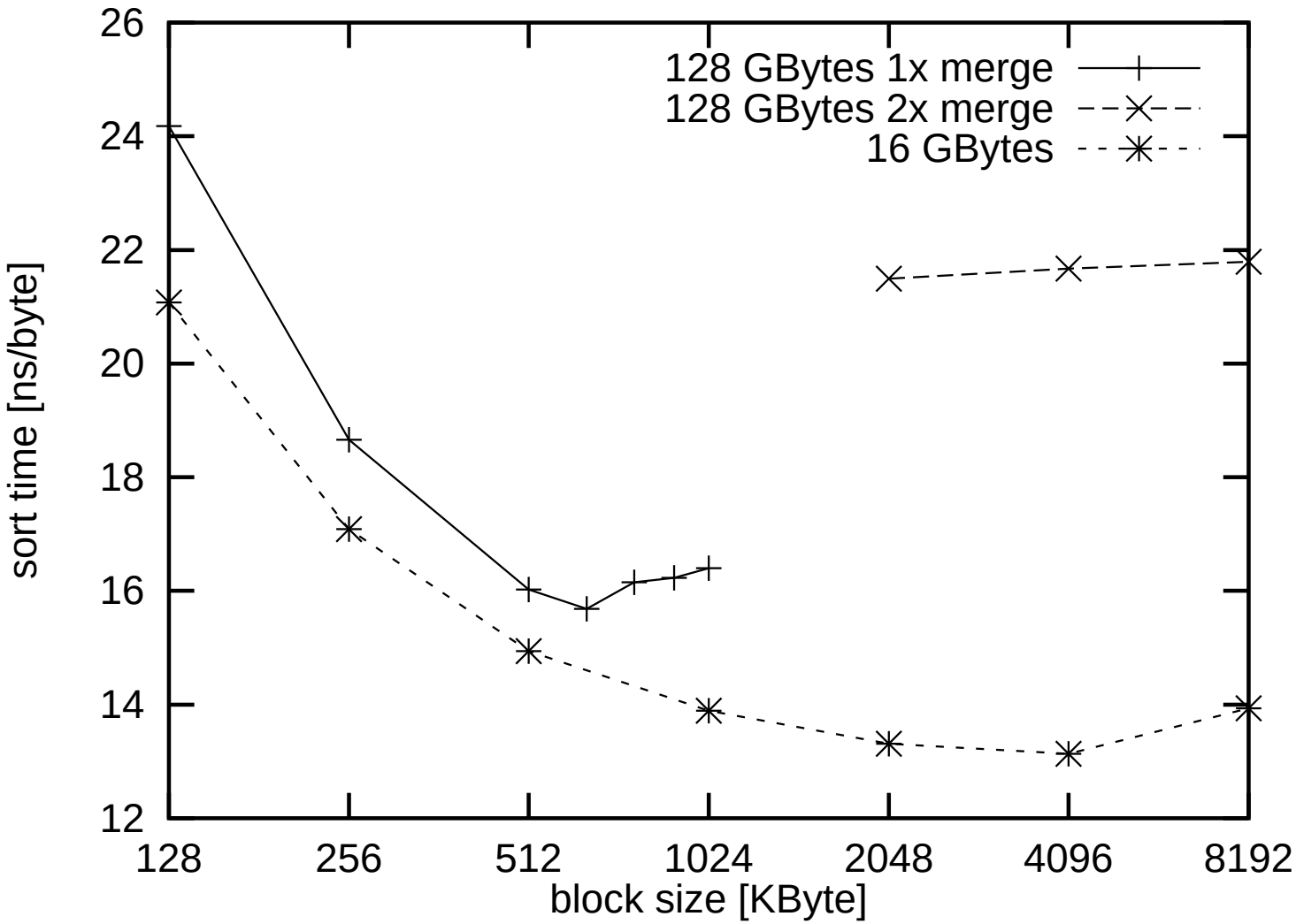
Single Disk, at most 2 GByte, old measurements use artificial M



Earlier Academic Implementations: Multiple Disks



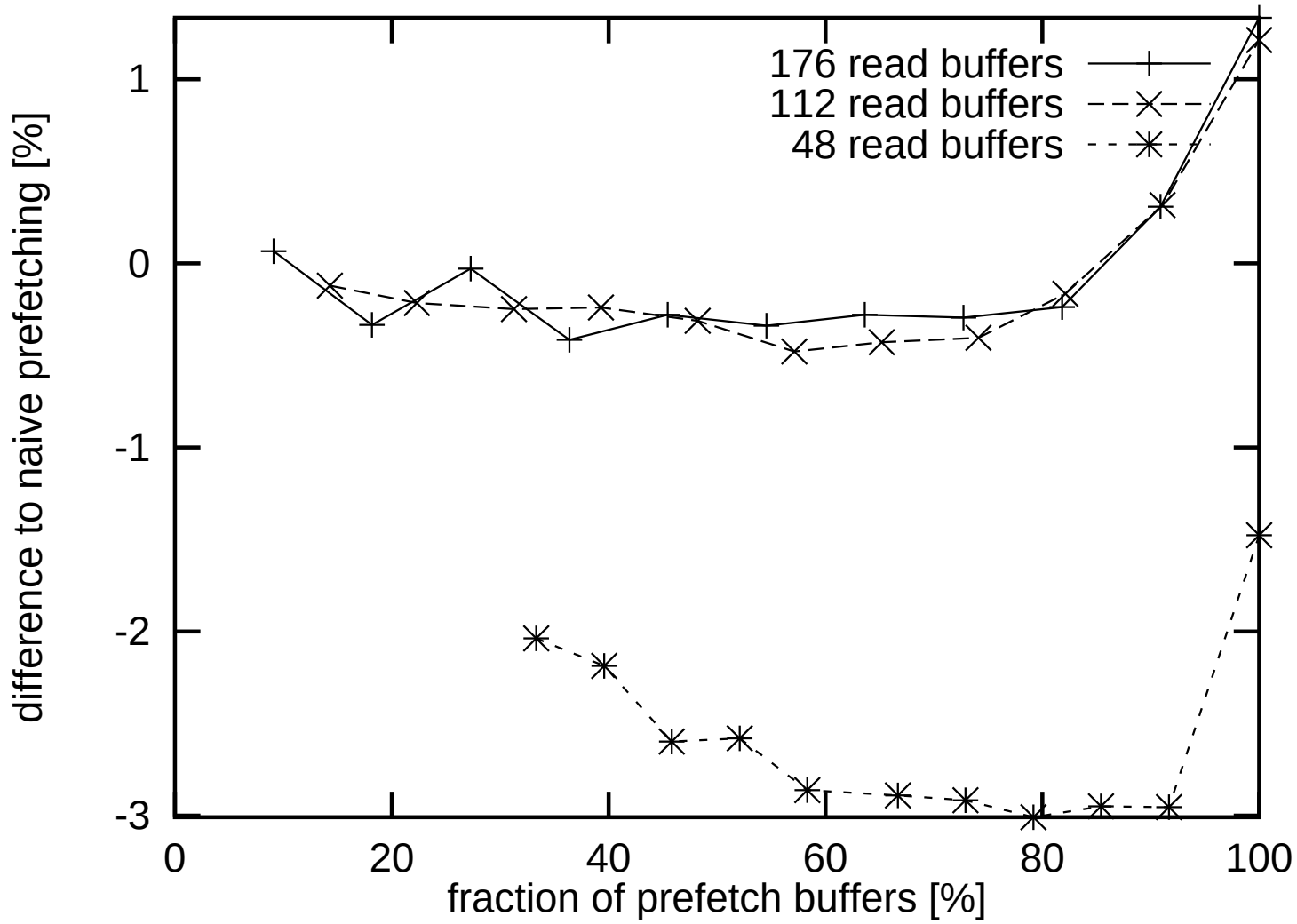
What are good block sizes (8 disks)?



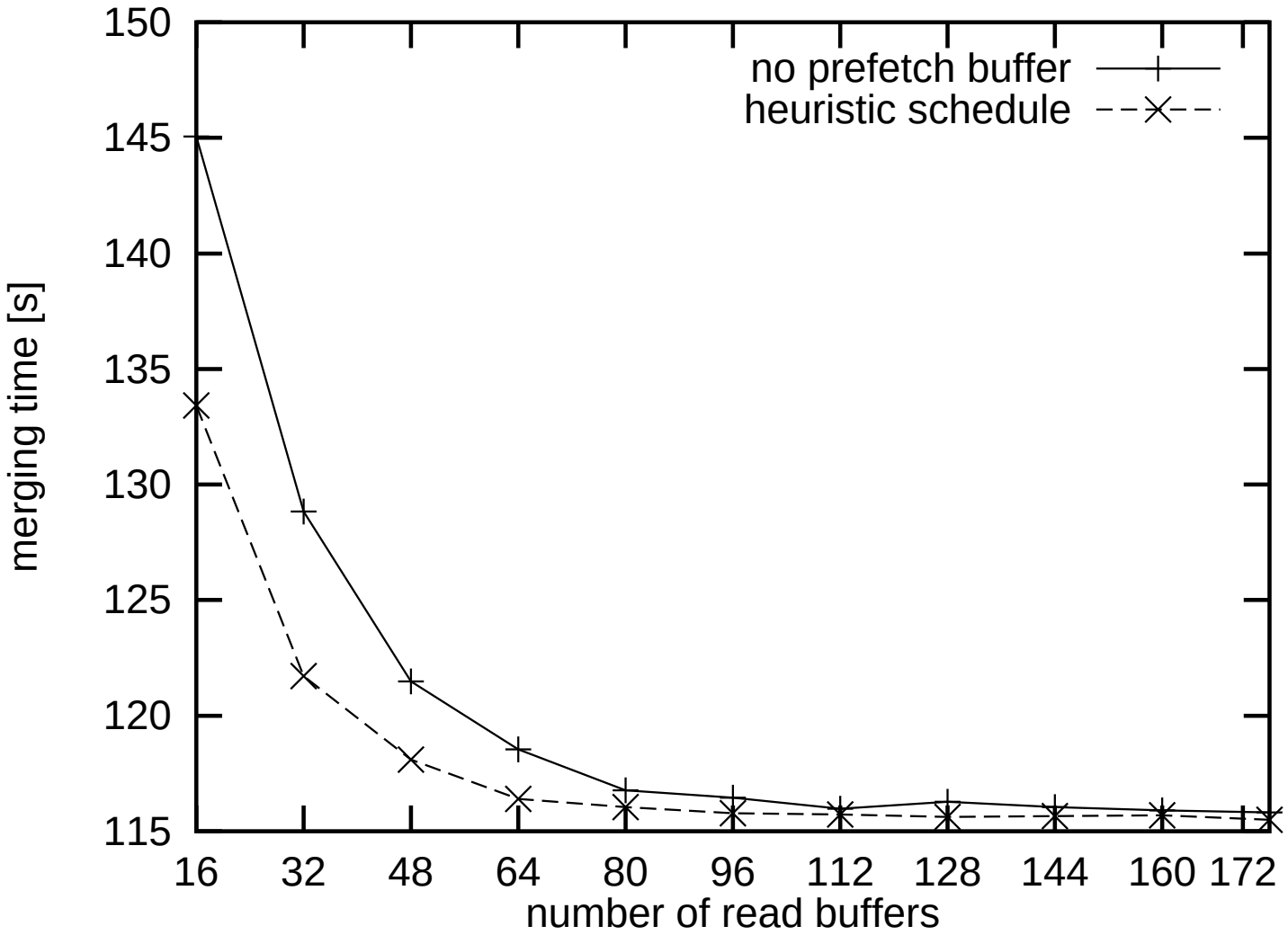
B is not a technology constant

Optimal Versus Naive Prefetching

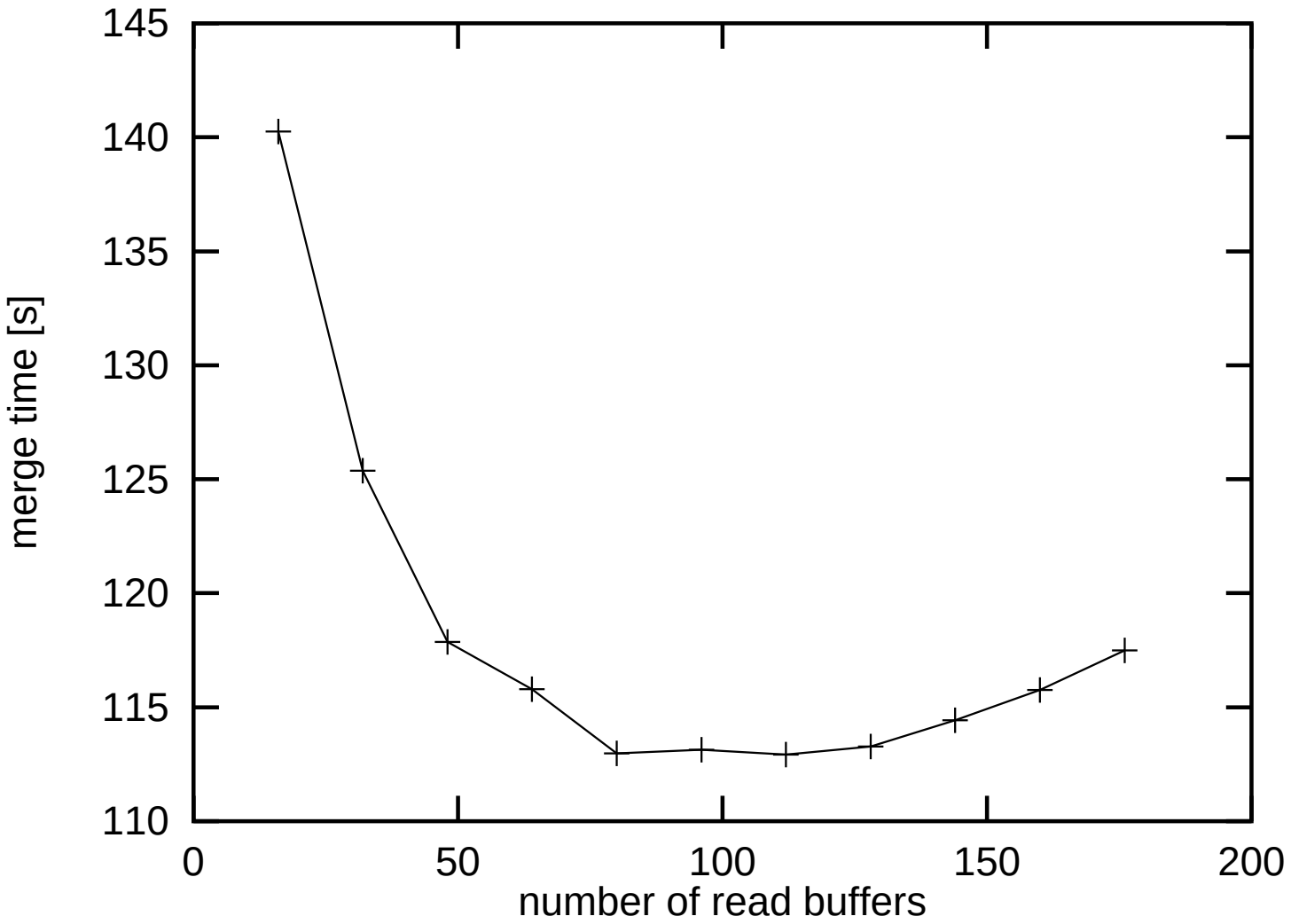
Total merge time



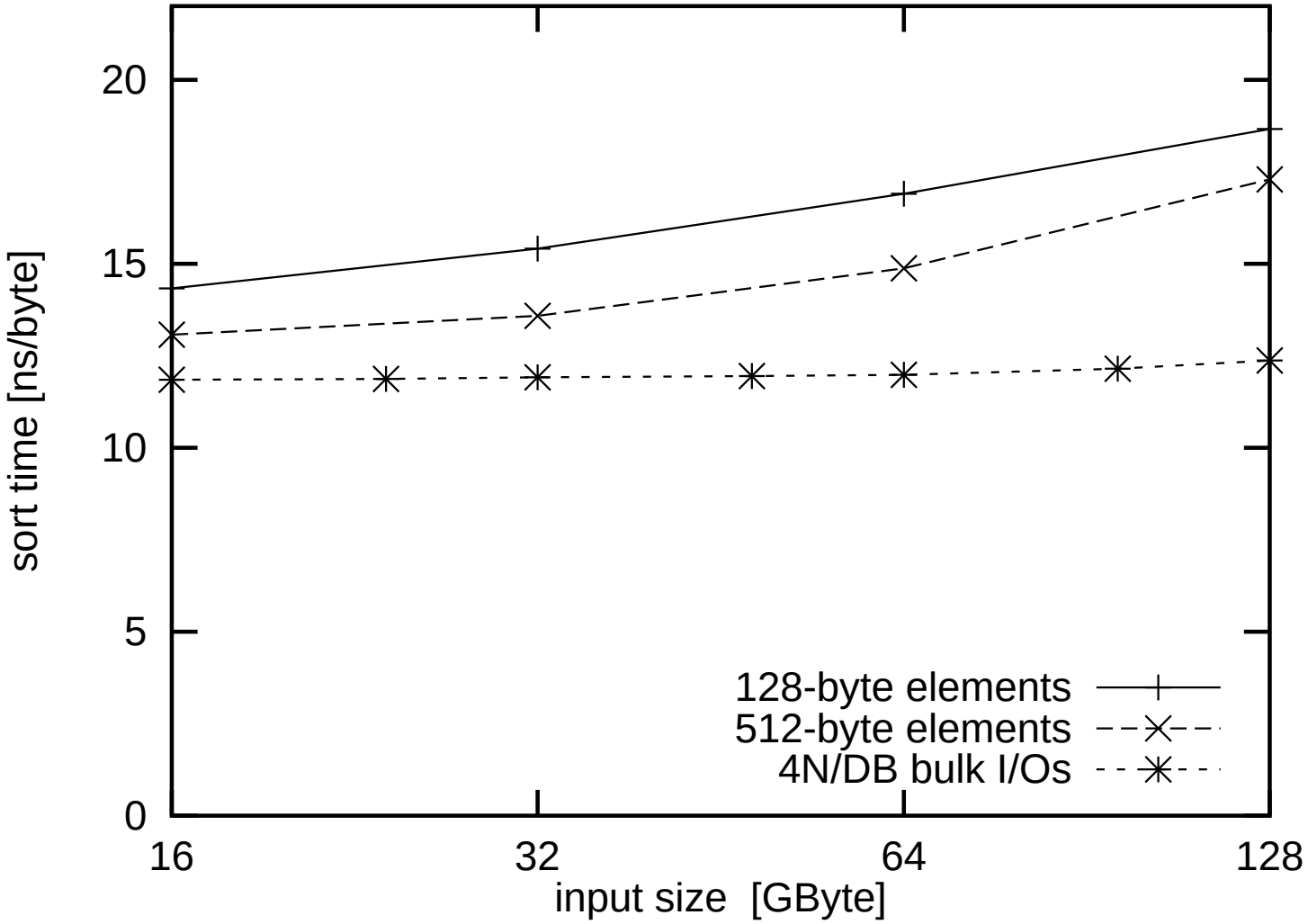
Impact of Prefetch and Overlap Buffers



Tradeoff: Write Buffer Size Versus Read Buffer Size



Scalability



Discussion

- Theory and practice harmonize
- No expensive server hardware necessary (SCSI,...)
- No need to work with artificial M
- No 2/4 GByte limits
- Faster than academic implementations
- (Must be) as fast as commercial implementations but with performance guarantees
- Blocks are much larger than often assumed. Not a technology constant
- Parallel disks $\rightsquigarrow$
  bandwidth "for free" $\rightsquigarrow$ don't negliect internal costs

More Parallel Disk Sorting?

Pipelining: Input does not come from disk but from a logical input stream. Output goes to a logical output stream

⇝ only half the I/Os for sorting

⇝ often no I/Os for scanning todo: better overlapping

Parallelism: This is the only way to go for really many disks

Tuning and Special Cases: ssssort, permutations, balance work between merging and run formation?...

Longer Runs: not done with guaranteed overlapping, fast internal sorting !

Distribution Sorting: Better for seeks etc.?

Inplace Sorting: Could also be faster

Determinism: A practical and theoretically efficient algorithm?

Procedure formLongRuns

q, q' : PriorityQueue

for $i := 1$ **to** M **do** q.insert(readElement)

invariant $|q| + |q'| = M$

loop

while $q \neq \emptyset$

writeElement($e :=$ q.deleteMin)

if input exhausted **then** break outer loop

if $e' :=$ readElement $< e$ **then** q'.insert(e')

else q.insert(e')

$q := q'$; $q' := \emptyset$

output q in sorted order; output q' in sorted order

Knuth: average run length $2M$

todo: cache-effiziente Implementierung