SIAM J. COMPUT.
Vol. 34, No. 6, pp. 1443–1463

DUALITY BETWEEN PREFETCHING AND QUEUED WRITING WITH PARALLEL DISKS[*]

DAVID A. HUTCHINSON[†], PETER SANDERS[‡], AND JEFFREY SCOTT VITTER[§]

Abstract. Parallel disks promise to be a cost effective means for achieving high bandwidth in applications involving massive data sets, but algorithms for parallel disks can be difficult to devise. To combat this problem, we define a useful and natural duality between writing to parallel disks and the seemingly more difficult problem of prefetching. We first explore this duality for applications involving read-once accesses using parallel disks. We get a simple linear time algorithm for computing optimal prefetch schedules and analyze the efficiency of the resulting schedules for randomly placed data and for arbitrary interleaved accesses to striped sequences. Duality also provides an optimal schedule for prefetching plus caching, where blocks can be accessed multiple times. Another application of this duality gives us the first parallel disk sorting algorithms that are provably optimal up to lower-order terms. One of these algorithms is a simple and practical variant of multiway mergesort, addressing a question that had been open for some time.

Key words. caching, external memory sorting, load balancing, lower bound, prefetching, randomized algorithm

AMS subject classifications. 68W10, 68W20, 68W40, 68M20, 68P10, 68P20, 68Q17

DOI. 10.1137/S0097539703431573

1. Introduction.

External memory (EM) algorithms are those for which the problem data set is too large to fit into the high-speed random access memory (RAM) of a computer and therefore must reside on external devices such as disk drives [23]. In order to cope with the high latency of accessing data on disks, efficient EM algorithms exploit locality in their design. In the *I/O model*, EM algorithms access a large *block* of B contiguous data elements in one *I/O step* and perform the necessary algorithmic operations on the elements in the block while in the high-speed memory. The speedup can be significant. However, even with blocked access, a single disk provides much less bandwidth than the internal memory. This problem can be mitigated by using multiple disks in parallel. For each input/output operation, one block is transferred between a fast memory of size M and each of the D disks. The algorithm therefore transfers D blocks at the cost of a single-disk access delay.

A simple approach to algorithm design for parallel disks is to employ large logical blocks, or *superblocks*, of size $B \cdot D$ in the algorithm. This reduces the problem to designing an EM algorithm for one disk with logical block size BD. A superblock is split into D physical blocks—one on each disk. All D physical blocks are accessed

[*] Received by the editors July 18, 2003; accepted for publication (in revised form) March 21, 2005; published electronically August 17, 2005. A substantial part of this work was done while the authors were at Duke University.

http://www.siam.org/journals/sicomp/34-6/43157.html

[†] Department of Systems and Computer Engineering, Carleton University, Ottawa K1S 5B6, ON, Canada (hutchins@sce.carleton.ca). This author's research was supported in part by the National Science Foundation through research grant CCR–0082986.

[‡] Fakultät für Informatik, Universität Karlsruhe, 76128 Karlsruhe, Germany (sanders@ira.uka.de). This author's research was partially supported by the Future and Emerging Technologies programme of the EU under contract IST-1999-14186 (ALCOM-FT).

[§] Department of Computer Sciences, Purdue University, West Lafayette, IN 47907–2066 (jsv@purdue.edu). This author's research was supported in part by the NSF through research grants CCR–9877133 and EIA–9870724 and by the Army Research Foundation through MURI grant DAAH04–96–1–0013 and grants DAAD19–01–1–0725 and DAAD19–03–1–0321.

simultaneously whenever the superblock is accessed. We refer to this technique as *superblock striping*. Unfortunately, this approach is suboptimal for em algorithms like sorting that deal with many blocks at the same time. For sorting and many related EM problems, an optimal algorithm requires *independent access* to the D disks, in which each of the D blocks in a parallel I/O operation can reside at a different position on its disk [25, 23]. Designing algorithms for independent parallel disks has been surprisingly difficult [25, 21, 20, 10, 11, 5, 6, 23, 22, 24]. In this paper we consider parallel disk output and parallel disk input separately, in particular as the parallel *output scheduling problem* and the parallel *prefetch scheduling problem*, respectively.

The (online) *output scheduling (or queued writing) problem* takes as input a fixed size pool of m (initially empty) memory buffers each capable of storing a block, and the sequence $\langle w_0, w_1, \ldots, w_{L-1}\rangle$ of L block *write requests* as they are issued. Each write request is prelabeled with the disk it will use. The solution of the output scheduling problem is a schedule that specifies when the blocks are output (i.e., the contents of each parallel output operation). The buffer pool can be used to reorder the outputs with respect to the logical writing order given by $\langle w_0, w_1, \ldots, w_{L-1}\rangle$ so that the total number of parallel output steps is minimized.

We use the term *write* for the logical process of moving a block from the responsibility of the application to the responsibility of the scheduling algorithm. The scheduling algorithm orchestrates the physical *output* of these blocks to disks.

The (offline) *prefetch scheduling problem* takes as input a fixed size pool of m (empty) memory buffers for storing blocks, and the sequence $\langle r_0, r_1, \ldots, r_{L-1}\rangle$ of L distinct block *read requests* that will be issued. Each read request is prelabeled with the disk it will use. The resulting *prefetch schedule* specifies when the blocks should be fetched so that they can be consumed by the application in the right order.

By the term *read*, we mean the logical process of moving a block from the responsibility of the scheduling algorithm to the application. We use the term *fetch* (or *prefetch*) to refer to the physical disk access.

The central theme in this paper is the *duality* between these two problems. Roughly speaking, an output schedule corresponds to a prefetch schedule with reversed time axis, and vice versa. The power of this idea is that computations in one domain can be analyzed via duality with respect to computations in the other domain.

In section 2, we formally introduce the duality principle for the case of distinct blocks to be written or read (*write-once* and *read-once* scheduling). In section 3, we derive an optimal write-once output scheduling algorithm and apply the duality principle to obtain an optimal read-once prefetch scheduling algorithm. In section 4, we modify the previous algorithm so that blocks are fetched as early as possible, so as to be more robust against delays in practical implementations.

For difficult input sequences, an optimal schedule might use parallel disks very inefficiently because most disks might still be idle most of the time. In section 5, we therefore give performance guarantees for two particular classes of input sequences: randomly placed data and arbitrarily interleaved data streams. A data stream is a sequence of blocks that is read or written sequentially by the application. Many algorithms access several such streams in an interleaved manner, and the order of accesses to the streams is not predictable at the time the streams are allocated. Nevertheless, we obtain performance guarantees for the following allocation strategies of the data streams:

Fully randomized (FR): Each block is allocated to a random disk.

Striping (S): Consecutive blocks of a data stream are allocated to consecutive disks

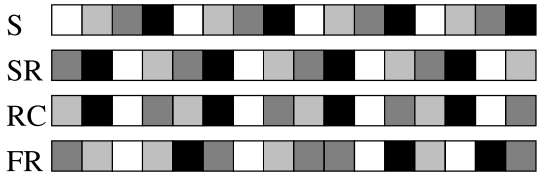

FIG. 1.1. *A sequence of* 16 *blocks allocated to* 4 *disks* (1 = *white,* 2 = *light grey,* 3 = *dark grey,* 4 = *black*) *using different allocation strategies.*

in a simple, round-robin manner.

Simple randomized (SR): For each data stream, this strategy follows a striping allocation, where the disk selected for the first block is chosen randomly and independently of the other data streams.

Randomized cycling (RC): Each data stream i chooses a random (and independent) permutation π_i of disk numbers and allocates the jth block of stream i on disk $\pi_i(j \bmod D)$.

Figure 1.1 gives an example.

In section 6, we relax the restriction that blocks are accessed only once and allow caching of blocks and repeated block requests (*write-many* and *read-many* scheduling). Again we derive a simple optimal algorithm for the writing case and obtain an optimal algorithm for the reading case using the duality principle. A similar result has been obtained by Kallahalla and Varman [16, 17] using more complicated arguments.

In section 7, we apply the results from sections 3 and 5 to parallel disk sorting. Results on online writing translate into improved sorting algorithms using the distribution paradigm. Results on offline reading translate into improved sorting algorithms based on multiway merging. By appending a "D" for distribution sort or an "M" for mergesort to an allocation strategy (FR, S, SR, RC) we obtain a descriptor for a sorting algorithm (FRD, FRM, SD, SM, SRD, SRM, RCD, RCM). This notation is an extension of the notation used in [24]. RCD and RCM turn out to be particularly efficient. Let

$$\mathrm{Sort}(N) = \frac{N}{DB}\left(1 + \log_{M/B}\frac{N}{M}\right).$$

In section 8, we show that $2 \cdot \mathrm{Sort}(N)$ is the lower bound for sorting N elements on D disks. Our versions of RCD and RCM are the first algorithms that provably match this bound up to a lower-order term if $M = \omega(DB)$. The good performance of RCM is particularly interesting. The question of whether there is a simple variant of mergesort that is asymptotically optimal for multiple disks has been open since the model was formalized in [25]. A summary of the notation used in this paper is included in the appendix.

Related work. An announcement [14] and a preliminary version [13] of this paper have appeared in conference volumes. The problems of prefetching and caching have been intensively studied and can be quite difficult. We begin our overview with offline algorithms for the I/O model. Belady [7] solved the caching problem for a single disk. Kallahalla and Varman [15, 17] developed an optimal parallel disk prefetching algorithm for read-once sequences. Besides a simpler algorithm and analysis, our contribution is a linear time algorithm (which is, however, also implicit in [16], which was published simultaneously with the first announcement of our result [14].) Moreover, the concept of duality allows us to translate performance guarantees for writing into performance guarantees for reading. The main contribution in [16] is an optimal result for prefetching plus caching. Again, the concept of duality yields a simpler algorithm and proof.

Cao et al. [9] and Kimbrel and Karlin [18] have introduced a model that allows us to study integrated prefetching and caching with overlapping of I/O and computation. In this *penalty model*, internal computation is linked to I/Os by a penalty of F time units for an I/O step. For $F \rightarrow \infty$, the penalty model becomes equivalent to the I/O model since the internal computations become insignificant. Kimbrel and Karlin [18] already introduced the idea of time reversal and the *reverse aggressive* algorithm that has our algorithm as a special case. They also defined a similar kind of duality, namely between fetches and evictions of a caching algorithm. The analysis in the penalty model predicts that the performance ratio between reverse aggressive and the optimal algorithm goes to infinity as $F \rightarrow \infty$. Hence it is a bit surprising that the algorithm turns out to be *optimal* in the I/O model.

The prudent prefetching algorithm introduced in section 4 is similar to the *conservative* algorithm described in [9]. The main difference is that it applies to the optimal parallel disk prefetching algorithm rather than to the optimal schedule in a sequential system.

Albers, Garg, and Leonardi [4] gave an optimal polynomial time offline algorithm for the single-disk case in the penalty model, but it does not generalize well to multiple disks. Albers and Büttner [3] overcame this problem by requiring synchronized parallel disk access (as in the I/O model) and by postulating $\mathcal{O}(D)$ additional buffer blocks not available to the optimal algorithm. Both these algorithms are based on linear programming and hence are quite complicated and time consuming.

There has also been intensive work on online integrated prefetching and caching. Albers [2] showed for a single disk that a lookahead for the next $\Omega\left(M/B\right)$ *different* blocks is needed to get good competitiveness. For parallel disks, Kallahalla and Varman [15, 17] showed that another factor of $\Omega\left(D\right)$ lookahead is needed even for the read-once problem. Applying the optimal offline algorithm to the lookahead matches this lower bound for read-once sequences.

There are deterministic algorithms for parallel disk external sorting [21, 20] that are optimal up to a constant factor, but the constant factors are not ideal. The first optimal (up to a constant factor independent of the parameters N, M, B, and D) sorting algorithm was a randomized one by Vitter and Shriver [25]. Barve, Grove, and Vitter [5] and Barve and Vitter [6] introduced a simple and efficient randomized sorting algorithm called *simple randomized mergesort (SRM)*. For each run, SRM allocates blocks to disks using the SR allocation discipline. For $\gamma < 1$, SRM comes within an additive term of about $\gamma\text{Sort}(N)$ of the sorting lower bound if $M/B = \Omega(D\log(D)/\gamma^2)$, but for $M/B = o(D\log D)$, the bound proven is not asymptotically optimal. It is an open problem whether SRM or another variant of striped mergesort could be asymptotically optimal for small internal memory. Knuth [19, Exercise 5.4.9–31] gives the question of a tight analysis of SR a difficulty rating of 48 on a scale between 1 and 50.

Sanders, Egner, and Korst [22] analyzed a (slightly) suboptimal output scheduling algorithm for FR allocation that would yield a good parallel disk distribution sorting algorithm (FRD) yet has the disadvantage that reading cannot be done in a striped fashion. To overcome the apparent difficulty of analyzing SR, Vitter and Hutchinson [24] analyzed RC allocation, which provides more randomness but retains the advantages of striping. RCD is an asymptotically optimal distribution sort algorithm that allocates successive blocks of a bucket to the disks according to the RC discipline. The present paper uses the concept of duality to apply these results to external mergesort.

The lower bound in section 8 is a refinement of the analysis by Aggarwal and Vitter [1]. In particular, our analysis gives the precise constant factor in the leading term, and it applies to algorithms that do not necessarily use the same number of inputs and outputs. The remaining gap between the upper and lower bound is only a lower-order term if $M = \omega(DB)$.

Building on the results in the present paper, Dementiev and Sanders [12] develop a parallel disk external sorting algorithm that perfectly overlaps I/O and computation. This algorithms works independently of the failure penalty F and need not know how much internal work is done between I/O requests. (These times are far from constant and not easy to predict so that previous results on prefetching in the penalty model are inapplicable for sorting.)

2. The duality principle.

Duality is a quite simple yet powerful concept once the model is properly defined. Therefore, we start with a more formal description of the model.

Our machine model is the (independent parallel disk) I/O model of Vitter and Shriver [25] with a single[1] processor, D disks, and an internal memory of size M. All blocks have the same size B. In one *I/O step*, one block on each disk can be accessed in a synchronized fashion. We consider either a queued writing or a buffered prefetching arrangement, where a pool of m block buffers is available to the algorithm (see Figure 2.1).

DEFINITION 2.1. *A* write-once output scheduling problem *is defined by a sequence* $\Sigma = \langle b_0, \ldots, b_{L-1}\rangle$ *of distinct blocks which are to be output using parallel output operations. Let* $\mathrm{disk}(b_i)$ *denote the disk on which block* b_i *is to be located. An application* writes *these blocks in the order specified by* Σ*. We use the term* write *for the logical process of moving a block from the responsibility of the application to the responsibility of the scheduling algorithm. The scheduling algorithm orchestrates the physical* output *of these blocks to disks. Time is measured in I/O steps actually performed. In particular, in each time step at least one block is output.*

An output schedule is specified by giving a function $\mathrm{oStep} : \{b_0, \ldots, b_{L-1}\} \to \mathbb{N}$ *that specifies for each disk block* $b_i \in \Sigma$ *the time step when it will be output. An output schedule is* correct *if the following conditions hold:*

(i) *No disk is referenced more than once in a single time step. That is, if* $i \neq j$ *and* $\mathrm{disk}(b_i) = \mathrm{disk}(b_j)$*, then* $\mathrm{oStep}(b_i) \neq \mathrm{oStep}(b_j)$*.*

(ii) *The buffer pool is large enough that it does not overflow. That is, if we define*

$$\mathrm{o\mathcal{B}acklog}(b_i) = |\{j < i : \mathrm{oStep}(b_j) \geq \mathrm{oStep}(b_i)\}|$$

to be the number of blocks b_j *that are written before block* b_i *but not output before* b_i*, then we require for all* $0 \leq i \leq L$ *that* $\mathrm{o\mathcal{B}acklog}(b_i) < m$*.*

The blocks are output in increasing order of oStep*. The number of steps needed by an output schedule is* $T = \max_{0 \leq i < L} \mathrm{oStep}(b_i)$*. An output schedule is* optimal *if it minimizes* T *among all correct schedules.*

[1] Our results generalize to multiple processors as long as data exchange between processors is much faster than disk access.

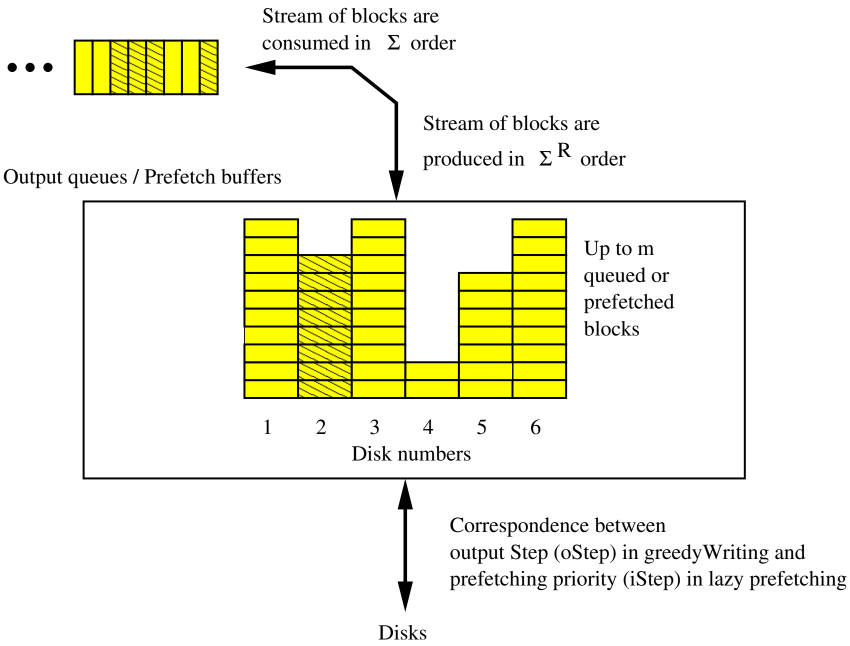
Stream of blocks are consumed in Σ order
Stream of blocks are produced in Σ R order
Output queues / Prefetch buffers
Up to m queued or prefetched blocks
1
2
3
4
5
6
Disk numbers
Correspondence between output Step (oStep) in greedyWriting and prefetching priority (iStep) in lazy prefetching
Disks

FIG. 2.1. *Duality between the prefetching priority and the output step. The hashed blocks illustrate how the blocks of disk* 2 *might be distributed.*

It will turn out that our write-once output scheduling algorithms work optimally even if they are given the blocks *online*, that is, one at a time without specifying Σ explicitly.

DEFINITION 2.2. *Analogously to a write-once output scheduling problem, a* read-once prefetch scheduling problem *is defined by a sequence Σ of blocks to be read. By the term* reading, *we mean the logical process of moving a block from the responsibility of the scheduling algorithm to the application. We use the term* fetching *(or* prefetching*) to refer to the physical disk access.*

A prefetch schedule is defined using a function $\mathrm{iStep} : \{b_0, \dots, b_{L-1}\} \to \mathbb{N}$. *The blocks are prefetched in increasing order of* iStep. *Let us define*

$$\mathrm{i}\mathcal{B}\mathrm{acklog}(b_i) = |\{j > i : \mathrm{iStep}(b_j) \le \mathrm{iStep}(b_i)\}|$$

to be the number of blocks b_j that are fetched no later than block b_i but are read after b_i. All blocks in $\mathrm{i}\mathcal{B}\mathrm{acklog}(b_i)$ *must be buffered. The limited buffer pool size requires the correctness condition* $\mathrm{i}\mathcal{B}\mathrm{acklog}(b_i) < m$. *The number of steps needed by a prefetch schedule is* $T = \max_{0 \le i < L} \mathrm{iStep}(b_i)$. *A prefetch schedule is* optimal *if it minimizes T among all correct schedules.*

It will turn out that our prefetch scheduling algorithms work *offline*; that is, they need to know Σ in advance. We explain in section 7 how this is sufficient for sorting applications.

The following theorem shows that reading and writing not only have similar models but are equivalent to each other in a quite interesting sense.

THEOREM 2.3 (duality principle). *Consider any sequence* $\Sigma = \langle b_0, \dots, b_{L-1} \rangle$ *of distinct write requests. Let* oStep *denote a correct output schedule for Σ that uses T output steps. Then we get a correct prefetch schedule* iStep *for* $\Sigma^R = \langle b_{L-1}, \dots, b_0 \rangle$

that uses T fetch steps by setting $\text{iStep}(b_i) = T - \text{oStep}(b_i) + 1$.

Vice versa, every correct prefetch schedule iStep *for* Σ^R *that uses T fetch steps yields a correct output schedule* $\text{oStep}(b_i) = T - \text{iStep}(b_i) + 1$ *for* Σ*, using T output steps.*

Proof. For the first part, consider the schedule $\text{iStep}(b_i) = T - \text{oStep}(b_i) + 1$. The resulting fetch steps are between 1 and T and all blocks on the same disk get different fetch steps. It remains to be shown that $\text{i}\mathcal{B}\text{acklog}(b_i) < m$ for $0 \le i < L$. With respect to Σ^R, we have

$$\begin{aligned}\text{i}\mathcal{B}\text{acklog}(b_i) &= |\{j > i : \text{iStep}(b_j) \le \text{iStep}(b_i)\}| \\ &= |\{j > i : T - \text{oStep}(b_j) + 1 \le T - \text{oStep}(b_i) + 1\}| \\ &= |\{j > i : \text{oStep}(b_j) \ge \text{oStep}(b_i)\}|\,.\end{aligned}$$

The latter value is $\text{o}\mathcal{B}\text{acklog}(b_i)$ with respect to Σ. It is smaller than m because oStep is a correct schedule.

The proof for the converse case is completely analogous. □

3. Optimal write-once and read-once scheduling.

We now give an optimal algorithm for writing a write-once sequence, prove its optimality, and then apply the duality principle to transform it into a read-once prefetching algorithm.

Consider the algorithm *greedyWriting* for writing a sequence $\Sigma = \langle b_0, \ldots, b_{L-1}\rangle$ of distinct blocks. Let Q denote the set of blocks in the buffer pool so, initially, $Q = \emptyset$. Let $Q_d = \{b \in Q : \text{disk}(b) = d\}$ denote the blocks queued for disk d. Write the blocks b_i in sequence as follows:

1. If $|Q| < m$, then simply insert b_i into Q.
2. Otherwise, each disk d with $Q_d \neq \emptyset$ outputs the block of Q_d that appears first in Σ. The blocks so output are then removed from Q and b_i is inserted into Q.
3. Once all blocks are written, the queues are flushed; that is, additional output steps are performed until Q is empty.

Figure 3.1 gives an example.

A schedule is called a *FIFO schedule* if blocks are output in arrival order on each disk. For the write-once case, the following lemma tells us that when we look for optimal schedules, it is sufficient to consider FIFO schedules. On real disks, FIFO schedules are not necessarily optimal, since they do not optimize seek times and rotational delays, but in our synchronous model, using FIFO suffices and simplifies subsequent proofs.

LEMMA 3.1. *For any sequence of blocks Σ and every correct output schedule* oStep *there is a FIFO output schedule* oStep$'$ *consisting of at most the same number of output steps.*

Proof. The proof of the lemma is based on transforming a non-FIFO schedule into a FIFO schedule by exchanging blocks in the schedule of a disk that are output out of order. Consider a non-FIFO schedule that services two block requests b_i and b_j for the same disk "out of order"; that is, we have $i < j$ but $\text{oStep}(b_i) > \text{oStep}(b_j)$. If we swap the output order of b_i and b_j, then the buffer pool consumption between output steps $\text{oStep}(b_j)$ and $\text{oStep}(b_i)$ can only decrease or remain the same. Such swapping operations can be repeated as necessary until a FIFO schedule is obtained. □

Algorithm greedyWriting is one way to compute a FIFO schedule. The following lemma shows that greedyWriting outputs every block as early as possible.

LEMMA 3.2. *For any sequence of blocks* Σ *and any FIFO output schedule* oStep' *for* $|\Sigma|$*, let* oStep *denote the schedule produced by algorithm greedyWriting. Then for all* $b_i \in \Sigma$*, we have* $\mathrm{oStep}(b_i) \le \mathrm{oStep}'(b_i)$.

Proof. The proof is by induction on $|\Sigma|$.

Base case for induction. $|\Sigma| = 0$. The claim is vacuously true for $|\Sigma| = 0$.

Induction hypothesis. Assume that the claim is true for $|\Sigma| = L - 1$.

Induction step $|\Sigma| = L-1 \rightsquigarrow |\Sigma| = L$. Consider the state of *greedyWriting* when the last block b_L is scheduled. Let $d = \mathrm{disk}(b_L)$. Using the induction hypothesis, it suffices to prove that $\mathrm{oStep}(b_L) \le \mathrm{oStep}'(b_L)$.

Case 1. $\mathrm{oStep}(b_L) = 1$. This case is trivial since 1 is the smallest possible output step.

Case 2. $\mathrm{oStep}(b_L) = \mathrm{oStep}(b_l) + 1$ for some other block b_l with $\mathrm{disk}(b_l) = d$. Applying the induction hypothesis to $|\Sigma| = \langle b_1, \ldots, b_{L-1} \rangle$, we have $\mathrm{oStep}(b_l) \le \mathrm{oStep}'(b_l)$. By the definition of FIFO output schedules, we have $\mathrm{oStep}'(b_l) < \mathrm{oStep}'(b_L)$; that is, $\mathrm{oStep}'(b_l) + 1 \le \mathrm{oStep}'(b_L)$. All in all, we get

$$\mathrm{oStep}(b_L) = \mathrm{oStep}(b_l) + 1 \le \mathrm{oStep}'(b_l) + 1 \le \mathrm{oStep}'(b_L).$$

All remaining cases. Let $t = \mathrm{oStep}(b_L) > 1$. Disk d is idle during step $t - 1$, and we have to explain why this is unavoidable. Let $C = \{b_l : \mathrm{oStep}(b_l) \ge t - 1\}$ denote the set of blocks that are queued during step $t - 1$. We must have $|C| \ge m$, since otherwise greedyWriting would have queued and output b_L already during step $t - 1$ or earlier. Assume that $\mathrm{oStep}(b_L) > \mathrm{oStep}'(b_L)$; that is, $\mathrm{oStep}(b_L) - 1 \ge \mathrm{oStep}'(b_L)$. By the induction hypothesis, $\mathrm{oStep}'(b_l) \ge \mathrm{oStep}(b_l)$ for all $b_l \in C$. In other words,

$$\mathrm{oStep}'(b_l) \ge \mathrm{oStep}(b_l) \ge \mathrm{oStep}(b_L) - 1 \ge \mathrm{oStep}'(b_L).$$

Hence, if the schedule defined by oStep' is used, all blocks in C are written no earlier than b_L, which requires that more than m blocks have to be buffered at the same time, contradicting the assumption that oStep' is correct. □

Combining Lemmas 3.1 and 3.2 we see that greedyWriting gives us optimal schedules for write-once sequences.

THEOREM 3.3. *Algorithm greedyWriting gives a correct, minimum length output schedule for any write-once reference sequence* Σ.

Combining the duality principle and the optimality of greedyWriting, we get an optimal algorithm for read-once prefetching that we call *lazy prefetching.*

COROLLARY 3.4. *An optimal prefetch schedule* iStep *for a sequence* Σ *can be obtained by using greedyWriting to get an output schedule* oStep *for* Σ^R *and setting* $\mathrm{iStep}(b_i) = T - \mathrm{oStep}(b_i) + 1$.

The schedule can be computed in time $\mathcal{O}(L + D)$ using very simple data structures. Figure 3.1 (top) gives an example. We refer to this approach as *lazy prefetching.*

4. Prudent prefetching.

Although the lazy prefetching approach in the previous section allows us to obtain a prefetch schedule with a minimal number of steps by means of reversing time, it has the practical disadvantage that blocks are accessed as late as possible even if most blocks could be fetched earlier. For example, in Figure 3.1 (top) only the bottom-most disk fetches a block in Step 1. This policy may result in unnecessary delays in real implementations where the access times to the blocks fluctuate. Many of these delays might be avoidable if some blocks were fetched earlier. One might instead use "eager prefetching" [6, 15], i.e., always accessing the

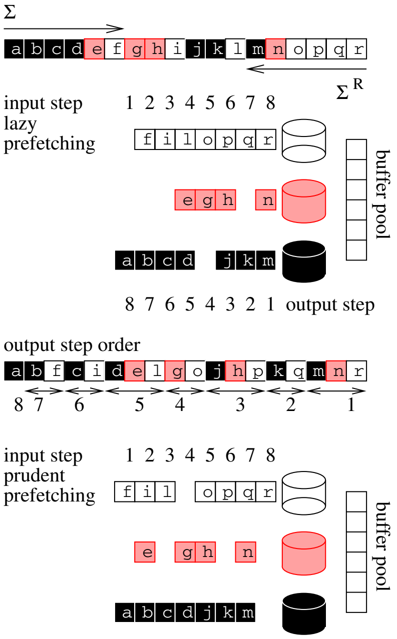

FIG. 3.1. *Duality between prefetching and output for a sequence* $\Sigma = \langle \mathtt{a}, \mathtt{b}, \dots, \mathtt{r}\rangle$ *of* $L = 18$ *blocks to be read using* $D = 3$ *disks and* $m = 6$ *buffers. Top part: A lazy prefetch schedule for* Σ *as the dual of an output schedule for* Σ^R*. The shading of the blocks indicates the disks where the blocks are located. Before output Step* 2*, the eight blocks* `hijklopq` *would have to be buffered in order to output block* `h` *in Step* 2*. But since only six blocks can be buffered, the middle disk has to remain idle in Step* 2*. Similarly, before output Step* 4*, the seven blocks* `defgilo` *would have to be buffered to output block* `d`*. Bottom part: The resulting schedule for prudent prefetching. For example, before Step* 3*, blocks* `a` *and* `b` *are fetched and buffers for blocks* `fcidel` *are reserved. Block* `g` *cannot be fetched because no buffer is reserved for it. Prudent prefetching using the reading order as a priority instead of the priorities based on the optimal lazy schedule would need one more I/O step.*

highest priority block on each disk. But eager prefetching sometimes has to discard and refetch blocks, causing complications and inefficiencies.

Here we propose *prudent prefetching*, a prefetching strategy that avoids both problems. It maintains optimal schedule length, but attempts to fetch blocks as early as possible. The idea is to use the oStep obtained by greedyWriting as a priority rather than as a direct indication of the input step for fetching a block. Algorithm prudent prefetching allows blocks to be fetched before blocks with higher priority are fetched but only if buffers have been reserved for them. This way, otherwise idle disks can prefetch low priority blocks without hindering any fetches of higher priority blocks in later steps.

This strategy is easy to implement. Prudent prefetching works with a sequence $\langle l_0, \dots, l_{L-1}\rangle$ of block requests sorted by nonincreasing priority (and hence by nondecreasing iStep of lazy prefetching). Blocks $l_0, \dots, l_{j-1}$ have either been fetched or have a reserved buffer while blocks $l_j, \dots, l_{L-1}$ are neither fetched nor have a reserved buffer.

Before each fetch step, all empty buffers are reserved for the next blocks in the priority sequence, and j is advanced by the number of buffers so reserved. The highest priority block from each disk is then fetched if a buffer has been reserved for it. Then from each disk the highest priority block is fetched if a buffer has been reserved for it.

As before, blocks are delivered to the application in the order prescribed by Σ. When a block is delivered to the application, its buffer becomes empty and is available.

An example of the algorithm is shown in Figure 3.1. We cannot expect algorithm *prudent prefetching* to be better than lazy prefetching as long as we only count I/O steps, but we can show that it is not worse.

THEOREM 4.1. *For any correct output schedule* oStep*,* prudent prefetching *takes no more I/O steps than* lazy prefetching*.*

Proof. We have already observed that fetching a reserved block can never hinder a higher priority block from being fetched. Hence, in the ith step, all unfetched blocks with iStep i will be fetched. We omit a trivial, more detailed proof by induction over the number of steps. □

Another advantage of *prudent prefetching* is that it can be implemented in an event driven manner, and the fetch steps for each disk need not be synchronized. When the next block from Σ is delivered to the application, its buffer can immediately be used for advancing j. When a disk finishes fetching a block, it waits (if necessary) until the next highest priority block on this disk has a reserved buffer and then starts to fetch this block. Thus there is never a need to synchronize the disks, the system can adapt to variances in access times, and the load of the interconnections between disks and processors is better balanced than for synchronous access.

5. How good is optimal?

When we have complex data access patterns, the knowledge that we have an optimal prefetching algorithm is often of little help. We also want to know "how good is optimal?". In the worst case, all requests may go to the same disk and no prefetching algorithm can cure the dreadful performance caused by this bottleneck. However, the situation is different if an appropriate block allocation strategy is used; for example, if blocks are allocated to disks using striping, randomization,[2] or both.

THEOREM 5.1. *Consider a sequence of L block requests, and a buffer pool of size $m \geq D$ blocks. The number of I/O steps needed by greedyWriting or lazy prefetching is given by the following bounds, depending on the block allocation strategy. For striping and randomized cycling, an arbitrary interleaving of sequential accesses to S sequences is allowed:*

$$\begin{aligned}
&S\text{:}\left\lfloor \frac{L}{D} \right\rfloor + S \quad \textit{if } m > S(D-1);\\
&FR\text{:}\left(1+\mathcal{O}\left(\frac{D}{m}\right)\right)\frac{L}{D}+\mathcal{O}\left(\frac{m}{D}\log m\right) \quad \textit{(expected)};\\
&RC\text{:}\left(1+\mathcal{O}\left(\frac{D}{m}\right)\right)\frac{L}{D}+\min\left\{S+\frac{L}{D}, \mathcal{O}\left(\frac{m}{D}\log m\right)\right\} \textit{(expected).}
\end{aligned}$$

For the case of writing, the second term can be dropped if we are only interested in the number of steps needed to write (but not necessarily output) all blocks.

Proof. Due to our result on duality, it suffices to prove the bounds for writing.

Striping (S). Since greedyWriting is optimal, it suffices to analyze the following specialized algorithm: Each sequence gets an exclusive allotment of $D-1$ buffer blocks. When a block from sequence k is written there are two possible cases. If the pool for k has a free buffer block, the block is buffered there. Otherwise, we have

[2] In practice, this will be done using simple hash functions. However, for the analysis we assume that we have a perfect source of randomness.

exactly D consecutive blocks from the striped sequence k so that we can output one block from sequence k to each disk. There can be at most $\lfloor L/D \rfloor$ of these output steps. When all blocks are written, one additional output step for each sequence suffices to empty all buffers.

Fully random allocation (FR). Since *greedyWriting* is optimal, it dominates the algorithm analyzed in [22]. This algorithm admits $(1-\epsilon)D$ blocks into the buffer pool before each output step. It is shown that with this regime the buffer pool size remains in $\mathcal{O}(D/\epsilon)$ most of the time. More precisely, the probability that the required pool size exceeds qD is less than $e^{(\ln 2-\epsilon q)D}$ [22, Lemma 3]. By setting $\epsilon = \Theta(D/m)$ we can make sure that the pool size is exceeded so rarely that we could afford to flush the queues whenever this happens. We omit the straightforward calculations with the tail bound showing that after an expected number of $(1+\mathcal{O}(D/m))L/D$ output steps all blocks have been written. The number of output steps needed to flush the buffers at the end is the maximum number of blocks queued at a disk. In [22] it is shown that the probability that the queue length at a particular disk exceeds q is bounded by $2e^{-\epsilon q}$. Setting $q = \ln(2Dm)/\epsilon$ and multiplying by D, we can see that the probability p_{fail} that some disk has a final load of more than $\ln(2Dm)/\epsilon$ is bounded by $D \cdot 2e^{-\epsilon \ln(2Dm)/\epsilon} = 1/m$. Since the load cannot exceed L, the expected maximum load is bounded by $\ln(2Dm)/\epsilon + L/m = \mathcal{O}\big(\frac{m}{D}\log m + L/m\big)$. The term L/m can be absorbed into $\big(1+\mathcal{O}\big(\frac{D}{m}\big)\big)\frac{L}{D}$.

Randomized cycling (RC). Vitter and Hutchinson [24] show that the algorithm from [22] performs at least as well on RC-streams as it does for fully random allocation. This extends to *greedyWriting* by Theorem 3.3. Furthermore, the reverse of a sequence accessing RC-streams is indistinguishable from a sequence accessing RC-streams in the forward direction. Theorem 2.3 therefore extends the result to prefetching. Hence, the bound

$$\left(1+\mathcal{O}\left(\frac{D}{m}\right)\right)\frac{L}{D}+\mathcal{O}\left(\frac{m}{D}\log m\right)$$

transfers from the fully random allocation case.

For small L and m this bound can be improved using the observation that the maximum number of blocks queued at a disk at the end cannot exceed the total number of blocks allocated to it. The bound for striping shows that this load cannot exceed $L/D + S$. □

6. Prefetching with caching.

We now relax the condition that the read requests in Σ are for distinct blocks, permitting the possibility of saving disk accesses by keeping previously accessed blocks in memory. For this *read-many* problem, we get a tradeoff for the use of the buffer pool because it has to serve the double purposes of keeping blocks that are accessed multiple times, and decoupling physical and logical accesses to equalize transient load imbalance of the disks. We define the *write-many* problem in such a way that the duality principle from Theorem 2.3 transfers: *The latest instance of each block must be kept either on its assigned disk, or in the buffer pool. The final instance of each block must be output to its assigned disk.*[3]

We prove that the following offline algorithm *manyWriting* minimizes the number of output operations for the write-many problem: Let Q denote the set of blocks in

[3] The requirement that the latest versions have to be kept might seem odd in an offline setting. However, this makes sense if there is a possibility that there are reads at unknown times that need an up-to-date version of a block.

the buffer pool, so initially $Q = \emptyset$. Let $Q_d = \{b \in Q : \text{disk}(b) = d\}$ denote the blocks queued for disk d. To write block b_i, if $b_i \in Q$, the old version is overwritten in its existing buffer. Otherwise, if $|Q| < m$, b_i is inserted into Q. If this also fails, an output step is performed before b_i is inserted into Q. The output analogue of Belady's MIN rule [7] is used on each disk; that is, each disk with $Q_d \neq \emptyset$ outputs the block in Q_d that is written again farthest in the future.

THEOREM 6.1. *Algorithm manyWriting solves the* write-many problem *with the fewest number of output steps.*

Applying duality, we also get an optimal algorithm for prefetching plus caching of a sequence Σ; using the same construction as in Corollary 3.4 we get an optimal prefetching and caching schedule.

COROLLARY 6.2. *The dual of manyWriting solves the* read-many problem *with the fewest number of input steps.*

It remains to prove Theorem 6.1.

Proof of Theorem 6.1. We generalize the proof of Belady's algorithm by Borodin and El-Yaniv [8] to the case of writing and multiple disks. Let $\Sigma = \langle b_0, \ldots, b_{L-1}\rangle$ be any sequence of blocks to be written. The proof is based on the following claim.

Claim. Let ALG be any algorithm for the write-many problem. Let d denote a fixed disk. For any $0 \leq i < L$ it is possible to construct an offline algorithm ALG_i that satisfies the following properties:

(i) ALG_i processes the first $i - 1$ writes exactly as ALG does.

(ii) If block b_i is the first block written after output step s, then immediately before output step s there was no free buffer slot.

(iii) If b_i is the first block written after output step s, then ALG_i performs this output according to the MIN rule on disk d.

(iv) ALG_i takes no more steps than ALG.

Once this claim is established, the theorem can be proven as follows: Starting with an optimal offline algorithm OPT, we apply the claim with $i = 0$ and $d = 0$ to obtain another optimal algorithm OPT_0, then apply the claim with $i = 1$ and $d = 0$ to obtain OPT_1 and so on. By induction over i, it can be seen that OPT_{L-1} never leaves unused buffer slots before an output step and uses MIN for deciding which blocks to output on disk 0. Subsequently, we apply this sequence of L transformations for each disk. Since these transformations do not undo property (iii) for other disks, we arrive at an optimal algorithm that works like manyWriting on all disks.

It remains to prove the claim. We initialize ALG_i to ALG and transform ALG_i until it fulfills all four properties. Note that this initialization automatically fulfills properties (i) and (iv). If properties (ii) and (iii) also hold, we are done.

If property (ii) is violated by ALG_i, then b_i is the first block written by ALG_i *after* some output step s, and before output step s a free buffer slot was available. In this case, ALG_i is modified so that b_i is now the last block written *before* output step s. Note that this transformation preserves the order in which blocks are written and properties (i) and (iv). This transformation is repeated until ALG_i also satisfies property (ii).

If properties (i), (ii), and (iv) hold but property (iii) is violated, there must be a write step s so that b_{i-1} is the last block written before output step s, and b_i is the first block written after output step s in ALG_i. Now we define a modified algorithm ALG'_i that mimics ALG_i (and hence ALG) until b_{i-1} is written but uses the MIN rule in step s so that properties (i)–(iii) hold for ALG'_i.

It remains to define the behavior of ALG'_i after step s so that property (iv) is also maintained. We use $\mathcal{X} + b$ as a shorthand for $\mathcal{X} + \{b\}$ for a set of blocks $\mathcal{X}$ and a block b. Immediately after output step s, the buffer pool of ALG_i can be written as $\mathcal{M} = \mathcal{X} + b$ whereas ALG'_i has buffer pool $\mathcal{M}' = \mathcal{X} + b'$ where b is the block on disk d whose next access is farthest in the future. ALG'_i mimics ALG_i as far as possible; that is, it performs output steps at the same time as ALG_i and outputs the same blocks. As long as neither b nor b' is written or output by ALG_i, these two blocks remain the only difference between $\mathcal{M}$ and $\mathcal{M}'$. There are only three types of events that require special treatment.

Event 1. ALG_i outputs b. In that case, ALG'_i outputs b'. Afterwards we have $\mathcal{M} = \mathcal{M}'$, and from now on ALG'_i can completely mimic ALG_i.

Event 2. Block b' is rewritten. By definition of b, this situation happens before block b is rewritten. After b' is rewritten, we have $\mathcal{M} = \mathcal{Y} + b + b'$ and $\mathcal{M}' = \mathcal{Y} + b'$ for some common set $\mathcal{Y}$ of blocks. In particular, $\mathcal{M}'$ has one unused buffer slot. If ALG_i outputs b in this situation, ALG'_i can again unify the states $\mathcal{M}$ and $\mathcal{M}'$ by *not* outputting anything on disk d.

Event 3. Block b is rewritten. As discussed above, if $\mathcal{M} \neq \mathcal{M}'$, we must have $\mathcal{M} = \mathcal{Y} + b + b'$ and $\mathcal{M}' = \mathcal{Y} + b'$ before b is rewritten. Now ALG'_i uses its free buffer slot to accommodate b. We get $\mathcal{M} = \mathcal{M}' = \mathcal{Y} + b + b'$.

We end up with an algorithm ALG'_i that fulfills properties (i)–(iv) and hence set ALG_i=ALG'_i. □

7. Application to sorting.

In this section we extend the duality between prefetching and queued writing to apply to problems of merging and distribution. In the merging phase of mergesort, there are several sorted runs on the disk, and the problem is to merge them together into a single sorted run. We assume that each run is striped across the disks using any given striping discipline, such as RC or FR, as described in the introduction. How to lay out the runs so as to permit fully parallel I/O is a challenging problem; recent work is surveyed in [23].

A big problem is that the input order Σ for the blocks, namely the order in which the blocks need to be accessed, is highly data-dependent. The key to duality is to characterize Σ in a simple and easily implementable way. If we examine the process of merging, as illustrated in Figure 7.1 from the bottom to top, we see that the merging buffer contains a partially filled block from each run (not yet expired). When the block empties all its items into the merged output stream, the next block from that run is inserted into the merging buffer. The merging buffer is pictured in the upper rectangle in Figure 7.1, which is distinct from the space reserved for the prefetch buffers (lower rectangle in Figure 7.1).

The first moment, therefore, that a block absolutely must be in memory is when the smallest key value of the block is merged into the output stream. We therefore define the *trigger* of a block to be its smallest key value. We say that a block is read when it is moved from the prefetch buffer to the merging buffer, where it stays until its items are exhausted by the merging process. Thus, the read order Σ of the blocks is given by the sorted order of the triggers.

We have now reduced the merging problem to that of prefetching for the input sequence Σ. The dual problem to merging is distribution. To solve it via the duality principle, we need to process Σ in reverse order. We equate the notion of bucket in distribution with that of run-in merging, so each block therefore has a bucket assigned to it. Since each bucket uses a fixed striping discipline, the blocks can then be assigned to disks. The dual distribution problem is thus well defined, and we get an optimal algorithm for merging by applying the algorithm of section 3.

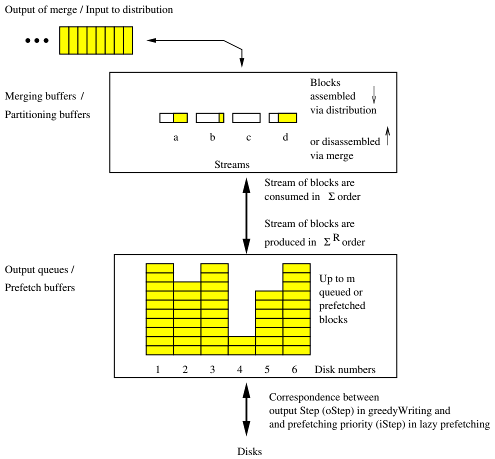
Output of merge / Input to distribution
Merging buffers / Partitioning buffers
Blocks assembled via distribution
or disassembled via merge
a
b
c
d
Streams
Stream of blocks are consumed in Σ order
Stream of blocks are produced in Σ^R order
Output queues / Prefetch buffers
Up to m queued or prefetched blocks
1
2
3
4
5
6
Disk numbers
Correspondence between output Step (oStep) in greedyWriting and and prefetching priority (iStep) in lazy prefetching
Disks

FIG. 7.1. *The relationship between merging and distribution. Buffer space is required both "privately" within the application (for storing the lead block of each run in merging, and for storing the next block being formed for each bucket in distribution), and for the Output queues / Prefetch buffers required for the techniques proposed in this paper. During distribution, the priorities of blocks correspond to their output step. For merging, blocks are read in the order given by the triggers. When an appropriate allocation discipline is used to allocate blocks of a stream to the parallel disks, the queued I/O techniques of this paper permit I/O complexity results for distribution sort to be applied to mergesort (and vice versa if desired).*

Mergesort with randomized cycling (RCM). How the blocks of each run are striped depends on the particular allocation discipline used. We start by discussing multiway mergesort using randomized cycling allocation (RCM) in some detail and then survey a number of additional results. Originally, the N input elements are stored as a single data stream using any kind of striping. During *run formation* the input is read in chunks of size M that are sorted internally and then written out in runs allocated using RC allocation. Neglecting trivial rounding issues, run formation is easy to do using $2N/(DB)$ I/O steps. For example, we need $\mathcal{O}\big(N/(DB^2)\big)$ additional I/Os for writing the trigger values to separate files. Then we set aside a buffer pool of size $m = D/\gamma$ for some parameter γ and perform $\lceil \log_{M/B-\mathcal{O}(D/\gamma)} \frac{N}{M} \rceil$ *merge phases.* In a merge phase, groups of $k = \frac{M}{B} - \mathcal{O}(D/\gamma)$ runs are merged into new sorted runs; that is, after the last merge phase, only one sorted run is left. Merging k runs of total size sB can be performed using s block reads by keeping one block of each run in the internal memory of the sorting application. The I/O schedule for a merging phase is found by sorting the triggers for groups of k runs each. These sorted trigger sequences are then concatenated, yielding the order in which the blocks are to be read. At this point we can apply duality.

The overhead for this precomputation of Σ (trigger values) is $\mathcal{O}\big(N/B^2\big)$ I/Os even for a single disk [6]. The triggers allow us to do optimal prefetching so that Theorem 5.1 gives an upper bound of

$$(1 + \mathcal{O}(\gamma))\,\frac{N}{BD} + \min\left\{k + \frac{N}{BD}, \mathcal{O}(\log(D/\gamma)/\gamma)\right\}$$

for the expected number of fetch steps of a phase. The number of output steps for a phase is $N/(BD)$ if we have an additional output buffer of D blocks. The final result

TABLE 7.1

Summary of the I/O complexity for parallel disk sorting algorithms. Each algorithm's I/O complexity is given by $\mathrm{Sort}_{\Delta}^{a,f}(N)$ *when the parameters are set according to the algorithm's entry in the table. Algorithms with* **boldface** *names are asymptotically optimal: M = Mergesort. SM/SD = Merge / Distribution sort with any S allocation. SRM and SRD use SR. RCD, RCD+, and RCM use RC allocation.*

$\mathrm{Sort}_{\Delta}^{a,f}(N)$ I/Os			Algorithm	Source
a	f	Δ		
2	0	0	Lower bound	
Deterministic algorithms				
2	0	0	M, $D=1$	[1]
$\mathcal{O}(1)$	0	0	**Greed sort**	[21]
$2+\gamma$	0	$\Theta\left((2D)^{1+\frac{2}{\gamma}}\right)$	M, superblock striping	
$2+\gamma$	0	$\Theta\left((2D)^{1+\frac{2}{\gamma}}\right)$	SM	here
$2+\gamma$	0	$\Theta\left((2D)^{1+\frac{2}{\gamma}}\right)$	SD	here
Randomized algorithms				
$2+\gamma$	0	$\Theta(D\log(D)/\gamma^2)$	SRM	[6]
$2+\gamma$	0	$\Theta(D\log(D)/\gamma^2)$	SRD	here
$3+\gamma$	0	$\Theta(D/\gamma)$	**RCD**	[24]
$2+\gamma$	$\min(\frac{N}{BD},\log(D)/\mathcal{O}(\gamma))$	$\Theta(D/\gamma)$	**RCM**	here
$2+\gamma$	0	$\Theta(D/\gamma)$	**RCD+**	here

is written using any striped allocation strategy; the application calling the sorting routine need not be able to handle RC allocation. For any constant $\gamma > 0$, we can write the resulting total number of I/O steps as $\mathrm{Sort}_{m+D}^{2+\mathcal{O}(\gamma),\min(\frac{N}{BD},\frac{\log D}{\mathcal{O}(\gamma)})}(N)$, where

$$\mathrm{Sort}_{\Delta}^{a,f}(N) = \frac{2N}{DB} + a\cdot\frac{N}{DB}\cdot\left\lceil \log_{\frac{M}{B}-\Delta}\frac{N}{M}\right\rceil + f + o\left(\frac{N}{DB}\right).$$

Table 7.1 compares a selection of sorting algorithms using this generalized form of the I/O bound for parallel disk sorting. The term $\frac{2N}{DB}$ represents the reading and writing of the input and the final output, respectively. The factor a is decisive for the I/O complexity for large inputs. Note that for $a = 2$ and $f = \Delta = 0$ this expression is the lower bound for sorting. The additive offset f may dominate for small inputs. The reduction of the memory by Δ blocks in the base of the logarithm is due to memory that is used for output or prefetching buffer pools outside the merging or distribution routines, and hence reduces the number of data streams that can be handled concurrently. One way to interpret Δ is to view it as the amount of *additional* memory needed to match the performance of the algorithm on the multihead I/O model [1] where load balancing disk accesses is not an issue.

Note that the RCM algorithm outlined above is the first asymptotically optimal parallel disk sorting algorithm that approaches the optimal constant factor 2 for $M/B \gg D$. The first two rows of Table 7.1 show that single disk sorting (e.g., multiway mergesort) is optimal. Greed sort [21] is an optimal (up to a constant factor) deterministic sorting method based on mergesort; it does it an approximate merge and then finalizes the merge using columnsort. Balance sort [20] is an equally optimal but more practical deterministic sorting algorithm that uses distribution sorting together with adaptive allocation of blocks. An algorithm frequently used in practice is a single disk algorithm together with superblock striping (i.e., using logical blocks of size BD). This algorithm is quite good if the input is sufficiently small that we can still sort in two passes despite the much larger block size. Using Theorem 5.1, we get the same asymptotic bounds if we use the parallel disk mergesort outlined above together with any deterministic striping discipline (SM); that is, even as a deterministic algorithm, our algorithm has performance comparable to algorithms used in practice.

Mergesort using SRM was analyzed in [6]. Although our optimal prefetching result simplifies and improves the prefetching algorithms given there, we do not get improved asymptotic bounds.

Distribution sort with randomized cycling (RCD+). Using random sampling and the duality between reading and writing, as shown in Figure 7.1, we can transfer the results for mergesort to results using distribution-based sorting algorithms. We obtain a new distribution sort using deterministic striping (RD) or simple randomized striping (SRD). We can also improve the analysis of the variant with RC [24] reducing the constant factor from three to two. Furthermore, an additional optimization can be used to remove the additive term $\min(\frac{N}{BD}, \log(D)/\mathcal{O}(\gamma))$ in the complexity of RCM. Below we describe the resulting algorithm RCD+ in more detail since it currently represents the parallel disk sorting algorithm with the best known bounds. The same algorithm underlies the results for the other allocation strategies SD and SRD.

The basic idea behind distribution sort is to use a generalization of quicksort where elements are classified into $k = \mathcal{O}(M/B)$ classes based on $k-1$ splitter elements. The splitters are chosen in such a way that each class has size $\mathcal{O}(N/k)$. These classes are then sorted recursively and the results are appended to form the final output.

As in mergesort, we start with an input that is striped over the disks using some arbitrary allocation strategy. We set $k = \min(N/(M - cBD), M/B - cBD)$ for an appropriate parameter c. To find the splitter elements we take $dk - 1$ random sample elements for an appropriate integer d. The sample is sorted and every dth element in the sorted sample is used as a splitter. Standard calculations using Chernoff bounds indicate that $d = \mathcal{O}(\log k)$ is sufficient to ensure that with high probability at most $\mathcal{O}(N/k)$ elements lie between two splitters. It can be seen that the number of I/Os needed for obtaining the sample is only a lower-order term compared to the number of I/Os needed to scan the input.

Now the input is classified into k classes by scanning the input and putting each element in the appropriate class. For each class we use an output stream allocated using RC. For each class, an output buffer block is maintained that is written to an RC allocated output stream when the buffer is completely filled. Writing uses greedyWriting. Here it is useful that the algorithm is an online algorithm since it is not known in advance in what order blocks have to be written out.

The additive term in the I/O bound for RCM mergesort can be avoided in RCD+ using the simple observation that the write buffers need not be flushed—blocks that are logically written but still in the output queue when the distribution finishes, are not flushed to disk but kept in the queues; see also the last sentence in Theorem 5.1. When we read a block in the subsequent recursive sorting phases, we therefore have to check whether this block is still in the output queue and should be taken from there.

Recursive sorting of the classes proceeds depth first, from left to right. As soon as a class fits into internal memory, it is loaded and sorted internally, then it is output using any kind of striping. No randomization is needed for the final output because there is only a single data stream. It suffices to keep D output buffer blocks for the final output. Since the output is generated in sorted order, these output buffers need not be flushed when we are finished with a class which would lead to load imbalance for writing and partially filled blocks. Rather these buffer blocks are kept until they are filled by the sorting of subsequent classes. This way the output is produced in a perfectly striped fashion without partially filled blocks.

8. A tight lower bound for external sorting.

Our main result for parallel disk sorting is that we close the gap between the upper and lower bounds up to lower-order terms. However, the lower bound from [1] leaves open the constant factors. In particular, it is not clear there what happens if the number of output steps and input steps differ. Therefore we now strengthen the lower bound to obtain the right constant factor.

THEOREM 8.1. *Assuming that M/B is an increasing function, the number of I/Os required to sort or permute n items, up to lower-order terms, is at least*

$$\frac{2N}{D}\frac{\log(N/B)}{B\log(M/B)+2\log N} \sim \begin{cases} \dfrac{2N}{DB}\dfrac{\log(N/B)}{\log(M/B)} & \text{if } B\log\dfrac{M}{B} = \omega(\log N), \\ \dfrac{N}{D} & \text{if } B\log\dfrac{M}{B} = o(\log N). \end{cases}$$

The second case in the theorem is the pathological case in which the block size B and internal memory size M are so small that the optimal way to permute the items is to move them one at a time in the naive manner, not making use of blocking.

The rest of this section is devoted to a proof of Theorem 8.1.

For the lower bound calculation, we can assume without loss of generality that there is only one disk, namely, $D = 1$. The I/O lower bound for general D follows by dividing the lower bound for one disk by a factor of D.

DEFINITION 8.2. *We call an input operation* simple *if each item that is transferred from the disk gets removed from the disk and deposited into an empty location in internal memory; similarly, an output is* simple *if the transferred items are removed from internal memory and deposited into empty locations on disk.*

LEMMA 8.3 (Aggarwal and Vitter [1]). *For each computation that implements a permutation of the N items, there is a corresponding computation strategy involving only simple I/Os such that the total number of I/Os is no greater.*

Proof. It is easy to construct the simple computation strategy by working backwards. We cancel the transfer of an item if its transfer is not needed for the final result. The resulting I/O strategy is simple. □

For the lower bound, we use the approach of Aggarwal and Vitter [1] and bound the maximum number of permutations that can be produced by at most t I/Os. If we take the value of t for which the bound first reaches $N!$, we get a lower bound on the worst-case number of I/Os. We can get a lower bound on the average case in a similar way.

DEFINITION 8.4. *We say that a permutation $p_1, p_2, \ldots, p_N$ of the N items can be* produced after t_I input operations and t_O output operations *if there is some intermixed sequence of t_I input operations and t_O output operations so that the items end up in the permuted order $p_1, p_2, \ldots, p_N$ in extended memory. (By extended memory we mean the memory locations of internal memory followed by the memory locations on disk in sequential order.) The items do not have to be in contiguous positions in internal memory or on disk; there can be arbitrarily many empty locations between adjacent items.*

As mentioned above, we assume that I/Os are simple. Each I/O transfers exactly B items, although some of the items may be *nil*. In addition, the I/Os obey block boundaries, in that all the non-*nil* items in a given I/O come from or go to the same block on disk.

Initially, the number of producible permutations is 1. Let us consider the effect of an output. There can be at most $N/B+o-1$ nonempty blocks before the oth output operation, and thus the items in the oth output can go into one of $N/B+o$ places relative to the other blocks. Hence, the oth output boosts the number of producible permutations by a factor of at most $N/B+o$, which can be bounded trivially by

$$N(1+\log N). \tag{8.1}$$

For the case of an input operation, we first consider a read I/O from a specific block on disk. If the b items involved in the read I/O were together in internal memory at some previous time (e.g., if the block was created by an earlier output operation), then the items could have been arranged in an arbitrary order by the algorithm while in internal memory. Thus, the $b!$ possible ordering of the b inputted items relative to themselves could already have been produced before the input operation. This implies in a subtle way that rearranging the newly inputted items among the other $M-b$ items in internal memory can boost the number of producible permutations by a factor of at most $\binom{M}{b}$, which is the number of ways to intersperse b indistinguishable items within a group of size M.

The above analysis applies to input from a specific block. If the input was preceded by a total of o output operations, there are at most $N/B+o \le N(1+\log N)$ blocks to choose from for the I/O, so the number of producible permutations is boosted further by at most $N(1+\log N)$. Therefore, assuming that at some point the b inputted items were previously together in internal memory, an input operation can boost the number of producible permutations by at most

$$N(1+\log N)\binom{M}{b}. \tag{8.2}$$

Now let us consider an input operation in which some of the inputted items were not together previously in internal memory (e.g., the first time a block is read). By rearranging the relative order of the items in internal memory, we can increase the number of producible permutations. Given that there are N/B full blocks initially, we get the maximum increase when the N/B blocks are read in full, which boosts the number of producible permutations by a factor of

$$(B!)^{N/B}. \tag{8.3}$$

Let I be the total number of input operations. In the ith input operation, let b_i be the number of items brought into internal memory. By the simplicity property, some of the items in the block being accessed may not be brought into internal memory, but rather may be left on disk. In this case, b_i counts only the number of items that are removed from disk and left in internal memory. In particular, we have $0 \le b_i \le B$.

By the simplicity property, we need to make room in internal memory for the new items arriving, and in the end all items are stored back on disk. Therefore we get the following lower bound on the number O of output operations:

$$O \ge \frac{1}{B}\left(\sum_{1\le i\le I} b_i\right). \tag{8.4}$$

Combining (8.1), (8.2), and (8.3), we find that

$$\big(N(1+\log N)\big)^{I+O}\prod_{1\le i\le I}\binom{M}{b_i}\ge\frac{N!}{(B!)^{N/B}},\tag{8.5}$$

where O satisfies (8.4).

Let $\bar{B}$ be the average number of items read during the I input operations. By a convexity argument, the left-hand side of (8.5) is maximized when each b_i has the same value, namely, $\bar{B}$. From (8.5) and (8.4), we get

$$\big(N(1+\log N)\big)^{I+O}\binom{M}{\bar{B}}^{I}\ge\frac{N!}{(B!)^{N/B}},\tag{8.6}$$

$$\big(N(1+\log N)\big)^{I+O}\binom{M}{\bar{B}}^{(I+O)/(1+\bar{B}/B)}\ge\frac{N!}{(B!)^{N/B}}.\tag{8.7}$$

The left-hand side of (8.7) is maximized when $\bar{B}=B$, so we get

$$\big(N(1+\log N)\big)^{I+O}\binom{M}{B}^{(I+O)/2}\ge\frac{N!}{(B!)^{N/B}}.\tag{8.8}$$

The theorem follows by taking logarithms of both sides of (8.8) and using Stirling's formula and the fact that M/B is an increasing function.

9. Conclusions. In this paper we have exploited a natural duality between prefetching (read problem) and outputting (write problem). We have shown that an optimal schedule for one problem is the reverse of an optimal schedule for the other. We have generalized our approach to the read-many case in which frequently accessed blocks can be cached in memory. We have further reduced the problem of mergesorting and distribution sorting to the read and write problems, and by duality we have given practical yet asymptotically optimal (up to lower-order terms) em algorithms for mergesort and distribution sort. The algorithms are practical [12] and have very low overheads, thus making them desirable in practice.

Appendix. Summary of notation.

B: Block size.
b_i: The ith block in a sequence of blocks.
D: Number of disks. In an acronym it stands for a sorting algorithm based on data Distribution.
d: Index of some disk.
disk(b_i): The disk where block b_i is allocated.
FR: Fully random allocation.
iStep(b_i): The input step when block b_i is fetched.
i$\mathcal{B}$acklog(b_i): $|\{j > i : \text{iStep}(b_j) \le \text{iStep}(b_i)\}|$.
L: Number of blocks in the access sequence Σ.
M: Size of the fast internal memory. In an acronym it stands for a sorting algorithm based on Merging.
m: Number of buffer blocks in the buffer pool. Note that $m \le M/B$. In the sorting algorithms $m = \Theta(M/B)$.
N: The number of elements to be sorted.
oStep(b_i): The output step when block b_i is fetched.
o$\mathcal{B}$acklog(b_i): $|\{j < i : \text{oStep}(b_j) \ge \text{oStep}(b_i)\}|$.

RC: Randomized Cycling allocation.
π_i: In RC allocation the random permutation used to allocate sequence i.
S: Stands for Striping in an allocation strategy or sorting algorithm.
SR: Simple randomized Striping using a random rotation.
Sort(N): $\frac{N}{DB}(1 + \log_{M/B} \frac{N}{M})$ the I/O complexity of sorting N elements "without the constant factor."
Σ: The sequence of blocks to be read or written.
Σ^R: The reverse of sequence Σ.

Acknowledgments. We would like to thank Jeffrey Chase, Andreas Crauser, S. Mitra, Nitin Rajput, Erhard Rahm, and Berthold Vöcking for valuable discussions. Anonymous referees have given detailed feedback that helped to improve the readability of the paper.

REFERENCES

[1] A. Aggarwal and J. S. Vitter, *The input/output complexity of sorting and related problems*, Comm. ACM, 31 (1988), pp. 1116–1127.

[2] S. Albers, *On the influence of lookahead in competitive paging algorithms*, Algorithmica, 18 (1997), pp. 283–305.

[3] S. Albers and M. Büttner, *Integrated prefetching and caching in single and parallel disk systems*, in Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures, San Diego, 2003, pp. 109–117.

[4] S. Albers, N. Garg, and S. Leonardi, *Minimizing stall time in single and parallel disk systems*, in Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC-98), New York, 1998, ACM Press, pp. 454–462.

[5] R. D. Barve, E. F. Grove, and J. S. Vitter, *Simple randomized mergesort on parallel disks*, Parallel Comput., 23 (1997), pp. 601–631.

[6] R. D. Barve and J. S. Vitter, *A simple and efficient parallel disk mergesort*, in Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures, St. Malo, France, 1999, pp. 232–241.

[7] A. L. Belady, *A study of replacement algorithms for virtual storage computers*, IBM Systems J., 5 (1966), pp. 78–101.

[8] A. Borodin and R. El-Yaniv, *Online Computation and Competitive Analysis*, Cambridge University Press, Cambridge, UK, 1998.

[9] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, *Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling*, ACM Trans. Comput. Systems, 14 (1996), pp. 311–343.

[10] F. Dehne, W. Dittrich, and D. Hutchinson, *Efficient external memory algorithms by simulating coarse-grained parallel algorithms*, in Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures, Newport, RI, 1997, pp. 106–115.

[11] F. Dehne, W. Dittrich, D. Hutchinson, and A. Maheshwari, *Reducing I/O complexity by simulating coarse grained parallel algorithms*, in proceedings of the 13th Annual International Parallel Processing Symposium, IEEE, San Juan, Puerto Rico, 1999, pp. 14–20.

[12] R. Dementiev and P. Sanders, *Asynchronous parallel disk sorting*, in Proceedings of the 15th Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Diego, 2003, pp. 138–148.

[13] D. A. Hutchinson, P. Sanders, and J. S. Vitter, *Duality between prefetching and queued writing with parallel disks*, in Proceedings of the 9th Annual European Symposium on Algorithms (ESA), Lecture Notes in Comput. Sci. 2161, Springer, Berlin, 2001, pp. 62–73.

[14] D. A. Hutchinson, P. Sanders, and J. S. Vitter, *The power of duality for prefetching and sorting with parallel disks*, in Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA Revue), Crete Island, Greece, 2001, pp. 334–335.

[15] M. Kallahalla and P. J. Varman, *Optimal read-once parallel disk scheduling*, in Proceedings of the 6th Annual Workshop on Input/Output in Parallel and Distributed Systems, Atlanta, GA, 1999, pp. 68–77.

[16] M. Kallahalla and P. J. Varman, *Optimal prefetching and caching for parallel I/O systems*, in Proceedings of the 13th Annual Symposium on Parallel Algorithms and Architectures, Crete Island, Greece, 2001, pp. 219–228.

[17] M. Kallahalla and P. J. Varman, *PC-OPT: Optimal offline prefetching and caching for parallel I/O systems*, IEEE Trans. Comput., 51 (2002), pp. 1333–1344.

[18] T. Kimbrel and A. R. Karlin, *Near-optimal parallel prefetching and caching*, SIAM J. Comput., 29 (2000), pp. 1051–1082.

[19] D. E. Knuth, *The Art of Computer Programming—Sorting and Searching*, Vol. 3, 2nd ed., Addison Wesley, Reading, MA, 1998.

[20] M. H. Nodine and J. S. Vitter, *Deterministic distribution sort in shared and distributed memory multiprocessors*, in Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures, Velen, Germany, 1993, pp. 120–129.

[21] M. H. Nodine and J. S. Vitter, *Greed sort: Optimal deterministic sorting on parallel disks*, J. ACM, 42 (1995), pp. 919–933.

[22] P. Sanders, S. Egner, and J. Korst, *Fast concurrent access to parallel disks*, in Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, 2000, pp. 849–858.

[23] J. S. Vitter, *External memory algorithms and data structures: Dealing with MASSIVE data*, ACM Computing Surveys, 33 (2001), pp. 209–271.

[24] J. S. Vitter and D. A. Hutchinson, *Distribution sort with randomized cycling*, in Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms, Washington, D.C., 2001, ACM, New York, SIAM, Philadelphia, pp. 77–86.

[25] J. S. Vitter and E. A. M. Shriver, *Algorithms for parallel memory* I. *Two-level memories*, Algorithmica, 12 (1994), pp. 110–147.