

# Algorithmic Ramifications of Prefetching in Memory Hierarchy

Akshat Verma and Sandeep Sen

<sup>1</sup> akshatverma@in.ibm.com, IBM India Research Lab

<sup>2</sup> ssen@cse.iitd.ernet.in, Dept of Computer Science and Engineering, IIT Delhi\*

**Abstract.** External Memory models, most notable being the I-O Model [3], capture the effects of memory hierarchy and aid in algorithm design. More than a decade of architectural advancements have led to new features not captured in the I-O model - most notably the prefetching capability. We propose a relatively simple *Prefetch model* that incorporates data prefetching in the traditional I-O models and show how to design algorithms that can attain close to peak memory bandwidth. Unlike (the inverse of) memory latency, the memory bandwidth is much closer to the processing speed, thereby, intelligent use of prefetching can considerably mitigate the I-O bottleneck. For some fundamental problems, our algorithms attain running times approaching that of the idealized Random Access Machines under reasonable assumptions. Our work also explains the significantly superior performance of the I-O efficient algorithms in systems that support prefetching compared to ones that do not.

## 1 Introduction

Algorithm analysis and design are based on models of computation that must achieve a balance between abstraction and fidelity. The incorporation of memory hierarchy issues in the traditional Random Access Machine (RAM) model took some time [1, 2, 4, 3, 14, 11], eventually culminating in the I-O model of Aggarwal and Vitter[3]. The I-O model derives wide acceptance from its simplicity. It manages to redress the lack of distinction among the memory access times of the different tiers of memory in the RAM model and has been used extensively in the design of various external memory algorithms [3, 18, 9]. Further work in this direction led to the Cache model of Sen, Chatterjee and Dumeer [20] that addresses the algorithm design issues under the constraints of limited associativity in memory hierarchy. These results show an inherent gap in complexities of several problems between the RAM and the I-O models.

### 1.1 Motivation

The I-O models report their results in terms of the number of I-Os thus making an implicit assumption that every I-O has same cost. Dementiev and Sanders [9] present an efficient sorting algorithm in terms of the I-O time thus moving to a more practical metric. However, all these models assume that the cost of I-O (in terms of time) is a fixed constant. A close look at memory access reveals that I-O cost can be broken into latency (time spent in seeking to the right location) and

---

\* Part of the research done when the author was visiting University of Connecticut and supported by NSF Grant ITR-0326155

the transfer time (time spent in actual transfer of the block). Hence, a latency  $L$  is incurred before the start of the transfer of a block. A large number of techniques (like increase in bus bandwidth, advances in semiconductor technology) have led us to a stage where primary memory bandwidth is approaching processor speed. Similarly, the disk transfer times have significantly improved over the years where packing density and disk rotation speeds have greatly increased. Techniques like using parallel disks have also been useful to ensure that I-O bandwidth approaches processor speed [23, 9, 6]. Unfortunately the access latencies for both primary and secondary memory have not reduced in tandem with increase in memory bandwidth and processor speed, and I-O bottleneck is dominated by it.

The traditional approach for speeding up memory access has been to minimize the number of I-O's to reduce the total latency and its parallelization on multiple disk architectures. Pipelining and Prefetching support in contemporary architectures (including Pentium IV) [7, 10] present another possibility, namely, overlapping access latencies (For a survey on system-level prefetching support, refer to [19] and references therein). Similarly, read-ahead caches on disks prefetch data in advance to hide the latency component. Also, disk scheduling algorithms like SCAN hide latency of queued requests while serving a block. Because of the huge difference in magnitude between latency and transfer times, the potential savings in I-O times are immense. As an example, consider a scenario where we read 1,000,000 10KB blocks sequentially where each read has a latency of 10 ms and transfer time of 0.1 ms per block. In the traditional I-O model we would incur a latency for each read and the total I-O time would be approximately 3 hours, whereas, with prefetching the total time is less than 2 minutes as we incur the latency only for the first block. On current systems with system-level prefetching [19], such sequential reads take significantly less time than predicted by the I-O model (Section 5). Moreover, many recently proposed disk scheduling algorithms strive to compensate for the lack of prefetching-awareness in algorithms by *idle waiting* at a head position or waiting to build large batches of requests before sending them to the disk controller [15, 17]. In [15], the controller waits for more contiguous requests to be issued after serving a request, thus introducing idling. If such requests are issued before time using prefetching, such idling is eliminated thus increasing disk efficiency directly. Hence, incorporating prefetching in the I-O model not only ensures that the I-O times predicted by the model are meaningful but may also improve disk efficiency.

Moreover, algorithms designed for single cost I-O models [3, 12, 5] may not translate to optimal algorithms in a 2-cost prefetch model. Note that algorithms in the I-O model do not specify the relative order in which blocks are fetched. Hence, such algorithms need to find an ordering of I-Os that is prefetch efficient (formally defined in Section 4). More importantly, computing a prefetch-efficient order of blocks ahead of time for certain problems (e.g. sorting) necessitates devising new techniques. Our experimental studies in Section 5 confirm that algorithms optimal in traditional I-O models but not prefetch aware may perform very poorly as compared to prefetch-aware algorithms.

## 1.2 Relationship with other Models

In order to take advantage of prefetching, we work with a two-level memory model where the I-O cost is in terms of two parameters -  $L$  is the time to access the memory location and  $B_M$  is the transfer time for a memory block of size  $B$ . The request for accessing a block from the slower memory can be sent out prior to its actual use and moreover several such requests can be pipelined. This model has some similarities to [2] where  $L = f(x_\ell)$  is a (monotonic) function of the last address  $x_\ell$  in a block transfer and additional cost 1 thereafter. The authors had derived bounds for different families of the function  $f$ . By choosing a step function  $L$  (left open by [2]), in conjunction with other parameters of the I-O model our algorithms exploit features hitherto not analyzed. We would like to note that some of the recent experimental studies of external sorting [9] make extensive use of parallel threads which may invoke prefetching at a system level. In another approach the authors [6] look at *oblivious* sorting algorithms on a multi-disk machine where prefetching could turn out to be extremely relevant.

The design of cache-oblivious [12] algorithms has drawn a lot of attention and it may be pertinent to mention that it does not automatically take care of the issue addressed by us. The basic algorithm should be inherently recursive in nature and must be **aware** of the size of the internal memory to pipeline memory access. This is to avoid latency for every block of a (sufficiently large) sub-problem that can fit inside the internal memory. It is an interesting question, if every cache-oblivious algorithm can be converted into a prefetch-efficient algorithm by adding an extra (cache-aware) pipelined memory transfer step. A related area of study that has attracted a great deal of attention is the design of efficient system-level prefetching techniques independent of the algorithm running on the system [19, 16]. These techniques identify regular data access patterns amongst the I-O requests and prefetch data accordingly. However, if the algorithms running on such systems are unaware of prefetching, the system-level prefetchers may not be effectively used. Hence, we design algorithms that efficiently use such prefetching support to reduce I-O times.

## 2 The Prefetch Model and some Preliminaries

Aggarwal and Vitter [3] proposed an I-O model for an input of size  $N$  that reads blocks of size  $B$ , can transfer  $D$  blocks concurrently and works with a fast memory of size  $M$ . We formalize an extension of the I-O models to capture prefetching by introducing the following additional parameters -

- $B_M$  as the time<sup>3</sup> to transfer one block of memory, where  $B_M/B \geq 1$ .
- $L$  as the normalized latency in transferring from slow memory to fast memory. We always use  $L$  to denote read latency unless otherwise stated. In cases where we deal with both read and write latency,  $L_r$  denotes read latency and  $L_w$  denotes write latency.
- There is an explicit prefetch instruction and the Prefetch Latency is  $L$ .

A large block-size does not reduce the *Prefetch Model* to I-O model as the algorithms for  $B_M = L$  may not be optimal. To simplify our presentation, we

---

<sup>3</sup> All the timing parameters are normalized wrt to the instruction cycle

initially ignore the parameter  $D$  from the prefetch model and propose optimal algorithms in a single disk prefetch model. In this paper we make the following assumptions that are consistent with the existing architectures. (i)  $N > M > B$  (ii)  $(M/B)B_M > L$ <sup>4</sup> (iii)  $N, M, B$  are of the form  $2^i$  to simplify analysis - the asymptotic bounds are not affected. The fast memory size (be it cache or registers)  $M$  is typically much larger than the size of the cache line  $B$ . Moreover, the latency incurred,  $L$ , is typically much smaller than the time to load the internal memory completely ( $= \frac{M}{B}B_M$ ). Prefetch latency is typically same as the memory latency  $L$  or may differ from it by one or two cycles.

**Definition 1** *The latency  $l_i$  of block  $i$  is defined as the additional latency that is incurred because of block  $i$ . Hence, if reads for block  $(i - 1, i)$  are given at  $(t_{i-1}, t_i)$  and the blocks are available at times  $(e_{i-1}, e_i)$ , then  $l_i = e_i - \max\{t_i, e_{i-1}\}$ , where  $e_i$ 's are ordered.*

Note that for blocking reads, this definition of latency is the same as  $l_i = e_i - t_i$ , which is the one commonly used. We modify the usual definition in order to define cumulative latency of a  $m$ -block  $L_m$  simply as sum of the latency of the  $m$  blocks, where an  $m$ -block denotes a set of  $m$  consecutive blocks.

We make a note here that complete control over prefetch is not realistic and, in practice, prefetching is constrained by the number of prefetch buffers, limitations due to associativity (in a Cache Model) and a streaming behavior in data access required for most forms of prefetching. Our results can be extended in a model that includes (a) limited prefetch buffers (b) small associativity (c) limited streams support for prefetching and (d) parallel disks, for which the reader is referred to [21].

### Running time

We analyze the algorithms in terms of the total time that includes computation time and the I-O time. This is normalized with respect to the instruction cycle that takes unit time. The only I-Os (reads/writes) that we consider are I-Os to slow memory. Access to fast memory is counted along with the number of I-O operations. Since memory bandwidth is now within a small constant factor (2 to 4) of the processor speed, the running times that we derive have a multiplicative factor of  $B_M/B$ , which is  $O(1)$  when  $B_M = cB$  for some constant  $c$ .

## 2.1 Lower Bounds in the Prefetch Model

In the prefetch model, a block that has not been prefetched takes time  $B_M + L$ , whereas a block that has been prefetched takes  $B_M$  time. It is easy to see that if  $k$  is the minimum number of I-Os needed to solve a problem  $A$ , then  $kB_M + L$  is the lower bound on total time in the prefetch model. The bound is obtained by assuming that there exists a prefetch algorithm that prefetches all but the first block. Similarly, if there exists an algorithm that uses  $k$  I-Os, then  $k(L + B_M)$  is the upper bound on the I-O time by multiplying the number of I-O's by the time to transfer each block without prefetching. This upper bound is same as the *lower bound* on I-O time in the traditional I-O models and differs from the lower bound of the prefetch model by a factor of  $L/B_M$  (a factor of 1000s for

---

<sup>4</sup> Many of the technical results revolve around this assumption - c.f. Section 4.

typical disks). This general lower bound and  $(M/B)B_M > L$  combined with the bound on the number of I-Os for individual problems [3] yields the following bounds in our prefetch model. For  $D$  disks, the bounds are divided by  $D$ .

**Theorem 1.** *The worst case I-O time required to sort  $N$  records and to compute any  $N$ -input FFT digraph or an  $N$ -input permutation network is  $\Omega(\frac{N \log(1+N/B)}{\log(1+M/B)} \frac{B_M}{B})$ .*

**Theorem 2.** *The worst case I-O time required to permute  $N$  records is  $\Omega(\min\{NB_M, \frac{N \log(1+N/B)}{\log(1+M/B)} \frac{B_M}{B}\})$ .*

**Theorem 3.** *The worst case I-O time required to transpose a matrix with  $p$  rows and  $q$  columns, stored in row major order under the assumption that  $M > B^2$ , is  $\Omega(N \frac{B_M}{B})$ .*

### 3 Prefetch Model and PDM Algorithms

We now investigate similarities between algorithms in a Parallel Disk Model (PDM) and algorithms in the proposed Prefetch Model. We observe that both class of algorithms exploit essentially the same features in memory access. One may note that if a prefetch model algorithm can perform  $M/B$  I-Os in a pipelined fashion and hide the latency of all but the first of these  $M/B$  blocks, it would be efficient, i.e., it would take  $O(B_M)$  time to perform a block I-O (since  $L < (M/B)B_M$ ). Similarly, a PDM algorithm with  $M = DB$  ( $D$  is number of disks) needs to perform  $M/B$  I-Os concurrently and hence needs to predict the next  $M/B$  blocks required. Moreover, in both of these models, if the algorithm performs the minimal number of I-Os possible while maintaining the  $O(M/B)$  pipelining or parallelism respectively, the algorithm is optimal in the respective models. This general idea has also been proposed in [22] to design efficient serial algorithms from parallel versions. We now present an emulation scheme to generate *Prefetch Model* algorithms from *PDM* algorithms using this insight.

#### 3.1 PDM Emulation

We restrict *PDM* algorithms to only those parallel disk algorithms that deal with the case  $M = DB$ . The emulation works in the following manner. The sequential algorithm with prefetching performs I-O in blocks of  $D$ . It emulates the  $D$  disks as contiguous locations in  $D$  zones of the single disk. For every parallel I-O  $p_i$  performed by the PDM algorithm, let  $S_i$  be the set of  $D$  I-Os that the PDM algorithm performs concurrently. The emulation algorithm starts the prefetch of all these  $|S_i|$  blocks together. When all the  $|S_i|$  blocks are available in the fast memory, the emulation algorithm starts prefetch of the blocks in  $S_{i+1}$  corresponding to the next parallel I-O  $p_{i+1}$ . We show the following result (all proofs are omitted for lack of space and the reader is referred to [21]).

**Theorem 4.** *If the PDM algorithm performs  $k$  parallel I-Os, the corresponding sequential prefetching algorithm takes an I-O time of  $O(kDB_M)$ .*

A similar emulation scheme is obtained for parallel disk prefetch algorithms with the number of parallel disks  $D' < D$ . In a parallel disk prefetch model, we make the additional assumption that the fast memory available per disk is large enough to hide the latency for that disk, i.e.,  $\frac{M}{D'B} B_M > L$ . Each of the  $D'$  disks now emulate  $D/D'$  disks and we have the following result.

**Theorem 5.** *If the PDM algorithm performs  $k$  parallel I-Os, the corresponding parallel prefetching algorithm with  $D'$  disks takes an I-O time of  $O(kD/D'B_M)$ .*

The above emulation scheme allows us to convert existing optimal PDM algorithms to algorithm optimal in our prefetch (sequential or parallel disk) model. It is easy to see that if a PDM algorithm is optimal in the number of parallel as well as block I-Os (i.e., it performs the minimal number of parallel I-Os as well as the total number of block I-Os across all the disks is minimum), the corresponding emulated prefetch algorithm is optimal in the prefetch model. Since the lower bound for most common problems in a PDM model is a factor  $D'$  (number of disks used) less than the lower bound in a sequential I-O model, an optimal PDM algorithm is also typically optimal in the traditional single disk I-O model. Hence, in most likelihood, such optimal PDM algorithms can be directly used to generate an optimal prefetch algorithm.

A drawback of the emulation strategy described here is that it is not easy to design theoretically optimal PDM algorithms (for  $D = \Omega(M/B)$ ). Further, direct design often leads to simpler algorithms and also allows overlapping computation with memory access (which could save up to a factor of two).

## 4 Designing Optimal Algorithms Directly

The different techniques (from *prediction sequence balancing* to *sequence preservation*) employed for direct design of algorithms have a common underlying strategy: perform minimal number of I-Os in a prefetch-efficient manner, i.e., hide latency for all blocks other than the first.

**Definition 2** *An algorithm that performs  $k$  I-Os is prefetch efficient if it takes I-O time  $O(L + kB_M)$ .*

We make a note here that the assumption  $L < (M/B)B_M$  dictates the techniques that we use in designing algorithms. Observe that if  $L = lB_M$ , then a prefetch-efficient algorithm needs to prefetch  $l$  blocks in advance. Our assumption of a large  $l$  ( $= M/B$ ) covers real systems but requires our algorithms to be intelligent enough to predict the next  $M/B$  blocks and start prefetching for them. On the other hand, consider the extreme (though unrealistic) case of  $l = O(1)$ , where an optimal algorithm does not need to prefetch any blocks and any existing I-O optimal algorithms are optimal in this model.

We have essentially devised three techniques for designing optimal algorithms. We prove a general result for a class of algorithms called *sequence-preserving algorithms* and use it to design optimal algorithms for all straight-line algorithms considered (e.g., matrix transpose, permutation and FFT). We have devised a technique for *dynamic re-balancing of prefetched data* for algorithms that merge constant number of sequences (2-way sorts) and *prediction sequence balancing* for algorithms that merge large number of sequences ( $M/B$ -way sorts). We first present results for sequence-preserving algorithms and show its applications.

### 4.1 Sequence Preserving Algorithms

We define a class of straight-line algorithms that we call sequence-preserving algorithms and prove that in this class of algorithms, prefetching can hide latency. We will show later that many straight-line algorithms fall in this class. We begin with a technical lemma and some definitions.

**Lemma 1.** For any set of  $k$  pre-determined block reads, the total time needed is  $O(L + kB_M)$ .

**Definition 3** An instruction  $I_i$  precedes  $I_j$  in an algorithm  $A$  (i.e.  $I_i < I_j$ ), iff  $I_i$  is executed before  $I_j$  in  $A$ .

We define  $I^{w,i}$  and  $I^{r,i}$  as the ordered sets consisting of all the instructions that write and read respectively from memory location  $s_i$ , where the order is based on their usage time in  $A$ .

**Definition 4** The neighbourhood set  $N_I$  is defined as a set containing all the tuples of the form  $\{I_1, I_2\}$  such that  $I_1, I_2 \in \{I^{w,i} \cup I^{r,i}, I^{w,j} \cup I^{r,j}\}$  for some  $i, j$  and  $\exists I_3 : I_3 \in (I^{w,i} \cup I^{r,i} \cup I^{r,j} \cup I^{w,j})$  and  $I_1 < I_3 < I_2$  or  $I_2 < I_3 < I_1$ .

The neighbourhood set of an algorithm  $A$  consists of tuples  $\{I_1, I_2\}$  of instructions such that  $I_1$  and  $I_2$  access (read or write) memory locations  $s_i$  and  $s_j$  at times  $T_1$  and  $T_2$  respectively. Also, none of the instructions in  $A$  executed between  $T_1$  and  $T_2$  access either of the two memory locations. We also define for all instructions of the form  $I_m \in I^{w,j}$ ,  ${}^*I_m$  as the last instruction in  $I^{r,j}$  s.t.  ${}^*I_m < I_m$  and  $I_m^*$  as the first instruction in  $I^{r,j}$  s.t.  $I_m < I_m^*$ .

**Definition 5** A straight-line algorithm  $A$  is sequence preserving iff for all  $I_1$  and  $I_1^*$ , s.t. (i)  $\exists \{I_2, I_1\} \in N_I$  or  $\exists \{I_2, {}^*I_1\} \in N_I$  and (ii)  ${}^*I_2 < I_1$ ; then (a)  $I_2$  exists, (b)  $\{I_1, I_2\} \in N_I$  (c)  $I_1 < I_2 \Leftrightarrow I_1^* < I_2^*$ .

Essentially, a sequence preserving algorithm reads data in the same order as it had last written them, if it had written them earlier. Moreover, before reading any data that had been written earlier, all the reads before that write should also be written back. For the cases where any of the  $\{I_1^*, I_2^*\}$  or  $\{I_1, I_2\}$  are not defined, the corresponding precedence relation is assumed to hold by default. Using a constructive proof ([21]) of the following lemma, we convert existing I-O optimal sequence-preserving algorithms to prefetch-efficient algorithms.

**Lemma 2.** For any I-O optimal sequence-preserving straight-line algorithm  $A$ , there exists a sequence-preserving algorithm  $A'$  such that if a write of block  $s_i$  of slow memory is made at time  $t$ , the read to block  $s_i$  of slow memory is made after at least  $M/B$  block I-Os. Also,  $A'$  performs no more I-Os than  $A$  and hence is also I-O optimal.

One may also note that an I-O optimal algorithm in general may not be the optimal algorithm for that problem. However, we show that the I-O optimal algorithm for the problems considered does not take computation time more than the I-O time and hence, it is also optimal. We now state the key result for straight-line algorithms.

**Theorem 6.** Any sequence-preserving straight-line algorithm that uses  $k$  I-Os has an equivalent algorithm that takes I-O time  $O(L + kB_M)$ .

**Corollary 1.** A sequence of  $k$  pre-determined reads,  $k \geq M/B$ , takes time  $O(kB_M)$ .

**Corollary 2.** A sequence of  $k$  pre-determined reads and writes,  $k \geq M/B$ , such that no writes follow reads, i.e., there does not exist a pair  $I_1, I_2 \in \{I^{w,i}, I^{r,i}\}$  s.t.  $I_1 < I_2$  for some  $i$ , takes time  $O(kB_M)$ .

We have now characterized a class of algorithms such that prefetching is able to hide the latency in reading the blocks. We now specify a writing order for various I-O optimal algorithms and use Theorem 6 to devise algorithms optimal in the prefetch model.

**Matrix Transpose Algorithm:** We show that the following transposition-by-blocks algorithm is sequence-preserving. The algorithm transposes sub-matrices with  $B$  rows and  $B$  columns. It transposes the  $B$  rows by taking  $M/B$  rows at a time and computing the partial transposes. While writing them back, the algorithm ensures that it writes them in the order they need to be read. It then iterates till the transposition is complete. After computing all the block transposes, it rearranges the blocks in the required order taking linear time. Our writing order immediately ensures that our algorithm is sequence-preserving. If  $M/B > B$ , then the algorithm needs only one pass of the data to compute the block transposes. This leads to the following corollary of Theorem 6.

**Corollary 3.** *The total time to transpose a matrix with  $p$  rows and  $q$  columns, stored in row major order, is  $\Theta(\frac{EM}{B}N)$*

Note that even in the case that  $M < B^2$ , the above algorithm is sequence preserving. Moreover, the number of I-Os required in that case matches the lower bound of Aggarwal and Vitter for the general case [3]. Hence, the algorithm runs in time equal to the lower bound for the problem.

**General Permuting Algorithm:** Note that permuting is a special case of sorting. The algorithm for permuting is thus based on the  $M/B$ -way merge-sort algorithm of [3]. The algorithm has two phases. In the first phase, we permute the elements within runs of size  $M$ . Later, we merge the permuted runs taking them  $M/B$  at a time. The difference from merge sort though is that the next set of blocks needed is known a priori in this case. We have the following theorem for permuting.

**Theorem 7.** *The total time required to permute  $N$  records is  $\Theta(\min\{NB_M, \frac{N \log(1+N/B)}{\log(1+M/B)} \frac{B_M}{B}\})$*

We have also shown that the algorithm of Cormen et al. [8] for bit-matrix-multiply/complement (BMMC) permutation is sequence preserving and hence optimal in I-O time. Similarly, the inner loop of the algorithm for FFT and Permutation network in [3] is sequence-preserving and hence optimal.

#### 4.2 Dynamic Rebalancing: Merge Sort

We illustrate the technique of *dynamic rebalancing of prefetched data* (balancing the amount of data being prefetched across all runs) using 2-way merge sort and show that it matches the I-O time to the Compute Time, i.e.,  $O(N \log N)$ .

**Merge Sort Algorithm:** The merge sort algorithm is identical to the standard 2-way merge sort. Our prefetching strategy is the one that achieves the bounds needed. We describe our prefetching algorithm for the merging procedure of merge-sort first. We define  $A_1$  and  $A_2$  with sizes  $n_1$  and  $n_2$  as the two sorted arrays that are to be merged. Without loss of generality, we assume that  $n_1 = n_2$ .

**Case (i)  $n_1 > M/2$ :** The prefetch algorithm prefetches  $M/(2B)$  blocks of both the arrays and labels them from 1 to  $M/(2B)$ . It then prefetches the next block from  $A_1$ , if the last element of block 1 of  $A_1$  is smaller than the last element



of block 1 of  $A_2$ . Otherwise, it prefetches the next block of  $A_2$ . If it prefetches from  $A_1$ , then it decrements the label on each block of  $A_1$  by 1. Otherwise, it does the same for  $A_2$ . The prefetching evaluation is performed every  $B_M$  cycles and either of  $A_1$  or  $A_2$  is prefetched depending on the evaluation. If at any time there are no blocks of  $A_1$  left to be prefetched, the next block to be prefetched is from  $A_2$  and vice versa. If both  $A_1$  and  $A_2$  have no blocks left to be prefetched, case (ii) is followed.

**Case (ii)  $n_1 \leq M/2$ :** The prefetch algorithm prefetches  $(M/2B)$  blocks each from both the arrays and labels them from 1 to  $M/2$ . It then prefetches the next set of arrays as the blocks of  $A_1$  or  $A_2$  are written out to slow memory, i.e., at most once every  $B_M$  cycles. Note that the data manipulation in merge-sort is done only in the merging procedure. Hence, the reads are done just prior to merging. The merge-sort is performed in this manner. We initially load  $M/B$  blocks of the array and merge-sort them. We do this for all the  $(N/M)$   $M/B$ -blocks. Hence, after this step, we have  $(N/M)$   $M/B$ -blocks that are all sorted and have to be merged taken 2 at a time, with the size doubling in each iteration of merge-sort. The prefetching algorithm for merging described earlier is then used for the remaining iterations. We have the following optimality result for merge sort.

**Theorem 8.** *The total time required to sort  $N$  numbers using 2-way merge sort in the prefetch model is  $O(\frac{B_M}{B} \cdot N \log N)$ .*

### 4.3 Randomized Merge Sort with Prediction Sequence Balancing

Although the two way mergesort has  $\Theta(N \log N)$  running time, it does perform more passes than an I-O optimal algorithm. It is easy to verify that the standard  $M/B$ -way Merge Sort [3] is unable to hide the latency for most blocks because an adversary may force it to prefetch blocks out of order of their use. Since it has only constant memory available per run (as opposed to 2-way Merge Sort that had  $M/2$  memory available), it can hide latency only for a constant fraction of blocks. The strategy of using a prediction sequence [9] used for parallel disks works either for small  $N$  ( $N/B < M$ ) or requires the complication of forming large meta-blocks *a priori*, which additionally increases the constants. Similarly, Columnsort algorithm [6] uses some novel techniques to ensure that data access is deterministic but the algorithm is not defined for large  $N$ .

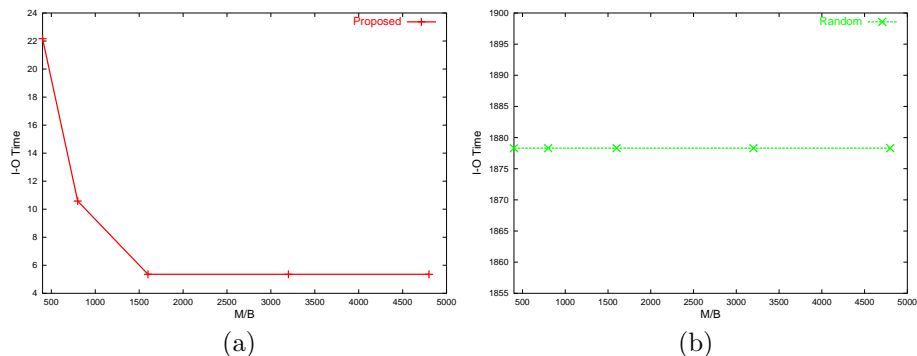
We also pursue the idea that if an algorithm  $A$  could predict the order in which blocks are needed in any merge phase of the merge sort algorithm,  $A$  would be prefetch efficient, i.e.,  $A$  would take  $O(kB_M)$  time to perform  $k$  I-Os. We show ([21]) that an  $O(M)$  sized *sliding window sample* of the prediction sequence is *sufficient* for predicting, with high probability, the order in which blocks are needed, if (a) the prediction sequence is *balanced* across all the runs being merged and (b) the input is randomized. Using the above result, the algorithm maintains one prediction sequence block from each of the  $M/B$  runs being merged in memory and uses it to prefetch blocks in advance. Hence, the technique is in some sense, a refinement of balancing prefetched data, the difference being that instead of balancing data over runs (as in Sec. 4.2), we now balance the in-memory prediction sequence across runs. For further details of the optimal

$M/B$ -way merge sort (optimalSort) algorithm and proof of its optimality, please refer to [21].

**Theorem 9.** *optimalSort sorts  $N$  records in an I-O time of  $O(\frac{N \log(1+N/B)}{\log(1+M/B)} \frac{B_M}{B})$*

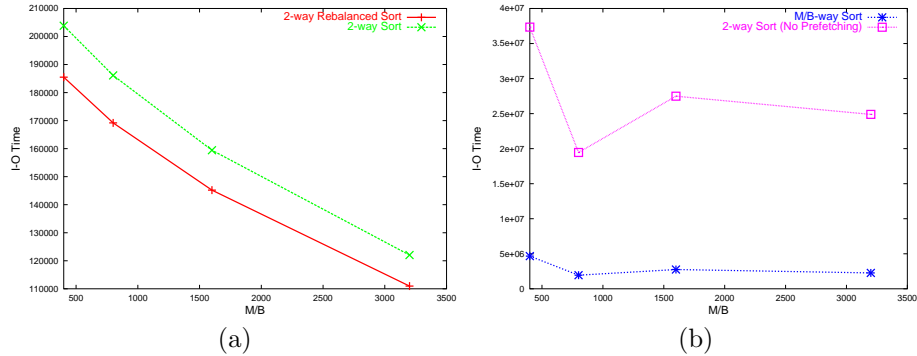
## 5 Experimental Results

We conducted a large number of experiments to study the relative performance of algorithms optimal in traditional I-O models ([3]) but prefetch-unaware as compared to algorithms that are prefetch-efficient. Matrix transpose and merge sort were used as sample problems to demonstrate the importance of incorporating prefetching when designing the algorithms. Matrix transpose represents the class of problems where the prefetch-optimal algorithm is derived from the optimal algorithm in the I-O model by finding a prefetch-efficient ordering, whereas, the standard  $M/B$ -way merge sort does not lead to any prefetch-efficient ordering and other algorithms (e.g. *2-way sort with dynamic rebalancing*) need to be devised in prefetch model. We use the *disksim* simulation environment to study the performance of various algorithms [13]. *Disksim* has been used in a large number of studies and approximates the behavior of a modern disk closely. We chose the disk model of *Seagate cheetah4LP* disk that has been validated against the real disk and matches its average response time to within 0.8%. *Seagate Cheetah4LP* supports sequential prefetching using readahead buffers. *C-SCAN* was chosen as the scheduling algorithm,  $N$  was 640000, and  $B$  was 512 Bytes.



**Fig. 1.** Performance of (a) Prefetch-efficient and (b) Random Matrix Transpose with increasing  $M/B$  (Note that scales are different)

We performed three sets of experiments with the optimal I-O model matrix transpose. In the first set, prefetching was disabled and the algorithm picked the  $B \times B$  sub-matrices in a random order. For the second set, the same algorithm was run with prefetching enabled. In the third set, the algorithm had a prefetch-efficient order (i.e. it was aware of the prefetching order and read the sub-matrices to match this order). We found that the first two sets showed no statistical difference with the second set performing marginally better in a few cases. Hence, we report only the second and third set of experiments. One may note that the random ordering of sub-matrices (Set 2), even though optimal in traditional I-O models ([3]), is not implemented in practice. We use such an ordering to demonstrate the inability of I-O models to differentiate between algorithms that



**Fig. 2.** Performance of (a) 2-way and (b) M/B-way Mergesort with increasing M/B

have very different I-O times on real systems. Note that the I-O model predicts the running time of all the 3 sets as the running time of second set but it is clear (Fig. 1) that prefetch-efficiency makes a huge difference in performance. In fact, the performance improvement (ratio of Prefetch-Unaware disk I-O time to prefetch-efficient disk I-O time) is fairly close to the maximum achievable theoretically for this disk. The disk can prefetch up to 282 blocks ahead and hence the performance improvement due to prefetching alone is bounded by 282. However, random block accesses not only leads to more disk accesses but the cost of each disk access is also higher. We looked at the logs generated by *DISKSIM* and noticed that average positioning time for random block accesses is higher than that for prefetched access. Hence, we notice that the performance improvement even exceeds the bound of 282 for large  $m$ . One may also note that prefetch-unaware algorithms (Fig. 1 (b)) fails to improve the performance with additional memory as they do not use it for prefetching whereas we use the additional memory to hide the latency of more blocks.

To evaluate the impact of prefetching on sorting, we studied the  $M/B$ -way mergesort that is optimal in the number of I-Os ([3]) but oblivious to prefetching. We compared it with the performance of our proposed 2-way mergesort that *dynamically rebalances data* and the standard 2-way mergesort. Prefetching was enabled for all the three algorithms. As a control experiment, we ran a *prefetch-disabled 2-way mergesort* (2-way sort with prefetching disabled). We observed that both the prefetch-enabled 2-way mergesorts comprehensively outperforms the  $M/B$ -way mergesort (Fig. 2). Note that the performance improvement for sorting does not approach the bound of 282. This is because 2-way mergesort performs more I-Os than the  $M/B$ -way mergesort. For the chosen value of parameters, a 2-way mergesort performs about 10 times more I-Os than  $M/B$ -way mergesort (evident from I-O time of Prefetch-disabled 2-way mergesort (Fig. 2(b)) as well). Hence, even though a 2-way mergesort has to perform a much larger number of I-Os, prefetching is not only able to compensate for it but allows it to outperform the prefetch-unaware algorithm by a significant margin. We also noticed that the average positioning time for the algorithms are almost same. Hence, prefetching alone attributes for the performance improvement of the 2-way sorting algorithm. We note that even the standard 2-way mergesort

approaches the behavior of the *rebalanced sort* we propose and comprehensively outperforms the  $M/B$ -way mergesort. This is attributed to the fact that the simple 2-way mergesort naturally uses the sequential prefetching present in disks due to readahead caches. This may be an explanation as to why the naturally prefetch-efficient standard 2-way mergesort performs better than the more sophisticated prefetch-unaware  $M/B$ -way mergesort on many real systems.

## References

1. A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of ACM Symposium on Theory of Computing*, 1987.
2. A. Aggarwal, A. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proceedings of IEEE Foundations of Computer Science*, pages 204–216, 1987.
3. A. Aggarwal and J. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
4. B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2):72–109, 1994.
5. G. S. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proceedings of STOC*, pages 307–315, 2003.
6. G. Chaudhry and T. H. Cormen. Getting more for out-of-core columnsort. In *Proceedings of ALENEX*, 2002.
7. T. Chen and J. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, 1995.
8. T.H. Cormen, T. Sundquist, and L.F. Wisniewski. Asymptotically tight bounds for performing bmmc permutations on parallel disk systems. *SIAM Journal on Computing*, 28(1):105–136, 1999.
9. R. Dementiev and P. Sanders. Asynchronous parallel disk sorting. In *Proceedings of SPAA*, 2003.
10. N. R. Adiga et al. An overview of the bluegene/l supercomputer. In *Proceedings of Supercomputing (SC)*, 2002.
11. R. Floyd. Permuting information in idealized two-level storage. In *Complexity of Computer Computations*, pages 105–109. 1972.
12. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of FOCS*, 1999.
13. B. Worthington G. Ganger and Y.Patt. The disksim simulation environment (version 2.0). In Available at <http://www.ece.cmu.edu/ganger/disksim/>.
14. J.-W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the 13th Symposium on the Theory of Computing*, may 1981.
15. S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *Proceedings of SOSP*, 2001.
16. M. Kallahalla and P. J. Varman. Optimal read-once parallel disk scheduling. In *Proceedings of IOPADS*, pages 68–77, 1999.
17. K. Lund and V. Goebel. Adaptive disk scheduling in a multimedia dbms. In *Proceedings of ACM Multimedia*, 2003.
18. U. Meyer and N. Zeh. I-o efficient undirected shortest paths. In *Proceedings of ESA*, pages 434–445, 2003.
19. K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *Proceedings of HPCA*, pages 96–105, 2004.
20. S. Sen, S. Chatterjee, and N. Dumir. Towards a theory of cache-efficient algorithms. In *Journal of the ACM*, 2002.
21. A. Verma and S. Sen. Model and algorithms for prefetching in memory hierarchy. In *Working Draft*, Available at [http://www.research.ibm.com/people/a/akshat\\_verma/akshat\\_verma.wip.html/\\$FILE/prefetch\\_main.ps](http://www.research.ibm.com/people/a/akshat_verma/akshat_verma.wip.html/$FILE/prefetch_main.ps), 2005.
22. U. Vishkin. Can parallel algorithms enhance serial implementation? In *Communications of the ACM*, 1996.
23. J. Vitter and E. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2):110–147, 1994.