CS–1998–09

External Memory Algorithms with Dynamically Changing Memory Allocations.

Rakesh Barve Jeffrey S. Vitter

Department of Computer Science

Duke University

Durham, North Carolina 27708–0129

June 1998

External Memory Algorithms with Dynamically Changing Memory Allocations.

Rakesh Barve[1]
Dept. of Computer Science
Duke University
Durham, N. C. 27708–0129

Jeffrey Scott Vitter[2]
Dept. of Computer Science
Duke University
Durham, N. C. 27708–0129.

May 1998

[1]Support was provided in part by an IBM Graduate Fellowship.
[2]Support was provided in part by the National Science Foundation under grant CCR–9522047 and by the U.S. Army Research Office under MURI grant DAAH04–96–1–0013.

Abstract

We consider the problem of devising external memory algorithms whose memory allocations can change dynamically and unpredictably at run-time. The investigation of "memory-adaptive" algorithms, which are designed to adapt to dynamically changing memory allocations, can be considered a natural extension of the investigation of traditional, non-adaptive external memory algorithms. Our study is motivated by high performance database systems and operating systems in which applications are prioritized and internal memory is dynamically allocated in accordance with the priorities. In such situations, external memory applications are expected to perform as well as possible for the current memory allocation. The computation must be reorganized to adapt to the sequence of memory allocations in an online manner.

In this paper we present a simple and natural dynamic memory allocation model. We define memory-adaptive external memory algorithms and specify what is needed for them to be dynamically optimal. Using novel techniques, we design and analyze dynamically optimal memory-adaptive algorithms for the problems of sorting, permuting, FFT, permutation networks, (standard) matrix multiplication and LU decomposition. We also present a dynamically optimal (in an amortized sense) memory-adaptive version of the buffer tree, a generic external memory data structure for a large number of batched dynamic applications. We show that a previously devised approach to memory-adaptive external mergesort is provably nonoptimal because of fundamental drawbacks. The lower bound proof techniques for sorting and matrix multiplication are fundamentally distinct techniques, and they are invoked by most other external memory lower bounds; hence we anticipate that the techniques presented here will apply to many external memory problems.

1 Introduction and Motivation

The disparity between the fast access time of main memory and the slow access time of magnetic disk external memory is the bottleneck in many large-scale applications and high performance systems. The I/O bottleneck gets accentuated as processors become increasingly faster with respect to disks, prompting ongoing research and development of *external memory* (or *out-of-core*) algorithms [Vit98].

Previous work in the field assumes that a statically allocated internal memory capable of holding M items is available throughout the execution of the external memory algorithm. A natural extension is to consider the performance of an external memory algorithm when the size of the available internal memory varies dynamically because of other ongoing activity on the computing machine. A common technique to attain high performance in database systems and operating systems is to prioritize applications or queries and to allocate available internal memory dynamically in accordance with the priorities; see [PCL93] for further references and discussion. As a result, external memory applications and queries are expected to efficiently *adapt* to situations in which portions of their internal memory are taken away from them or are allocated to them *unpredictably* and *dynamically*, in course of the execution of the computation. External memory algorithms that can adapt to dynamically changing amounts of internal memory are said to be *memory-adaptive* [PCL93]. Memory-adaptive algorithms should perform as efficiently as possible when memory is scarce and should take advantage of extra memory when it becomes available. The computation must be reorganized to adapt to the sequence of memory allocations in an online manner. To the authors' knowledge the only previous work on memory-adaptive algorithms is by Pang et al. [PCL93], who studied memory-adaptive sorting in an empirical framework, ans subsequently by Zhang and Larson [ZL97], who carried out further empirical studies of mergesorting in a model allowing a limited form of dynamic memory allocation.

In the next section, we present a realistic model for the design and analysis of memory-adaptive algorithms and we define *dynamically optimal* memory-adaptive algorithms. In Section 3, we present asymptotically tight *resource consumption* bounds for key problems such as permuting, sorting, FFT, permutation networks and matrix multiplication. Our lower bounds provide a reinterpretation of the lower bounds of [AV88] and [HK81] in a dynamic memory allocation context. In order to prove algorithms for the above problems to be dynamically optimal, we define natural, application-specific measures for the resource-consumption at each I/O step. The

measures determine how efficiently an algorithm adapts to memory fluctuations. In the remaining sections, besides the above problems, we also show how to design and analyze dynamically optimal algorithms for the a memory-adaptive version of the buffer tree [Arg94] and LU decomposition [WGWR93]. The lower bound proof techniques for sorting and related problems on the one hand, and the problem of matrix multiplication on the other, are fundamentally distinct techniques, and they are invoked by most other external memory lower bounds; hence we anticipate that the techniques presented here will apply to many external memory problems [VS94, GTVV93, CGG+95, AV96, Arg94, VV95].

In Section 4 we discuss an approach to design memory-adaptive algorithms using optimal static memory I/O algorithms and then provide intuition on the nature of difficulties caused by dynamic memory allocation.

Sections 5 through Section 10 discuss various aspects of memory-adaptive mergesorting. In Section 5, we present a natural framework to design memory-adaptive mergesort algorithms. In Section 6, we define the fundamental notion of merge potential, which is meant to quantify the "progress" made by a memory-adaptive merge algorithm up to any time during its execution. In Section 7, we use an adversarial argument and construct a "nemesis" sequence of memory allocations to prove that two variants of a mergesort algorithm based on the memory-adaptive merging techniques proposed by Pang et al. [PCL93] are not dynamically optimal. The notion of merge potential helps us isolate the fundamental drawback of the merging techniques of [PCL93]. In Section 8 we present an efficient and elegant memory-adaptive merging algorithm that forms the basis of our dynamically optimal sorting algorithm. Our algorithm uses novel data structures and online techniques to reorganize merge computation in response to dynamic fluctuations in memory. In Section 9, we analyze the resource consumption of mergesort our algorithm and establish that it is dynamically optimal. In Section 10, we discuss how, besides general insights into the problem of memory-adaptive merging, the notion of merge potential also gives interesting insights into the difference in the resource consumption required of a memory-adaptive merging algorithm when it is used as a subroutine during a mergesort and when it is used in isolation as a dynamically optimal algorithm for merging.

In Section 11, we show how the sorting algorithm can be used to obtain memory-adaptive algorithms for permuting, FFT and permutation networks. In Section 12, we show to apply our memory-adaptive sorting technique to implement the *buffer emptying* operation of the I/O buffer tree [Arg94], realizing a dynamically optimal (in an amortized sense) memory-adaptive buffer tree. This result is particularly significant with respect to the extendibility of our techniques to diverse applications since the buffer tree is an I/O optimal data structure for several several applications involving batched dynamic problems [Arg96, Vit98], including such time consuming operations as bulk-loading of B-trees and R-trees [AHVV98].

In Section 13, we present simple techniques resulting in a dynamically optimal algorithm for memory-adaptive matrix multiplication and LU factorization; our techniques can thus form a basis for memory-adaptive scientific computing.

2 Dynamic Memory Model

We wish to enable an external memory algorithm to perform efficiently even when the amount of allocated internal memory fluctuates unpredictably. The amount of internal memory is dynamically determined by the resource allocator, which is typically a database system or an operating system. In this section we propose a model for how a resource allocator may dynamically change the memory allocations of an algorithm during "run time", and, we specify when memory-adaptive algorithms are dynamically optimal.

Definition 1 We assume that the resource allocator allocates memory blocks to the external memory algorithm in a sequence of *allocation phases*. Each allocation phase is characterized by its *size*. Consider an external memory algorithm $\mathcal{A}$ with an input of size $n = N/B$ disk blocks. In the *dynamic memory* model, the resource allocation to $\mathcal{A}$ consists of a sequence of phases (called the *allocation sequence*) of sizes $s_1, s_2, s_3, \ldots$, where s_i is the size of the ith phase. The following constraints are met by the allocation phases:

1. In each I/O operation at most B contiguous items can be transferred between internal memory and disk.

2. During the ith allocation phase, the external memory algorithm $\mathcal{A}$ is allocated exactly s_i internal memory blocks by the resource allocator. These s_i blocks are $\mathcal{A}$'s to use as it sees fit until it executes $2s_i$ I/O operations. The external memory algorithm $\mathcal{A}$ can voluntarily terminate an allocation phase of size s_i before completing $2s_i$ I/O operations during that phase.

3. For each i, we have $\kappa_{model} \leq s_i \leq m_{max}$ where $\kappa_{model} \geq 4$ is some constant to be determined later and $m_{max} = \min\{cn, phy_{max}\}$ where phy_{max} is the maximum number of internal blocks the physical memory of the computer can accomodate and c is an application-specific positive constant.

The maximum allocation is trivially bounded by cn. Assuming that c is defined appropriately for the given application, the algorithm $\mathcal{A}$ can complete all computation if the internal memory allocation is cn blocks.

Definition 2 A *memory-adaptive* algorithm is an algorithm that adheres to the dynamic memory model of Definition 1. We assume that at any time a memory-adaptive algorithm $\mathcal{A}$ has access to the following in-memory variables relevant to $\mathcal{A}$ that can be easily maintained by the resource allocator in internal memory:

1. Variable *mem*, which contains the size of the ongoing allocation phase of algorithm $\mathcal{A}$.

2. Variable *left*, which contains the number of I/O operations remaining in the ongoing allocation phase of algorithm $\mathcal{A}$.

3. Variable *next*, which contains the size of the next allocation phase[1] of algorithm $\mathcal{A}$.

The internal memory resource allocator (adversary) has tremendous flexibility since it can dynamically choose allocation phases of arbitrary sizes, varying from κ_{model} blocks to the maximum possible memory allocation of m_{max} blocks. The memory-adaptive algorithm has to adapt to sizes of allocation phases in an online manner. Requirement 2 of Definition 1, which specifies that each memory allocation of m blocks must last for $2m$ I/Os, is a very natural assumption: The duration of allocation enables the memory-adaptive algorithm to load up to m blocks into memory, carry out internal memory computation, and then write up to m blocks back to disk so it is long enough to allow all m internal memory blocks to be used. That requirement is implicitly met in conventional *virtual memory paging* systems designed for non-adaptive external memory algorithms. For example, in a virtual memory system suppose at some point an application has m memory blocks. Now, if, for some reason the virtual memory system decides to leave that application with only (say) $\sqrt{m}$ memory blocks to create internal memory space for some other higher-priority application, the virtual memory system would immediately have to write $m - \sqrt{m} = \Theta(m)$ blocks to disk, thereby incurring $\Theta(m)$ I/O operations.

2.1 Dynamically Optimal Memory-Adaptive Algorithms

We now define what it means for a memory-adaptive algorithm to be dynamically optimal.

Definition 3 Consider a computational problem $\mathcal{P}$ and a memory-adaptive algorithm $\mathcal{A}$ that solves $\mathcal{P}$. Given any N-sized instance I_N of $\mathcal{P}$, we say that algorithm $\mathcal{A}$ *solves* I_N *during allocation sequence* σ if $\mathcal{A}$ begins execution with the first phase of σ and completes execution by the end of σ. We say that $\mathcal{A}$ *solves* $\mathcal{P}$ *during allocation sequence* σ if $\mathcal{A}$ can solve any instance I_N of $\mathcal{P}$ during σ.

Definition 4 Consider a memory-adaptive algorithm $\mathcal{A}$ for problem $\mathcal{P}$. We say that $\mathcal{A}$ is *dynamically optimal* for $\mathcal{P}$ if, for all minimal allocation sequences σ such that $\mathcal{A}$ solves $\mathcal{P}$ during σ (but $\mathcal{A}$ does not solve $\mathcal{P}$ during a proper prefix of σ), no other memory-adaptive algorithm can solve $\mathcal{P}$ more than a constant number of times during σ.

For instance suppose that a memory-adaptive sorting algorithm A_S can sort a file of N items during an allocation sequence $\sigma = s_1, s_2, \ldots, s_\ell$. Then for A_S to be dynamically optimal there must be no more than a constant number c of non-overlapping contiguous subsequences $\sigma_1, \sigma_2, \ldots, \sigma_c$ of σ such that some memory-adaptive sorting algorithm A_S^* can sort an arbitrary N-sized file during each σ_i.

[1] The variable *next* is relevant only in practice: our techniques (with minor modifications) and theoretical results hold even when the resource allocator cannot provide this information about the next phase until that phase begins.

3 Memory-Adaptive Lower Bounds

We now present asymptotically tight bounds for the memory and I/O resources consumed by memory-adaptive algorithms for the fundamental problems of permuting, sorting, fast fourier transform (FFT), permutation networks, buffer tree operations and matrix multiplication. The problem of permuting a file of N items is the same as sorting a file of N items except that the key values of the N items in the output are required to form a permutation of $\{1, 2, \ldots, N\}$. Buffer tree operations refer to operations on a memory-adaptive version of Arge's buffer-tree [Arg94] data structure, discussed in Section 12.

In this section, we prove only the pertinent lower bounds; the upper bounds are proved in subsequent sections as indicated below. Based on the lower bound on the resources that are needed to solve a problem, we can define the resource consumption at each I/O step of a memory-adaptive algorithm for that problem.

In the following theorem, we use the notion of dynamically optimal algorithms to present our resource consumption bounds.

Theorem 1 *Suppose that A is a memory-adaptive algorithm that finishes its computation during an allocation sequence σ of sizes $m_1, m_2, \ldots, m_{\ell(A)}$. Let T_A denote the total number $\sum_{j=1}^{\ell(A)} 2m_j$ of I/O operations incurred by A.*

1. *Suppose that A is a dynamically optimal algorithm for permuting, then*

$$\frac{1}{B}\left(T_A \lg N\right) + \sum_{j=1}^{\ell(A)} 2m_j \lg m_j \quad = \quad \Theta(n \lg n). \tag{1}$$

2. *Suppose that A is a dynamically optimal algorithm for sorting or FFT or permutation networks or executing a sequence of insert/delete operations[2] on our memory-adaptive buffer tree. Then we have*

$$\sum_{j=1}^{\ell(A)} 2m_j \lg m_j \quad = \quad \Theta(n \lg n). \tag{2}$$

3. *Suppose that A is a dynamically optimal algorithm for (standard) matrix multiplication of two $\hat{N} \times \hat{N}$ matrices, or LU decomposition of an $\hat{N} \times \hat{N}$ matrix[3]. Then*

$$\sum_{j=1}^{\ell(A)} m_j^{3/2} \quad = \quad \Theta(n^{3/2}). \tag{3}$$

The buffer tree alluded to in the theorem is a memory-adaptive version of the original buffer tree [Arg94]. It is described in Section 12. The bounds in Theorem 1 lead to natural notions of resource consumption of memory-adaptive algorithms for the various problems discussed.

[2] In the case of the buffer tree, N denotes the number of insert/delete operations.

[3] In the case of matric multiplication and LU decomposition, N denotes $\hat{N}^2$.

Definition 5 Consider the $2m$ I/O operations of a memory-adaptive algorithm A during any allocation phase of size m.

1. If A is a permuting algorithm, the *resource consumption* of each I/O operation is defined to be the quantity $\frac{1}{B}\lg N + \lg m$.
2. If A is an algorithm for sorting, FFT, permutation networks, or for executing a sequence of operations on a buffer tree [Arg94], then the *resource consumption* of each I/O operation is defined to be the quantity $\lg m$.
3. If A is an algorithm for matrix multiplication or LU decomposition, then the *resource consumption* of each I/O operation is defined to be the quantity $\sqrt{m}/2$.

The *resource consumption* of algorithm A is defined to be the sum of the resource consumptions the I/O operations of A.

We can recast Theorem 1 in terms of resource consumption as follows:

Corollary 1 *A memory-adaptive algorithm A is dynamically optimal if and only if its resource consumption is $\Theta(n \lg n)$ for permuting, $\Theta(n \lg n)$ for sorting, FFT, permutation networks, and buffer tree operations, and $\Theta(n^{3/2})$ for (standard) matrix multiplication and LU decomposition.*

Below, we prove the lower bounds implicit in Theorem 1 for permuting, sorting, FFT, permutation networks and matrix multiplication by reinterpreting the original I/O lower bounds proved in [AV88] and [HK81, SV87] in a dynamic memory context. The lower bound for buffer tree operations can be proved by adapting the arguments of [AKL93] relating comparison tree lower bounds to I/O lower bounds to the dynamic memory model.

In Section 8 and Section 13, we present dynamically optimal algorithms for sorting and matrix multiplication respectively. We demonstrate optimality in each case by showing that the resource consumption meets the bound given above. In Section 11, we show how to apply our memory-adaptive mergesort and related techniques to obtain dynamically optimal algorithms for permuting, FFT and permutation networks. In Section 12, we show how to use our sorting algorithm as a subroutine to devise a dynamically optimal memory-adaptive buffer tree. By observations made in [WGWR93], the dynamically optimal algorithm developed for matrix multiplication can be modified to obtain a dynamically optimal algorithm for LU factorization.

3.1 Memory-Adaptive Lower Bounds for Permuting

Now we prove a lower bound on the resource consumption incurred by *any* memory-adaptive algorithm to permute a file of n blocks of items.

Theorem 2 *Consider any memory-adaptive algorithm A that permutes a file containing $N = nB$ items during the allocation sequence $\sigma = m_1, m_2, \ldots, m_{\ell(A)}$. Let T_A denote the total number $\sum_{i=1}^{\ell(A)} 2m_i$ of I/O operations incurred by A. Then we have*

$$\frac{1}{B}(T_A \lg N) + \sum_{i=1}^{\ell(A)} 2m_i \lg m_i \quad = \quad \Omega(n \lg n), \tag{4}$$

where, by definition, the left hand side is the resource consumption of algorithm A. In the case when $\frac{1}{B}(T_A \lg N) \geq \sum_{i=1}^{\ell(A)} 2m_i \lg m_i$, the bound (4) implies the lower bound

$$T_A = \Omega(N). \tag{5}$$

Otherwise, the bound (4) implies the lower bound

$$\sum_{i=1}^{\ell(A)} (2m_i \lg m_i) = \Omega(n \lg n). \tag{6}$$

Proof: Without loss of generality, we make the following assumptions made in [AV88]: I/O operations are "simple" and respect block boundaries. If a block of B items is input into memory at any time and those B items had been output to disk during an earlier output operation, then we assume that the relative order of those B items was computed when they were last together in internal memory. We also assume that if $mB - B$ items reside in memory at the time of initiating an input operation, then the relative ordering of the mB in-memory items is determined on completion of the input operation.

Consider any one of the $2m_i$ I/O operations of the ith allocation phase m_i of algorithm A and suppose it is an input I/O operation. There are less than $n + \sum_{j=1}^{i} 2m_i \leq n + T_A \leq N \lg N$ blocks on disk of which A must read one, so there are no more than $N \lg N$ choices available to A. Let us consider how the number of realizable orderings changes when a given disk block is read into internal memory. By definition, the maximum number of in-memory items at the time of any I/O operation during the ith phase is $M_i - B$, where $M_i = m_i B$. There are at most B items in the input block and they can intersperse among the M_i items in internal memory in at most $\binom{M_i}{B}$ ways, so the number of realizable orderings increases by a factor of $\binom{M_i}{B}$. If the input block has never before resided in internal memory, the number of realizable orderings increases by a $B!$ factor since the B items can be permuted amongst themselves. (This extra contribution of $B!$ can happen only once for each of the n original blocks.) The increase in the number of realizable orderings from writing a disk block is considerably less than reading it. Thus the number of distinct orderings that can be realized by algorithm A increases by a factor of at most

$$(B!)^{i'} \times \left(N(\lg N) \binom{M_i}{B} \right)^{2m_i}$$

during the ith allocation phase, where i' is the number of previously unread blocks read during the ith phase. The total number of distinct orderings that can be realized by A during allocation sequence σ is no more than

$$(B!)^{N/B} \prod_{1 \leq i \leq \ell(A)} \left(N(\lg N) \binom{M_i}{B} \right)^{2m_i}. \tag{7}$$

Setting the above expression to be at least $N!/2$, taking logarithms and applying Stirling's formula [Knu97], we have

$$N \lg B + \sum_{i=1}^{\ell(A)} 2m_i \left(\lg N + B \lg \frac{M_i}{B} \right) = \Omega(N \lg N)$$

$$\Longrightarrow \qquad \sum_{i=1}^{\ell(A)} 2m_i \left(\lg N + B \lg \frac{M_i}{B}\right) = \Omega\left(N \lg \frac{N}{B}\right),$$

which can be simplified further to

$$\begin{aligned} \sum_{i=1}^{\ell(A)} 2m_i \left(\lg N + B \lg m_i\right) &= \Omega(N \lg n) \\ \Longrightarrow \qquad T_A \lg N + \sum_{i=1}^{\ell(A)} 2m_i B \lg m_i &= \Omega(N \lg n). \end{aligned}$$

Dividing throughout by B and simplifying, we establish the lower bound (4).

We now have to consider two separate cases. First we consider the case $\frac{1}{B}(T_A \lg N) \geq \sum_{i=1}^{\ell(A)} 2m_i \lg m_i$, so that we have the lower bound

$$\frac{1}{B}\left(T_A \lg N\right) = \Omega(n \lg n). \tag{8}$$

Since $\frac{1}{B}(T_A \lg N) \geq \sum_{i=1}^{\ell(A)} 2m_i \lg m_i$, we have

$$\lg N \geq \frac{B}{T_A}\left(T_A \lg(\min_{1 \leq i \leq \ell(A)} \{m_i\})\right). \tag{9}$$

Using the fact that

$$\lg(\min_{1 \leq i \leq \ell(A)} \{m_i\}) \geq \lg \kappa_{model},$$

where κ_{model} is the constant defined in the dynamic memory model, we have

$$\lg(\min_{1 \leq i \leq \ell(A)} \{m_i\}) \geq c' \geq 2.$$

By (9), we have $\lg N \geq c'B$ and so

$$\begin{aligned} N^{1/c'} &= (2^{\lg N})^{1/c'} \\ &\geq (2^{c'B})^{1/c'} \\ &\geq 2^B, \end{aligned}$$

which implies that $B < \sqrt{N}$. The bound (5) follows from (8) after some simplification.

In the case when $\frac{1}{B}(T_A \lg N) < \sum_{i=1}^{\ell(A)} \lg m_i$, the lower bound (6) follows trivially from (4). □

The lower bound (1) on the resource consumption of dynamically optimal permuting algorithms in Theorem 1 follows from (4).

Intuitively, Theorem 2 says that the resource consumption of permuting is identical to the resource consumption of sorting (Theorem 1) except when the allocation sequence is what we call a *scanty allocation sequence*. An allocation sequence of ℓ phases $m_1, m_2, \ldots, m_\ell$ is said to be scanty if

$$(\lg N) \sum_{i=1}^{\ell} 2m_i = \Omega(N \lg n)$$

but

$$\sum_{i=1}^{\ell} 2m_i \lg m_i \leq \frac{1}{B}(\lg N) \sum_{i=1}^{\ell} 2m_i.$$

It is unlikely for an allocation sequence to be scanty in practice, so, in the most likely case when an allocation sequence is not scanty, the resource consumption of permuting is identical to that of sorting. Interestingly, when the allocation sequence is scanty, even the naive permuting algorithm incurring $T = \Theta(N)$ I/O operations is a dynamically optimal permuting algorithm.

3.2 Memory-Adaptive Lower Bounds for Sorting, FFT and Permutation Networks

Permuting is a special case of sorting, and hence the lower bound for permuting applies to sorting as well. However, in the case in which the naive $\Theta(N)$-I/O permuting algorithm is dynamically optimal for permuting, we can prove a stronger lower bound for sorting using an adversarial argument and the comparison model of computation. Using the same notation as in Theorem 2 and arguments similar to the ones in [AV88] we can show that the maximum number of total orders consistent with comparisons made during the allocation sequence $m_1, m_2, \ldots, m_{\ell(A)}$ is no more than

$$(B!)^{N/B} \prod_{1 \leq i \leq \ell(A)} \binom{M_i}{B}^{2m_i}. \tag{10}$$

Using arguments similar to the ones in [AV88] we can show that the maximum number of realizable orderings for a memory-adaptive permutation network algorithm during $\ell(A)$ allocation phases is no more than (10) as well. In contrast to sorting, the proof of the above bound for permutation networks does not involve any comparison model based arguments arguments and follows directly from a counting argument.

As in [AV88], we can exploit the fact that any permutation network can be constructed by stacking together at most three appropriate FFT networks to conclude that the FFT and permutation network problems are essentially equivalent. Thus the problem of FFT computation has the same lower bound as that for permutation networks.

The theorem below follows by setting the expression (10) to be at least $N!$, taking logarithms, applying Stirling's inequality [Knu97] and then simplifying.

Theorem 3 *Consider any memory-adaptive algorithm A for sorting a file containing $N = nB$ items or for computing the N-input FFT digraph or an N-input permutation network during the allocation sequence $\sigma = m_1, m_2, \ldots, m_{\ell(A)}$. Then we have*

$$\sum_{i=1}^{\ell_A} 2m_i \lg m_i \quad = \quad \Omega(n \lg n), \tag{11}$$

where, by definition, the left hand side is the resource consumption of algorithm A.

The lower bound (2) of Theorem 1 follows from Theorem 3.

3.3 Memory-Adaptive Lower Bounds for Matrix Multiplication

We now derive lower bounds on the resources consumed by any memory-adaptive algorithm to multiply two $\hat{N} \times \hat{N}$ matrices.

Consider the problem of multiplying two $\hat{N} \times \hat{N}$ matrices, consisting of $N = \hat{N}^2$ elements in total. Hong and Kung [HK81] proved fundamental I/O bounds for this problem using *graph pebbling* arguments. Based on their arguments, Savage and Vitter [SV87] showed that if the amount of main memory available to an external memory algorithm is fixed to be M then any (standard) matrix multiplication algorithm takes at least $\Omega(N^{3/2}/B\sqrt{M}) = \Omega(n^{3/2}/\sqrt{m})$ I/O operations, where $n = N/B$ and $m = M/B$. This bound is easily realized by a simple external memory algorithm [VS94].

The problem of computing the product $C = A \times B$ of two $\hat{N} \times \hat{N}$ matrices can be viewed as pebbling through a directed acyclic graph (DAG) containing $\Theta(\hat{N}^3)$ constant degree nodes. For our purposes, it suffices to say that in order to compute C it is necessary to pebble through all $\Theta(\hat{N}^3)$ nodes of the matrix multiplication DAG. We refer the reader to [SV87] for a study of issues related to pebbling-based I/O complexity arguments.

In order to adapt the lower bound argument for matrix multiplication to our memory-adaptive setting we consider the maximum number of nodes (of the matrix multiplication DAG) that may be pebbled in a single allocation phase of size s; that is, the maximum number of DAG nodes pebbled using at most s internal memory blocks and no more than $2s$ I/O operations. We state the following lemma whose proof follows from the arguments of [SV87] which are based on ideas from [HK81].

Lemma 1 *Let the pebbling operations of matrix multiplication be as defined in [SV87]. The maximum number of matrix multiplication DAG nodes that can be pebbled during an allocation phase of size m is $O(M^{3/2})$, where $M = mB \leq \hat{N}^2$.*

The lower bound of Theorem 1 for matrix multiplication follows easily from Lemma 1. The same lower bound applies to any standard LU factorization algorithm as well: This is because the DAG corresponding to a standard LU factorization algorithm on an $\hat{N} \times \hat{N}$ matrix also contains $\Theta(\hat{N}^3)$ nodes that need to be pebbled and a version of Lemma 1 is applicable to an LU factorization DAG.

4 Designing Memory-Adaptive Algorithms

We now consider how memory-adaptive algorithms can make "optimum utilization" of the memory and I/O resources comprising an m-sized allocation phase. In order to get an idea of how this may be achieved, we examine external memory algorithms that are optimal for a fixed internal memory allocation of m blocks.

4.1 Optimal non-adaptive external memory algorithms

Consider an optimal external mergesort algorithm. Given a fixed number m of internal memory blocks an optimal mergesort algorithm consists of

a run-formation stage followed by a sequence of $u(m)$-way external merge operations, where $u(m) = \Theta(m^c)$ for $0 < c \leq 1$. During a typical sequence of $\Theta(m)$ I/O operations, this algorithm's execution consists of generating $\Theta(m)$ blocks output by an $u(m)$-way external merge. Now consider the design of an external memory algorithm to carry out the matrix multiplication $AB = C$. Given a fixed number m of internal memory blocks, an optimal matrix multiplication algorithm consists of a sequence of "$v(m)$-operations" each of which consists of carrying out an internal memory multiplication of submatrices of A and B each consisting of $v(m)$ blocks, where $v(m) = \Theta(m)$. Clearly each such operation consists of $\Theta(m)$ I/O operations.

4.2 Mimicking Optimal Static Memory Algorithms

The computation carried out by an optimal external memory algorithm over a sequence of $\Theta(m)$ I/O operations is determined by the number m of internal memory blocks allocated to the algorithm. Optimality results from the fact that the algorithm achieves "optimal resource utilization" during a typical allocation phase of size m. By the above reasoning, for a memory-adaptive algorithm to be efficient, the computation it carries out over each allocation phase should be determined by the size of that phase. In order to attain dynamic optimality, a memory-adaptive algorithm's "progress" during an allocation phase of size m should mimic be comparable to that of its optimal, fixed-memory analog over $\Theta(m)$ I/O operations. Ideally, during each allocation phase of size m, where $1 \leq m \leq m_{max}$, a memory-adaptive mergesort algorithm should write to disk $\Theta(m)$ blocks resulting from $u(m)$-way merging and a memory-adaptive matrix multiplication algorithm should execute a $v(m)$-operation.

4.3 Adaptive Organization of Computation

The challenge in designing dynamically optimal memory-adaptive algorithms is to organize, in an *efficient* and *online* manner, the external memory computation so as to attain optimal resource utilization during every allocation phase as described above. The issue of efficiency in this online organization of external memory computation arises in two contexts.

Firstly, in order to cope with arbitrary variation in allocations, the computation needs to be broken down in an online manner into a sequence of "smaller granularity" computations such that executing that sequence of computations is a resource-efficient rendering of the original computation. For instance, an s-way merge of runs of the set $U = \{r_0, r_1, \ldots, r_{s-1}\}$ can be reorganized as follows: First we can compute the h runs of the set $U' = \{r'_0, r'_1, \ldots, r'_{h-1}\}$ such that run r'_i, where $0 \leq i \leq h-1$, is a merge of s_i runs of U and $\sum_{i=0}^{h-1} s_i = s$. For each i, j such that $i \neq j$ and $0 \leq i, j \leq h-1$, the s_i runs merged to obtain r'_i are all distinct from the s'_j runs merged to obtain r'_j. Then we can merge together the runs of the set U' to complete the original merge of runs of the set U. Each of h merge operations producing $r'_0, r'_1, \ldots, r'_{h-1}$ may or may not subsequently need to be broken down further into smaller merge operations and so on, depending on the allocation sequence. The dynamic optimality (or non-optimaility) of a

memory-adaptive mergesort depends on how it decides h, s_i and the s_i runs chosen to produce run r'_i. Similarly a computation consisting of mutliplying matrices, each consisting of s blocks, can be achieved by carrying out a series of appropriately chosen matrix multiplication operations each multiplying matrices consisting of $s' < s$ blocks. Each s'-sized multiplication operation may or may not need to be broken down into smaller operations, depending on the allocation sequence. The dynamic optimality of a memory-adaptive multiplication depends on how it determines the value of s'.

The second context in which efficiency is important is while breaking down a computation into subcomputation: Since such activity may also involve external memory operations, the data structures and techniques chosen to reorganize external memory computation in an online manner must themselves be efficient.

4.4 Allocation Levels

As observed in Section 4.2, a memory-adaptive algorithm's computation in an allocation phase should be determined by the size of the phase. While mimicking an optimal external memory algorithm, it is convenient to clump together ranges of allocation phase sizes into single allocation levels and thereby partition the whole range $\{\kappa_{model}, \ldots, m_{max}\}$ of allocation phase sizes into several allocation levels. Consider using the following mimicking strategy in an attempt to devise a dynamically optimal mergesort algorithm: During an allocation phase of size m, we always try to compute $\Theta(m)$ blocks corresponding to the output of a $u(m)$-way merge, where $u(m) = 2^{\lfloor \lg(m-1) \rfloor} \geq m/2$. In this case, any allocation phase of size s such that $s \in \{2^\ell + 1, 2^\ell + 2, \ldots, 2^{\ell+1}\}$ is said to be at allocation level ℓ with respect to our strategy since we always try to output $\Theta(s)$ blocks of a 2^ℓ-way merge during that allocation phase. On the other hand, suppose we set $u(m) = 2^{2^{\lfloor \lg \lg(m-1) \rfloor}} \geq \sqrt{m}$, then any phase of size $s \in \{2^{2^\ell} + 1, 2^{2^\ell} + 2, \ldots, 2^{2^{\ell+1}}\}$ can be said at allocation level ℓ with respect to the modified strategy. In this paper, we formally define allocation levels for each memory-adaptive algorithm we present.

5 A Framework for Memory-Adaptive Mergesort

In this section, we discuss a simple framework that can be used to construct memory-adaptive mergesort algorithms. The memory-adaptive mergesort based on techniques proposed by Pang et al. [PCL93] can be cast in terms of our framwork. We prove, however, that the techniques of [PCL93] suffer from fundamental drawbacks that make the resulting sorting algorithm nonoptimal. In Section 8, we design a new memory-adaptive merging algorithm that yields a dynamically optimal sorting algorithm when applied using our framework.

External mergesorts [Knu98, AV88] consist of a *run formation stage* in which sorted runs are formed (by reading in memoryloads of items, sorting them and writing them out to disk) followed by a *merging stage* in which the mergesort algorithm keeps merging (as large as possible) a number of

runs together until there is only one run remaining. Thus we need to devise memory-adaptive techniques for run formation and merging.

5.1 Memory-Adaptive Run Formation

The straightforward memory-adaptive run formation technique we propose is as follows: In an allocation phase of size m, read in m blocks of items from the input file, sort them using an optimal in-place sorting technique, and write them out using m write I/O operations.[4] The number and the lengths of the runs formed during run formation depends on the allocation sequence.

The following lemma follows easily from the above definition.

Lemma 2 *For a file of n blocks, each run that is formed using the above memory-adaptive run formation strategy, with the possible exception of one run, is at least κ_{model} blocks long and the number of runs formed is at most $\lceil n/\kappa_{model} \rceil$. The resource consumption of the above run formation strategy is no more than $2n \lg m_{max}$, where m_{max} is defined in Definition 1.*

5.2 Memory-adaptive merging stage

The main difficulty of memory-adaptive mergesorting lies in merging. If an external mergesort algorithm has a static number m of internal memory blocks to use throughout the algorithm, an I/O-optimal strategy [AV88] is to merge together the $m-1$ shortest runs and to repeat the process until a single sorted run remains.[5] Thus if the input file is n blocks long and there are n' sorted runs after run formation, $\lceil \log_{m-1} n' \rceil = \lceil (\lg n')/\lg(m-1) \rceil$ *merge passes* are required to complete the sort.

The merging stage we propose is a modification of the above approach. Let $\mathcal{Q}$ be the queue of (pointers to) runs that need to be merged during the merging stage. We implement $\mathcal{Q}$ as a blocked list of pointers. Whenever a run is formed suring the run-formation stage a pointer to that run is inserted into $\mathcal{Q}$. At the end of run formation $\mathcal{Q}$ can have up to $\lceil n/\kappa_{model} \rceil$ pointers. Let $\mathcal{M}$ be any memory-adaptive merge subroutine that can merge up to $\rho_{\mathcal{M}}$ runs, where $\rho_{\mathcal{M}} \geq 2$, to produce a single output run. In the merging stage, the memory-adaptive merge subroutine $\mathcal{M}$ is used repeatedly in the following manner until only a single run remains in $\mathcal{Q}$:

1. Remove the leading $\min\{\rho_{\mathcal{M}}, |\mathcal{Q}|\}$ (pointers to) runs from $\mathcal{Q}$.
2. Merge these runs together using $\mathcal{M}$ in a memory-adaptive manner.
3. Append (the pointer to) the output run to the end of $\mathcal{Q}$.

We have the following estimate of resource consumption over the merging stage as a function of the resource consumption of $\mathcal{M}$.

[4] In order to simplify discussions we neglect details regarding small amounts of additional memory space required by in-memory sorting.

[5] If double buffering is used to overlap I/O and CPU time,then approximately $m/2$ runs are merged together.

Lemma 3 *Suppose that the resource consumption of merge algorithm $\mathcal{M}$ when merging $\rho_{\mathcal{M}}$ runs totally containing n' blocks is bounded by $n' \cdot g(\rho_{\mathcal{M}})$. Suppose that the size of the file to sort is n blocks and that the total number of runs in $\mathcal{Q}$ immediately after the run formation stage is $n_0 \leq \lfloor n/\kappa_{model} \rfloor$. Then the resource consumption of the merging stage is no more than*

$$n \left(\left\lceil \frac{\lg n_0}{\lg \rho_{\mathcal{M}}} \right\rceil + 1 \right) g(\rho_{\mathcal{M}}).$$

Proof: Each time an item x is in one of the runs being merged by $\mathcal{M}$ we say that $\mathcal{M}$ touches item x. By assumption, each item touched by $\mathcal{M}$ is charged a resource consumption of $g(\rho_{\mathcal{M}})/B$ for that execution of $\mathcal{M}$. We prove below that the maximum number of times any element can be touched is $\lceil \lg n_0 / \lg \rho_{\mathcal{M}} \rceil + 1$, thus proving the lemma.

Instead of using the list $\mathcal{Q}$ of runs throughout the merging stage, consider the following modified merging stage using several different queues for the sake of this analysis. Let $\mathcal{Q}_0$ be the list of n_0 runs immediately after the run formation stage. At *merge level* i, we merge runs from queue $\mathcal{Q}_i$ and insert each run output by such a merge into list $\mathcal{Q}_{i+1}$. At the beginning of merge-level i, if the number of runs in $\mathcal{Q}_i$ is no greater than $\rho_{\mathcal{M}}$, we merge all runs in $\mathcal{Q}_i$ to terminate the merging stage. If this is not the case, then while the number of runs in $\mathcal{Q}_i$ is at least $\rho_{\mathcal{M}}$, we repeat the following operation: We remove ρ runs from $\mathcal{Q}_i$, merge them together to form a new run r, and insert run r into list $\mathcal{Q}_{i+1}$. When the number of runs in $\mathcal{Q}_i$ becomes less than $\rho_{\mathcal{M}}$, we append $\mathcal{Q}_i$ to $\mathcal{Q}_{i+1}$.

The maximum number of times any item can be touched during our original merging stage is no more than the number of merge levels in the merging stage described above. Below we bound the total number of merge levels, which we denote by $\#Passes$. Consider an $\rho_{\mathcal{M}}$-ary representation of the number $|\mathcal{Q}_i|$ of runs in $|\mathcal{Q}_i|$ just prior to beginning the ith merge-level, for $0 \leq i \leq \#Passes - 1$. The number of digits in the $\rho_{\mathcal{M}}$-ary representation of $|\mathcal{Q}_i|$ is always one less than the number of digits in the $\rho_{\mathcal{M}}$-ary representation of $|\mathcal{Q}_{i-1}|$, for all except possibly one value of i in the range $1 \leq i \leq \#Passes - 1$. If the number of $\rho_{\mathcal{M}}$-ary digits in the representation of $|\mathcal{Q}_i|$ is the same as that of $|\mathcal{Q}_{i-1}|$, then the most significant digit in the $\rho_{\mathcal{M}}$-ary representation of $|\mathcal{Q}_i|$ is 1 and all other digits are strictly less than $\rho_{\mathcal{M}} - 1$. It follows that $\#Passes \leq \lceil \frac{\lg n_0}{\lg \rho_{\mathcal{M}}} \rceil + 1$. This proves the lemma. □

5.3 Resource consumption of memory-adaptive external mergesort

We will now discuss the requirements that the memory-adaptive merging subroutine $\mathcal{M}$ needs to satisfy in order for our approach to result in a dynamically optimal memory-adaptive sorting algorithm. First we note the simple resource consumption bound of a sorting algorithm based on our paradigm.

Lemma 4 *The resource consumption of the sorting algorithm based on our memory-adaptive mergesort (run formation and merging stage) framework*

is at most

$$2n \lg m_{max} + n \left(\left\lceil \frac{\lg n_0}{\lg \rho_{\mathcal{M}}} \right\rceil + 1 \right) g(\rho_{\mathcal{M}})$$

where n is the number of blocks in the input file, $\mathcal{M}$ and $\rho_{\mathcal{M}}$ are as defined in Section 5.2 and $g(\ )$ and $n_0 \leq \lceil n/\kappa_{model} \rceil$ are as defined in Lemma 3.

If, for a given memory-adaptive subroutine $\mathcal{M}$, $g(\rho_{\mathcal{M}}) = O(\lg \rho_{\mathcal{M}})$, then by Lemma 4, our framework results in a total resource consumption of $O(n \lg n)$ and by Corollary 1 the sorting algorithm is dynamically optimal. On the other hand if $g(\rho_{\mathcal{M}}) = \omega(\lg \rho_{\mathcal{M}})$ then using $\mathcal{M}$ in our framework results in a dynamically nonoptimal sorting algorithm.

Corollary 2 *A memory-adaptive sorting algorithm cast in our mergesort framework is dynamically optimal if and only if $g(\rho_{\mathcal{M}}) = O(\lg \rho_{\mathcal{M}})$.*

Thus as far as our framework is concerned the optimality of the memory-adaptive sorting algorithm depends on the resource consumption of the memory-adaptive merging subroutine $\mathcal{M}$.

Let us consider the quantity $\rho_{\mathcal{M}}$, the number of runs merged by a single application of the memory-adaptive merging subroutine $\mathcal{M}$. If $\rho_{\mathcal{M}}$ is a constant, say 2, then the resource consumption of $\mathcal{M}$ can be as high as $\Omega(n' \lg m_{max}) = \omega(n' \lg \rho_{\mathcal{M}})$ where n' is the total number of blocks involved in the binary merge carried out by $\mathcal{M}$. This high resource consumption is incurred by $\mathcal{M}$ if the allocation sequence consists of allocation phases of m_{max} blocks throughout the duration of $\mathcal{M}$'s execution, which spans $\Omega(n')$ I/O operations. Clearly, the strategy of restricting $\rho_{\mathcal{M}}$ to 2 is in contrast to the strategy (suggested in Section 4.2) of having memory-adaptive algorithms mimic optimal static memory algorithms: When the memory size is m blocks, an optimal static memory mergesort algorithm always merges $\Omega(m^c)$ runs where c is a positive constant. However, as illustrated in Section 7, the task of dynamically reorganizing merge computation in a manner that ensures that the arity of merging computation is proportional to the memory allocation is not a straightforward one.

6 Potential of a Merge

We now discuss the notion of the potential of a merge which applies to any merging algorithm. The potential of a merge at any time is a quantification of the progress made by the merge up to that time.

We first define the association of runs with sets and the notion of the physical sequence of a run at any time during the execution of the merging algorithm.

Definition 6 Let U be a set of runs input to a merging routine $\mathcal{M}$. Every subset u of U has a run r_u associated with it: If u is the singleton set then r_u is the run it contains otherwise r_u is the run output by a merge of the runs contained in u. The *rank* $p(r_u)$ of a run r_u is defined to be the cardinality $|u|$ of u. The *physical sequence* $q(r_u, t)$ corresponding to run r_u at time t is defined as follows:

1. At time $t = 0$, the physical sequence $q(r_u, 0)$ of any input run r_u, with $|u| = 1$, is the entire sequence of elements of r_u taken in order. The physical sequence $q(r_{u'}, 0)$ of run $r_{u'}$ at time $t = 0$, where $u' \subseteq U$ is not a singleton set, is the empty sequence.

2. Suppose that at time t, algorithm $\mathcal{M}$ is in the process of executing a merge operation. Let r_u be the run corresponding to the output run of that merge operation and runs $r_{u_1}, r_{u_2}, \ldots, r_{u_h}$ are the input runs of the merge operation, so we have $u = \bigcup_{1 \le j \le h} u_j$. Suppose that the physical sequences $q(r_u, t)$ and $q(r_{u_j}, t)$, where $1 \le j \le h$, are defined inductively. Then if, at time t, algorithm $\mathcal{M}$ removes the leading item x of the physical sequence of run r_{u_j} and *appends* it to the physical sequence of run r_u, we have $q(r_u, t+1) = q(r_u, t) \cdot x$ and $q(r_{u_j}, t) = x \cdot q(r_{u_j}, t+1)$, where "$\cdot$" denotes concatenation.

We say the formation of the run r_u associated with set u is *logically complete* at time t if no append operation (as in Step 2 above) corresponding to $q(r_u, t'+1) = q(r_u, t') \cdot x$ for some x can be executed at any time $t' \ge t$.

We illustrate this notion by means of an example: Suppose that the merge of runs of the set $S' = \{p_0, p_1, p_2, p_3, p_4\}$ is in progress and by some point t of time, runs p_2 and p_4 have already been depleted by that merge. Then at time t the formation of the run r_s associated with the set $s = \{p_2, p_4\}$ is logically complete: Since all elements of p_2 and p_4 have already been appended into the physical sequence $q(r_{S'}, t)$ of run $r_{S'}$ by time t, no element can ever be appended to the empty physical sequence $q(r_s, t)$ at any time after time t.

Definition 7 The *rank* $p(x, t)$ of an item x at time t is the rank $p(r)$ of the run r such that x lies in the physical sequence $q(r, t)$ of run r at time t. Suppose that $x_1, x_2, \ldots, x_{N'}$ are the $N' = n'B$ items of all the runs in the set U being merged by the merging routine $\mathcal{M}$. Then the *potential* $\Psi(t)$ *of the merge* at time t is defined to be

$$\Psi(t) = \frac{1}{B} \sum_{i=1}^{N'} \lg p(x_i, t).$$

Clearly, at the time $t = 0$ of the beginning of the merge, the potential of the merge is 0. When algorithm $\mathcal{M}$ finishes the merge of runs of set U, the potential of the merge (or rather, the progress made by $\mathcal{M}$) is $n' \lg |U|$, where $n'B$ is the total number of items merged.

The manner in which the potential of a merge changes over an allocation sequence depends on the merging algorithm. We are able to relate the cumulative resource consumption of our algorithm **MAMerge**, presented in Section 8, up to any time t to the potential $\Psi(t)$ of the merge at time t; thus we use potential of the merge to keep track of the resource consumption of **MAMerge**.

7 Nonoptimality of a simple memory-adaptive mergesort algorithm

In this section, we discuss an elegant memory-adaptive mergesort based on the techniques developed by Pang et al. [PCL93]. We prove that its resource consumption is nonoptimal.

The memory-adaptive merging algorithm presented in [PCL93] is not described completely. Below, we present an intuitively appealing algorithm for memory-adaptive merging called $\mathcal{M}^0$ which can reasonably be considered to be an extension of the memory-adaptive merging techniques of [PCL93]. We present two variants of $\mathcal{M}^0$: One variant, which we call the *linear variant*, tries to execute $\Omega(m)$-way merges when the memory allocated is m blocks and the other variant, which we call the *sublinear variant*, tries to execute $\Omega(\sqrt{m})$-way merges when the memory allocated is m blocks. Using an adversarial argument we show that sorting algorithms based on the linear and sublinear variants of $\mathcal{M}^0$ respectively incur a resource consumption of $\Omega(n(\lg n)^2)$ and $\Omega(n(\lg n)(\lg\lg n))$ respectively and so, by Corollary 1, are dynamically nonoptimal.

7.1 Sketch of the memory-adaptive external memory mergesort

The run formation stage remains the one we proposed in Section 5.[6] During the merging stage, a single application of the subroutine $\mathcal{M}^0$ of [PCL93] can be considered to merge at most $\rho_{\mathcal{M}^0} = m_{max} - 1$ runs, where m_{max} is the maximum number of disk blocks that can be fit in physical memory. We will only sketch the memory-adaptive merging algorithm $\mathcal{M}^0$ in this section: Our focus is on aspects of the algorithm that make the resource consumption of $\mathcal{M}^0$ sub-optimal, so we skip several of the details of the algorithm [PCL93].

7.1.1 Memory-adaptive subroutine $\mathcal{M}^0$

We now describe the memory-adaptive merging subroutine $\mathcal{M}^0$ which attempts to mimic an optimal (static) external memory mergesort as described in Section 4.2. The two variants of $\mathcal{M}^0$ correspond to the two mimicking strategies mentioned in Section 4.4 for memory-adaptive mergesort.

Suppose that $\mathcal{M}^0$ tries to execute a $u(m)$-way merge during allocation phases of size m. Then the linear variant of $\mathcal{M}^0$ corresponds to $u(m) = 2^{\lfloor \lg(m-1) \rfloor}$ while the sublinear variant of $\mathcal{M}^0$ corresponds to $u(m) = 2^{2^{\lfloor \lg \lg(m-1) \rfloor}}$. We now define the *level* function $f(\ )$ that maps allocation phases to allocation levels, as explained in Section 4.4.

Definition 8 For the linear variant of $\mathcal{M}^0$, we define $f(m)$ to be $\lfloor \lg(m-1) \rfloor$ and $f^{-1}(\ell)$ to be 2^ℓ. For the sublinear variant of $\mathcal{M}^0$ we define $f(m)$ to be $\lfloor \lg\lg(m-1) \rfloor$ and $f^{-1}(\ell)$ to be 2^{2^ℓ}. (Strictly speaking, the functions $f(\ )$ and

[6] The paper [PCL93] considers quicksort and different variants of replacement-selection in the context of practical issues such as lengths of runs, response-time to fluctuations in memory and disk-locality during the run-formation stage. Our approach can be extended to address most of the practical issues they consider with respect to the run-formation stage.

$f^{-1}(\ )$ are not mathematical inverses of each other.) An allocation phase of size m is said to be at allocation level $f(m)$; or alternatively, during an allocation phase of size m, the allocation level is said to be $f(m)$.

In the context of the memory-adaptive mergesort framework we introduced in Section 5, we use $\mathcal{M}^0$ as a subroutine to memory-adaptively merge $\rho_{\mathcal{M}^0} = f^{-1}(f(m_{max}))$ runs at a time. This means that the linear variant of $\mathcal{M}^0$ has $\rho_{\mathcal{M}^0} = \Theta(m_{max})$ whereas the sublinear variant has $\rho_{\mathcal{M}^0} = \Omega(\sqrt{m_{max}})$.

The basic idea of $\mathcal{M}^0$ is to always associate each allocation level ℓ with a set S_ℓ of runs such that the rank $p(r_{S_\ell})$ of the run r_{S_ℓ} associated with S_ℓ is $f^{-1}(\ell)$. The set S_ℓ of input runs associated with allocation level ℓ may change over the course of the allocation sequence. The computation linked to level ℓ at any time is the merge computation necessary to produce blocks of the run r_{S_ℓ} associated with S_ℓ. Whenever the level of allocation is ℓ, $\mathcal{M}^0$ executes a portion of the computation linked to level ℓ. Whenever the formation of the run r_{S_ℓ} is logically complete, $\mathcal{M}^0$ is assigned a new set of runs such that the condition $p(r_{S_\ell}) = f^{-1}(\ell)$ is satisfied even with the new value of S_ℓ and the process continues.

Member runs of the set S_ℓ may either be *marked* or *unmarked*. The sets S_ℓ for $\ell \in \{1, 2, \ldots, f(m_{max})\}$ are maintained by $\mathcal{M}^0$ over time as follows:

1. Initialization: The set $S_{f(m_{max})}$ is initialized to the $\rho_{\mathcal{M}^0}$ runs that are the runs to be merged. All runs in $S_{f(m_{max})}$ are unmarked. The sets S_ℓ corresponding to all other allocation levels are set to *nil*.

2. During the algorithm, as soon as the merge producing the run r_{S_ℓ} associated with a set S_ℓ becomes logically complete, the set S_ℓ is set to an empty set.

3. During an allocation phase of size m, we check to see if set S_ℓ, where $\ell = f(m)$, is empty on account of points 1 or 2. If set S_ℓ is empty, we execute procedure $load(S_\ell)$ defined in point 4 below. Supposing S_ℓ is non-empty, then during that allocation phase $\mathcal{M}^0$ computes blocks of the run r_{S_ℓ} associated with set S_ℓ.

4. Whenever a set S_ℓ, where $\ell \neq f(m_{max})$, needs to be loaded we execute the procedure $load(S_\ell)$. In this procedure, if set $S_{\ell+1}$ is empty we recursively load it by executing $load(S_{\ell+1})$. Supposing $S_{\ell+1}$ is not empty, if it contains 0 unmarked runs, we unmark all runs of that set and set S_ℓ to $S_{\ell+1}$. On the other hand, if $S_{\ell+1}$ contains unmarked runs, we remove a subset $s \subset S_{\ell+1}$ containing $f^{-1}(\ell)$ unmarked runs[7] from $S_{\ell+1}$, set $S_\ell = s$ and add a marked run r_s that is associated with subset s into set $S_{\ell+1}$.

The biggest drawback of the above scheme for memory-adaptive merging is that merge computation producing blocks of run r_{S_ℓ} associated with set S_ℓ is carried out in allocation phases of size at least $f^{-1}(\ell)$ even if only a 2-way (binary) merge is needed to produce run r_{S_ℓ}, owing to previously executed computations at lower levels: The first time set S_ℓ gets loaded

[7] If set $S_{\ell+1}$ contains even one unmarked run it contains at least $f^{-1}(\ell)$ unmarked runs.

it may have a large number $f^{-1}(\ell)$, of runs to merge. At such times, the merge computation producing blocks of the run r_{S_ℓ} makes very efficient use of m-sized allocation phases, where $\ell = f(m)$. However, in general, allocation levels fluctuate. If the allocation levels remain smaller than ℓ for long enough, it is possible that when the allocation level next becomes ℓ, a binary merge is what is required in order to produce blocks of the run r_{S_ℓ}. The transformation of what was originally an $f^{-1}(\ell)$-way merge into a 2-way merge can take place if the formation of both the runs $r_{S'}$ and $r_{S''}$, where $S_\ell = S' \cup S''$, is logically completed when the allocation level is smaller than ℓ.

Even when the merge required to produce blocks of r_{S_ℓ} is transformed into a binary merge, the algorithm $\mathcal{M}^0$ will persist in using level ℓ allocation phases to execute that binary merge. This is inefficient because huge allocation phases of size m such that $f(m) = \ell$ can end up being used to execute the binary merge: The increase in merge potential registered by $\mathcal{M}^0$ during such an m sized phase is only $O(m)$ whereas the resource consumption is $2m \lg m$. The fact that resource consumption is much greater than the increase in merge potential registered by $\mathcal{M}^0$ during such phases turns out to be a fundamental inefficiency, as will be seen from our analysis below and in the following two sections. This inefficiency is precisely what we exploit to produce a *nemesis* sequence of allocation phases resulting in sub-optimal resource consumption for the algorithm.

We do not mention details regarding data structures used to maintain the sets S_ℓ of runs associated with levels during $\mathcal{M}^0$ since we assume, conservatively, that the sets of runs can be maintained without any cost.

7.2 Lower Bound on Resource Consumption

We will construct a sequence of allocation phases that forces sub-optimal resource consumption for the memory-adaptive sorting algorithm obtained by applying the previous section's memory-adaptive algorithm $\mathcal{M}^0$ to our memory-adaptive mergesort framework. We will express the nemesis sequence construction in terms of $f(m)$ and $f^{-1}(\ell)$ so that it is applicable to both the linear and sublinear variants of the merging scheme.

For convenience, we assume that in case of the linear variant of $\mathcal{M}^0$, $m_{max} = 2^{\ell_{max}} + 1$ whereas in case of the sublinear variant of $\mathcal{M}^0$, we assume that $m_{max} = 2^{2^{\ell_{max}}} + 1$. We use a simple technique to construct our nemesis allocation sequence. We first introduce some terminology that is applicable to the nemesis allocation sequence we construct for the algorithm $\mathcal{M}^0$.

Definition 9 We use $\rho(n', r, m)$ to denote the allocation sequence $m_1, m_2, \ldots, m_f$ satisfying the following conditions:

1. $m_i \leq m$ for $1 \leq i \leq f$.
2. Beginning with the allocation phase m_1, the algorithm $\mathcal{M}^0$ completes the merge of r runs totally consisting of n' blocks of items precisely at the end of the allocation sequence $\rho(n', r, m)$.

We use $\Upsilon(n', r, m)$ to denote the resource consumption corresponding to the allocation sequence $\rho(n', r, m)$.

Below we show how to recursively construct the nemesis sequence $\rho(n', m_{max}-1, m_{max})$. We prove a lower bound on $\Upsilon(n', m_{max}-1, m_{max})$ assuming that all the $m_{max}-1$ runs being merged are of length $n'/(m_{max}-1)$ blocks.

Recursive Formulation

We use the notation $\rho_1 \cdot \rho_2$ to mean the concatenation of the two sequences ρ_1 and ρ_2. We also use the notation ρ_1^p to mean $\rho_1 \cdot \rho_1^{p-1}$. Our construction only uses r's of the form 2^ℓ (respectively 2^{2^k}) in the linear (respectively sublinear) variant. In the definition below, we use $\hat{r}$ to denote $f^{-1}(f(r))$. We have $\hat{r} = r/2$ in the linear variant and $\hat{r} = \sqrt{r}$ in the sublinear variant, for the r's we consider. Our recursive construction of the nemesis allocation sequence $\rho(n', m_{max}-1, m_{max})$ is as follows:

1. Base Case We define

$$\rho(n', 2, m) = m^{n'/m}$$

   Thus $\Upsilon(n', 2, m) = 2n' \log m$, by definition.

2. Recursion We define

$$\rho(n', r, m) = \rho(\frac{n'\hat{r}}{r}, \hat{r}, \hat{r}+1)^{\frac{r}{\hat{r}}} \cdot \rho(n', \frac{r}{\hat{r}}, m)$$

   Thus we have

$$\Upsilon(n', r, m) = \frac{r}{\hat{r}}\Upsilon(\frac{n'\hat{r}}{r}, \hat{r}, \hat{r}+1) + \Upsilon(n', \frac{r}{\hat{r}}, m)$$

It is easy to prove inductively that $\rho(n', r, m)$ constructed as above meets the requirements mentioned in the definition above. We will now prove lower bounds on the resource consumption of the memory-adaptive merging algorithm by solving for $\Upsilon(n', m_{max}-1, m_{max})$.

Lemma 5 *In the linear variant of the memory-adaptive external memory algorithm,*

$$\Upsilon(n', m_{max}-1, m_{max}) = \Omega(n'(\lg m_{max})^2)$$

whereas in case of the sublinear variant,

$$\Upsilon(n', m_{max}-1, m_{max}) = \Omega(n'(\lg m_{max})(\lg\lg m_{max}))$$

Proof: In case of the linear version of the algorithm, we have

$$\Upsilon(n', r, r+1) = 2\Upsilon(n'/2, r/2, r/2+1) + \Upsilon(n', 2, r+1)$$

Using the base case and supposing inductively that $\Upsilon(n'', r', r'+1) \geq n''(\lg(r'+1))^2$ for $n'' \leq n'$, $r' < r$, we have

$$\begin{aligned}
\Upsilon(n', r, r+1) &\geq 2(n'/2)(\lg(r/2+1))^2 + 2n'\lg(r+1) \\
&\geq n'(\lg(r+1)-1)^2 + 2n'\lg(r+1) \\
&\geq n'(\lg(r+1))^2 - 2n'\lg(r+1) + 2n'\lg(r+1) \\
&\geq n'(\lg(r+1))^2
\end{aligned}$$

Thus $\Upsilon(n', m_{max} - 1, m_{max}) \geq n'(\lg m_{max})^2$ in case of the linear variant.

In case of the sublinear variant the recurrence for $\Upsilon(n', r, r+1)$ unfolds as

$$\Upsilon(n', r, r+1) = \sqrt{r}\Upsilon(n'/\sqrt{r}, \sqrt{r}, \sqrt{r}+1) + \Upsilon(n', \sqrt{r}, r+1)$$

Recall that in the sublinear variant we assume that the r's are such that $\lg\lg r$ is integral. We tranform the second and third variables of $\Upsilon(n', r, r'+1)$ by defining the function $\overline{\Upsilon}(n', \lg\lg r, \lg\lg r')$ to be identical to $\Upsilon(n', r, r'+1)$, where $\lg\lg r$ and $\lg\lg r'$ are both integral. Thus $\Upsilon(n', r, r+1) = \overline{\Upsilon}(n', k, k)$, where $k = \lg\lg r$ and we bound $\Upsilon(n', r, r+1)$ as follows.

$$\begin{aligned}\overline{\Upsilon}(n', k, k) &\geq 2^{2^{k-1}}\overline{\Upsilon}(n'/2^{2^{k-1}}, k-1, k-1) + \overline{\Upsilon}(n', k-1, k) \\ &= \sum_{i=1}^{k-1} 2^{2^i}\overline{\Upsilon}(n'/2^{2^i}, i, i) + \overline{\Upsilon}(n', 0, k)\end{aligned}$$

Using the base case and inductively assuming that $\overline{\Upsilon}(n'', k', k') \geq n''k'2^{k'-1}$ for $n'' \leq n'$ and $k' \leq k$, we have

$$\begin{aligned}\overline{\Upsilon}(n', k, k) &\geq n'/2\sum_{i=1}^{k-1} i2^i + 2n' \cdot 2^k \\ &\geq n'k2^{k-1}\end{aligned}$$

Thus we have $\Upsilon(n', m_{max} - 1, m_{max}) \geq n'(\lg\lg m_{max})(\lg m_{max})/2$ with respect to the sublinear variant. □

Using the above lemma in conjunction with Lemma 4, we have the following theorem regarding resource consumption of the memory-adaptive external memory sorting algorithm based on techniques of [PCL93].

Theorem 4 *While sorting a file of n blocks, the memory-adaptive external memory sorting algorithms based on the linear and sublinear variants of the memory-adaptive external memory merging subroutine $\mathcal{M}^0$ have resource consumption of $\Omega(n(\lg n)(\lg m_{max}))$ and $\Omega(n(\lg n)(\lg\lg m_{max}))$ respectively.*

By Theorem 4 and Corollary 1, the above approach to memory-adaptive sorting is dynamically nonoptimal.

8 Dynamically Optimal Memory-Adaptive Sorting

In this section we present a new memory-adaptive merging subroutine **MAMerge** that can be used as $\mathcal{M}$ in the framework of Section 5.2 to obtain a dynamically optimal sorting algorithm.

Throughout this section, we use ρ to denote the number $\rho_{\mathbf{MAMerge}}$ of runs that each application of **MAMerge** merges together. The value of ρ is appropriately chosen to be $\Omega(m_{max})$ except possibly for the final application in which case it can be as small as 2. The novelty of **MAMerge** lies in the data structures and techniques it uses to reorganize the original merge computation adaptively so as to ensure that in "typical" allocation

phases, the resource consumption of **MAMerge** is within a constant factor of the increase in merge potential it registers, and, during all other allocation phases, the total number of I/O operations it incurs is linear in the total number of blocks output by **MAMerge**.

Theorem 5 *Suppose that* **MAMerge** *is used to merge together a set of ρ input runs totally comprising n' blocks. Consider any time t during the execution of* **MAMerge**, *up to and including the time* **MAMerge** *finishes execution. Then the resource consumption of* **MAMerge** *during the allocation sequence up to time t is $O(\Psi(t) + n' \lg m_{max})$, where $\Psi(t)$ is the potential of the merge at time t. In the case $\rho = \Omega(m_{max})$, we have $O(\Psi(t) + n' \lg m_{max}) = O(n' \lg \rho)$.*

The final application of **MAMerge** may merge only a small number $o(m_{max})$ of runs. This application of **MAMerge** may incur a resource consumption of $O(n' \lg m_{max})$ as opposed to $O(n' \lg \rho)$. However, we can show that our sorting algorithm remains dynamically optimal.

A sketch of the above theorem gives a "high level" idea of the technique used by **MAMerge** to tie its resource consumption at any time to the potential of its merge at that time. In order to sketch the proof of Theorem 5, we define "optimal" and "nonoptimal phases".

Definition 10 An allocation phase of size m, where $\kappa_{model} \leq m \leq m_{max}$, in which the potential of the merge being carried out by **MAMerge** increases by an additive amount of $\Theta(m \lg m)$ is an *optimal phase*. Every other allocation phase is said to be a *nonoptimal phase.*

The novel aspects of **MAMerge** are the techniques and data structures it employs to ensure that a typical allocation phase is an optimal phase. In a typical allocation phase of size m, **MAMerge** can efficiently access the physical sequences of $m' = \Omega(\sqrt{m})$ appropriate runs $r_u, r_{u_1}, r_{u_2}, \ldots, r_{u_{m'}}$ such that

1. $u = \bigcup_{1 \leq i \leq m'} u_i$, and
2. For $1 \leq i \leq m'$, we have $p(r_u) = |u| = m'|u_i| = m' p(r_{u_i})$.

Whenever these conditions are satisfied, **MAMerge** can use the $2m$ I/O operations of the phase to append $\Theta(mB)$ new items to the physical sequence of run r_u, where each appended item belongs to the physical sequence of one of the runs $r_{u_1}, r_{u_2}, \ldots, r_{u_{m'}}$. By definition, the increase in the potential of the merge during such an allocation phase is $\Theta(mB \frac{1}{B} \lg(\frac{p(r_u)}{p(r_{u_i})})) = \Theta(m \lg m)$, and thus the phase is optimal. The resource consumption $2m \lg m$ incurred during an optimal phase of size m can be charged to the potential increase $\Theta(m \lg m)$ registered by **MAMerge** during that phase. Since the potential of the merge can never exceed $n' \lg \rho$, the net resource consumption during all optimal phases is no more than $O(n' \lg \rho)$.

On the other hand, the techniques used by **MAMerge** also ensure that the total number of I/O operations obtained by summing the I/O operations over all nonoptimal phases is $O(n')$. Since the maximum resource consumption of an I/O operation is $\lg m_{max}$, the resource consumption during all the nonoptimal phases of **MAMerge** remains $O(n' \lg m_{max})$. This concludes the sketch of the proof of Theorem 5.

8.1 Overview

Each application of **MAMerge**, except possibly the final application, merges together $\rho = 2^{2^{\lceil \lg \lg(m_{max}/\kappa_{level}) \rceil}}$ runs from $\mathcal{Q}$ as described in Section 5.2, where κ_{level} is a constant to be determined later. **MAMerge** partitions the set of possible sizes of allocation phases into "allocation levels".

Definition 11 An allocation phase of size s is said to be at allocation level $level(s) = \lceil \lg \lg(s/\kappa_{level}) \rceil$; or alternatively, during allocation phases of size s, the (ongoing) allocation level is said to be $level(s)$. In our scheme, we require that the integral constant κ_{model} defined in Definition 1 be large enough for $level(\kappa_{model})$ to be 1. We use ${\ell_{max}}'$ to denote the integer $level(m_{max})$.

By definition, each allocation phase is at one of the allocation levels ℓ, where $\ell \in \{1, 2, \ldots, {\ell_{max}}'\}$.

The basic strategy employed by **MAMerge** is to dynamically maintain an association of a merge operation "appropriate for level ℓ" with each allocation level ℓ and to generate $\Theta(m)$ blocks output by that merge operation during an allocation phase of size m at level ℓ. [8] In the case when the formation of the output run of that merge operation is logically completed before $\Theta(m)$ blocks can be output, **MAMerge** ends up generating less than $\Theta(m)$ blocks during that phase. Whenever the formation of the output run of the merge operation associated with level ℓ gets logically completed, **MAMerge** has to reorganize the global merge computation so as to find a new merge operation "appropriate for level ℓ".

Definition 12 A merge operation is said to be *appropriate for level* ℓ if the rank $p(x,t)$ of an element x appended at time t to the physical sequence of the output run of that merge operation is such that

$$p(x,t) \geq 2^{2^{\ell-1}} \cdot p(x,t-1).$$

Put another way, if x lies in the physical sequence of run r' at time $t-1$ and in the physical sequence of the output run r of the merge operation at time t, then we have $p(r) \geq 2^{2^{\ell-1}} \cdot p(r')$.

Every allocation phase in which **MAMerge** outputs $\Theta(m)$ blocks of a merge operation appropriate for level ℓ is an optimal phase, by definition. Allocation phases spent by **MAMerge** in reorganizing the global merge so as to find merge operations for levels not currently associated with appropriate merges may be nonoptimal. An allocation phase of size m at level ℓ can also be nonoptimal if the formation of the output run of the merge operation appropriate for level ℓ gets completed during that phase. Another source of possibly nonoptimal phases are phases at any allocation level $\ell > maxlevel$, where *maxlevel* is a special variable maintained by **MAMerge**. The number of nonoptimal phases is small enough that the number of I/O operations summed over all nonoptimal phases is $O(n')$, where n' is the number of blocks output by **MAMerge**.

[8] An exceptional case is when $\ell > maxlevel$, where *maxlevel* is a variable maintained by **MAMerge** as described below.

In Section 8.2 we present the recursively defined data structure of a "run-record" that plays a central role in the manner in which **MAMerge** dynamically reorganizes its merge computation. In Section 8.3, we descibe the preprocessing stage of **MAMerge** and some other preliminaries of **MAMerge**. In Section 8.4, we present a data structure called level-record that stores, for each level ℓ, the merge operation appropriate for level ℓ. In Section 8.5, we mention the invariants pertaining to run-records and level-records that **MAMerge** maintains. In Section 8.6, we describe the simple procedure executed by **MAMerge** during a phase at allocation level ℓ when there does exist an appropriate merge operation for level ℓ. In Section 8.7, we present the **download**( ) operation used to find new appropriate merge operations for levels that are currently not associated with any merge operation. In Section 8.8, we sew together our data structures and techniques to obtain the memory-adaptive merging algorithm **MAMerge**. We analyze **MAMerge**'s resource consumption in Section 9.

8.2 Run-Records

We associate a "run-record", defined below, with every run formed in course of our mergesort algorithm. Each run-record contains a pointer to the start and end of its run's physical sequence on disk. The queue $\mathcal{Q}$ of runs defined in Section 5.2 is, in fact, implemented as a queue of run-records.

At any time the "state" of **MAMerge** consists of a set of merge operations that can collectively be viewed as an adaptive re-organization of the original ρ-way merge that **MAMerge** sets out to compute. Linked implicitly to each such merge operation in the state of **MAMerge** is a *run-record* defined below: Consider, for example, merging a set P of runs $r_0, r_1, \ldots, r_{p-1}$, where $p = |P|$, into the run r and suppose that $p = 2^{2^\ell}$ for a non-negative integer ℓ. In our scheme, we ensure that the number of runs in all merge operations in **MAMerge**'s state is always of the form 2^{2^x}, where x is a non-negative integer. Algorithm **MAMerge** maintains a run-record rr associated with the output run r. Run-record rr contains pointers to the leading and trailing disk blocks of run r, if any, and to a list of the run-records rr_i associated with the runs $r_i \in P$. Thus whenever we wish to work on the merge operation whose output is run r, we can do so by using the run-records rr_i to get pointers to blocks of the runs r_i. We append the merge output to the trailing disk block of run r pointed to by rr. In general, the runs $r_i \in P$ can themselves be runs that correspond to the outputs of some other merge operations. More importantly, the design of algorithm **MAMerge** easily handles situations in which the formation of a run $r_i \in P$ may not be logically complete, in the sense of Definition 6. Algorithm **MAMerge** has the flexibility of implementing the p-way merge linked to rr by recursively splitting it into $\sqrt{p}$-way merge operations: The run-record rr stores a pointer to a list of $\sqrt{p}$ run-records $rr'_0, rr'_1, \ldots, rr'_{\sqrt{p}-1}$ associated with the runs $r'_0, r'_1, \ldots, r'_{\sqrt{p}-1}$ such that run r is logically the output of the $\sqrt{p}$-way merge of runs $r'_0, r'_1, \ldots, r'_{\sqrt{p}-1}$, and each run r'_i is logically the output of the $\sqrt{p}$-way merge of the runs $r_{i\sqrt{p}}, r_{i\sqrt{p}+1}, \ldots, r_{(i+1)\sqrt{p}-1}$.

Below we give a precise definition of the fields that form a run-record. It is useful to separate the logical notion of a run from the way it may actually

exist on disk at any time.

Definition 13 In our scheme, the physical sequence $q(r,t)$ at any time t of a run r is stored in a blocked manner on disk.

Before defining the recursive run-record data structure, we define the various fields of a run-record.

Definition 14 The fields of a run-record associated with a run r' are as follows:

1. *begin*: At any time t, the *begin* field points to the leading element of the physical sequence $q(r',t)$, assuming $q(r',t)$ is non-empty, on disk.

2. *end*: At any time t, the *end* field points to the trailing element of the physical sequence $q(r',t)$, assuming $q(r',t)$ is non-empty, on disk.

3. *Order*: An integer field.

4. *inputs*: The *inputs* field points to the disk location of the leading run-record of a list of *Order* run-records stored in a blocked manner on disk. The *inputs* field implicitly represents this blocked list of *Order* run-records so we sometimes refer to the pointer *inputs* as a list of run-records. This list of run-records may actually exist as a sub-list of a larger blocked list of run-records on disk.

5. *flag*: The *flag* field records whether or not the formation of run r' is logically complete, as in Definition 6. Accordingly, *flag* is set to *Done* or *NotDone* respectively. The *flag* field of the run-record associated with any run input to **MAMerge** is initialized to *Done*.

6. *splitters*: The *splitters* field points to the disk location of the leading run-record of a list of $\sqrt{Order}$ run-records stored in a blocked manner on disk. The *splitters* pointer implicitly represents the blocked list of $\sqrt{Order}$ run-records.

Each run-record occupies only $O(1)$ amount of space, which is proportional to $O(1/B)$ disk blocks.

Definition 15 If run r is one of the ρ input runs of **MAMerge**, then in the run-record rr associated with r, we have $rr.Order = 1$ and $rr.inputs = rr.splitters = nil$. On the other hand suppose that run r is logically the run corresponding to the output of the merge of the runs r_i, where $0 \leq i \leq p-1$ and $p = 2^{2^\ell}$ for a non-negative integer ℓ, and rr_i is the recursively defined run-record associated with run r_i. Then given a run-record rr at any time, we say

$$rr = \bigvee \{rr_0, rr_1, \ldots, rr_{p-1}\}$$

if and only if the following conditions are satisfied:

1. $rr.Order = p$.

2. The list $rr.inputs$ contains precisely the p run-records $rr_0, rr_1, \ldots, rr_{p-1}$.

3. The list $rr.splitters$ contains $\sqrt{p}$ run-records rr'_j, where $0 \leq j \leq \sqrt{p}-1$, such that $rr'_j = \bigvee\{rr_{j\sqrt{p}}, rr_{j\sqrt{p}+1}, \ldots, rr_{(j+1)\sqrt{p}-1}\}$

We say that the computation associated with run-record rr is the computation involved in merging the runs associated with the $rr.Order$ run-records in $rr.inputs$ to produce blocks appended to the physical sequence of run r associated with rr.

Although the definition of a run-record is recursive, we do not employ recursion to construct a run-record rr such that $rr = \bigvee\{rr_0, rr_1, \ldots, rr_{p-1}\}$, given the run-records rr_i.

Definition 16 Given a blocked list L_p of $p = 2^{2^\ell}$ run-records $rr_0, rr_1, \ldots, rr_{p-1}$, where ℓ is a non-negative integer, the construction of a run-record rr satisfying the condition $rr = \bigvee\{rr_j | 0 \leq j \leq p-1\}$ is called a **construct**(rr, L_p) operation.

The **construct**(rr, L_p) operation can be implemented by successively constructing a sequence of ℓ blocked lists L^i of run-records, for $0 \leq i \leq \ell - 1$ and then setting $rr.inputs$ to be the list L_p and $rr.splitters$ to be the list $L^{\ell-1}$. In general, the blocked lists L^i are constructed so as to satisfy the the following conditions:

1. List L^i is a blocked list containing $p/2^{2^i}$ run-records, which we denote $rr(i,j)$, where $0 \leq j \leq p/2^{2^i} - 1$.

2. The jth run-record $rr(i,j)$ in list L^i, where $0 \leq i \leq \ell - 1$ and $0 \leq j \leq p/2^{2^i} - 1$, satisfies $rr(i,j).Order = 2^{2^i}$ and the list $rr(i,j).inputs$ consists of the 2^{2^i} run-records $rr_{jx}, rr_{jx+1}, \ldots, rr_{(j+1)x-1}$, of list L_p, where $x = 2^{2^i}$.

3. Consider a run-record $rr(i,j)$, where $1 \leq i \leq \ell - 1$ and $0 \leq j \leq p/2^{2^i} - 1$, from any list other than list L^0. Let x denote 2^{2^i}. Then, the $\sqrt{x} = 2^{2^{i-1}}$ run-records in the list $rr(i,j).splitters$ are precisely the run-records $rr(i-1, j\sqrt{x})$, $rr(i-1, j\sqrt{x}+1)$, ..., $rr(j+1)\sqrt{x} - 1$, of list L^{i-1}. Each run-record in list L^0 has its *splitters* field set to *nil*.

4. The fields $rr.inputs$ and $rr.splitters$ are set so that $rr.inputs$ represents list L_p and $rr.splitters$ represents list $L^{\ell-1}$.

5. The *flag* fields of all run-records of the lists $L^0, L^1, \ldots, L^{\ell-1}$ and the run-record rr are said to *NotDone*; the *begin* and *end* fields of these run-records are set to *nil* indicating that their respective physical sequences at that time are all empty.

It can be inductively shown that the above conditions imply for $0 \leq i \leq \ell - 1$ and $0 \leq j \leq x - 1$ that

$$rr(i,j) = \bigvee\{rr_{jx}, rr_{jx+1}, \ldots, rr_{(j+1)x-1}\}$$

where $x = 2^{2^i}$. This means that $rr = \bigvee\{rr_j \ | 0 \leq k \leq p-1\}$, as desired.

If the lists L^i are constructed in ascending order of i, then all the run-records of the blocked list L^i can be constructed in a single traversal of

list L^{i-1} for $i > 0$ and list L_p for $i = 0$: For $i > 0$, Step 2 above is implemented by setting $rr(i,j).inputs$ to be equal to the (value of) the field $rr(i-1, j2^{2^{i-1}}).inputs$ and $rr(i,j).splitters$ to store the disk location of the run-record $rr(i-1, j2^{2^{i-1}})$. The following lemma follows from the fact that the total number of run-records summed over all the lists L^i and list L_p is $O(p)$ and that constructing each list L^i requires no more than $O(1)$ blocks of internal memory.

Lemma 6 *The total number of I/O operations incurred by a* **construct**(rr, L_p) *operation, where L_p is a blocked list of p run-records, is $O(p/B + \lg\lg p) = O(p)$. The total number of memory blocks required to implement* **construct**(rr, L_p) *is $O(1)$.*

Another useful observation is expressed by the following lemma.

Lemma 7 *The total number of new run-records created by* **construct**(rr, L_p)*, where L_p contains p run-records is $O(p)$.*

During the preprocessing stage of **MAMerge**, we execute a **construct**$(rr, L_{p'})$ operation in which the number p' of run-records in $L_{p'}$ is not of the form 2^{2^ℓ} but rather of the form $w2^{2^\ell}$, where w is a non-negative integer less than 2^{2^ℓ}. In this case the above procedure can be modified so that the number of run-records in list $L^{\ell-1}$ is w, instead of $\sqrt{p'}$. However, the condition that $rr = \bigvee\{rr' | rr' \in L_{p'}\}$ remains satisfied.

Corollary 3 *The total number of I/O operations incurred by the* **construct**$(rr, L_{p'})$ *operation described in the above paragraph, in which $L_{p'}$ is a blocked list of $p' = w2^{2^\ell}$ run-records and w is a positive integer less than 2^{2^ℓ}, is no more than $O(p'/B + \lg\lg p')$, which is $O(p')$. The total number of memory blocks required in the implementation is $O(1)$.*

8.3 Preprocessing

Having defined run-records and the **construct**( ) operation, we now describe the preprocessing carried out by **MAMerge** before it starts merging. First we introduce some terminology that we will be using throughout the next two sections.

Definition 17 The number of runs merged by an application of **MAMerge** is given by

$$\rho = \min\{2^{2^{\lceil \lg\lg(m_{max}/\kappa_{level})\rceil}}, |\mathcal{Q}|\},$$

where $|\mathcal{Q}|$ is the number of runs in list $\mathcal{Q}$ at the beginning of that application of **MAMerge**. We use ℓ_{max} to denote the integer $\lceil \lg\lg\rho \rceil$, and $\overline{\rho}$ to denote the integer $2^{2^{\ell_{max}}}$.

The value of ρ for each application of **MAMerge**, in our framework is such that, except perhaps in the final application of **MAMerge**, we have $\ell_{max} = \ell_{max}{}' = \lceil \lg\lg(m_{max}/\kappa_{level}) \rceil$ and $\rho = \overline{\rho}$.

During the final application of **MAMerge**, it is possible that $\rho < \overline{\rho}$, in which case, it is convenient to introduce some dummy[9] run-records to

[9] We define the run associated with a dummy run-record in Section 9.

the list of run-records corresponding to runs being merged. Adding dummy run-records enables us to make the convenient assumption that that the number of run-records in the list $rr'.inputs$ of any run-record rr' is always of the form 2^{2^ℓ} for integral ℓ. If $\rho < \overline{\rho}$, one possibility is to add $\overline{\rho} - \rho$ dummy run-records, but this can be extremely inefficient since $\overline{\rho}$ can be as high as $\Theta(\rho^2)$. So we first add enough dummy run-records to obtain a total of $w\overline{\rho}^{1/2}$ run-records, where w is an integer no larger than $\overline{\rho}^{1/2}$. Then after running the modified version of the **construct**( ) operation alluded to in Corollary 3, we add some more dummy run-records appropriately to ensure that, after preprocessing, the number of run-records in the list $rr'.inputs$ of any run-record rr' is of the form 2^{2^ℓ} while maintaining the condition that the number of dummy run-records added is $O(\rho)$.

We execute the following steps during the preprocessing stage of **MAMerge** in our framework.

1. Copy the ρ leading run-records from $|\mathcal{Q}|$ into a new blocked list L of run-records and remove these run-records from $\mathcal{Q}$.

2. Append at most $\overline{\rho}^{1/2} - 1$ dummy run-records to the end of L so that the number of run-records in L is $w\overline{\rho}^{1/2}$ for a non-negative integer w.

3. Execute a modified version of the **construct**(rr', L) operation (alluded to in Corollary 3) on the blocked list L containing $w\overline{\rho}^{1/2}$ run-records. The blocked list $rr'.splitters$, denoted L', contains w run-records.

4. Discard run-record rr' and append less than $\overline{\rho}^{1/2}$ dummy run-records to L' so that L' now contains w' run-records, where w' is the smallest integer such that $w' \geq w$ and $w' = 2^{2^x} \leq \overline{\rho}^{1/2}$ for an integral x.

5. Execute a **construct**(rr, L') operation on the blocked list L' containing w' run-records.

The run-record rr contructed in Step 5 has $rr.inputs$ pointing to a blocked list of w' run-records rr^j, where $0 \leq j \leq w' - 1$. With one possible exception, the blocked list $rr^j.inputs$ of each non-dummy run-record rr^j of list $rr.inputs$ contains a unique set of $\overline{\rho}^{1/2}$ run-records from the set of ρ run-records associated with runs **MAMerge** sets out to merge; one non-dummy run-record of list $rr.inputs$ may possibly contain the dummy run-records introduced in Step 2. The union of the sets represented by the lists $rr^j.inputs$ includes the ρ run-records corresponding to runs input to **MAMerge** and thus, the run associated run-record rr is logically the run corresponding to a merge of the ρ input runs.

Lemma 6 and Corollary 3 above imply the following lemma.

Lemma 8 *The total number of I/O operations incurred during the preprocessing stage is no more than $O(\rho/B + \ell_{max})$, which is $O(\rho)$. The total number of memory blocks required to implement the preprocessing is $O(1)$. The total number of new run-records created during the preprocessing stage, including dummy run-records is $O(\overline{\rho}^{1/2}) = O(\rho)$.*

The following definitions refer to variables maintained and used by **MAMerge**.

Definition 18 The algorithm **MAMerge** maintains a variable rr_{global} storing a special run-record. At the end of Step 5 above, we initialize rr_{global} to the run-record rr resulting from the execution of the **construct**(rr, L') operation. We use W_{top} to denote the number of non-dummy run-records in the list $rr_{global}.inputs$ immediately after Step 5 of the preprocessing completes.

Throughout **MAMerge**'s execution, the variable rr_{global} stores the run-record whose run is logically the merge of all the ρ runs input to **MAMerge**.

8.4 Level-record Data Structure

We now describe the "level-record" data structure that associates with each level ℓ, where $1 \leq \ell \leq \ell_{max}$, a merge operation appropriate for level ℓ. A level-record stores a pointer to a run-record together with some supplementary information.

Definition 19 Every allocation level ℓ, where $1 \leq \ell \leq \ell_{max}$, is associated with its *level-record*, denoted $lr[\ell]$. Level-record $lr[\ell]$ is either *nil* or it comprises of the following three fields:

1. *rr*: The *rr* field stores the location of a run-record which can be viewed as a repository of computation that may carried out at level ℓ.

2. *current*: The *current* field stores a non-negative integer.

3. *active*: The *active* field is a pointer to a run-record which we call the *active run-record* of level ℓ and denote by rr_ℓ, in short. The merge operation associated with run-record rr_ℓ is always a merge operation appropriate for level ℓ.

Level-records $lr[1]$ through $lr[\ell_{max}]$ are stored in a blocked list so they occupy $O(\ell_{max}/B+1)$ disk blocks in total. By "computation associated with level ℓ" we refer to the merge operation associated with run-record rr_ℓ.

Initialization of level-records

Immediately on completion of the preprocessing of **MAMerge**, all level-records except $lr[\ell_{max}]$ are initialized to *nil*. Level record $lr[\ell_{max}]$ is initialized as follows: Its *rr* field is set to run-record rr_{global}, its *current* field is set to 0 and its *active* field is set to the same value as $rr_{global}.inputs$ (which points to the first run-record in the blocked list implicitly represented by $rr_{global}.inputs$.)

The following definition refers to a variable maintained and used by **MAMerge**.

Definition 20 Throughout its operation, **MAMerge** maintains a special variable *maxlevel* initialized to ℓ_{max}.

The value of *maxlevel* is always such that $lr[maxlevel].rr = rr_{global}$.

8.5 Invariants for run-records and level-records

We now present the invariants pertaining to run-records and level-records maintained by **MAMerge**. The invariants hold immediately after preprocessing is completed and after each **llmerge**( ) and **download**( ) operation during **MAMerge**'s execution.

In the invariants specified below, we use the following variables and shortened names.

Definition 21 By ℓ, we denote an integer such that $1 \leq \ell \leq \ell_{max}$. We use *current* to denote the field $lr[\ell].current$ of level-record $lr[\ell]$, rr to denote $lr[\ell].rr$, m to denote $lr[\ell].rr.Order$, and *active* to denote $lr[\ell].active$. By rr^i, where $0 \leq i \leq m-1$, we denote the ith run-record in the list $rr.inputs$ of m run-records.

In case of the invariants below that refer to level-record $lr[\ell]$, it is obviously assumed that $lr[\ell]$ is not *nil*.

Invariants

1. $1 \leq maxlevel \leq \ell_{max}$.
2. The run associated with run-record rr_{global} is always the run corresponding to the output of the merge of ρ runs input to **MAMerge**.
3. If $maxlevel < \ell \leq \ell_{max}$, then level-record $lr[\ell] = nil$.
4. $lr[maxlevel].rr$ stores the location of run-record rr_{global}.
5. $rr.flag$ is set to *NotDone*.
6. We have $0 \leq current \leq m$. For the case $\ell = \ell_{max}$, $current \notin \{W_{top}, W_{top}+1, \ldots, m-1\}$.
7. If $current < m$ then the run-record rr_ℓ pointed to by *active* is the *current*-th run-record $rr^{current}$ in the list $rr.inputs$, otherwise run-record rr_ℓ is the run-record rr itself.
8. If $\ell = maxlevel$, then $m = 2^{2^x}$, where x is an integer no larger than $\ell - 1$.
9. If $\ell < maxlevel$, then $m = 2^{2^{\ell-1}}$.
10. Each one of the $m - current$ run-records rr^i, where $current \leq i \leq m-1$, has $rr^i.flag$ set to *NotDone* and $rr^i.Order$ set to $2^{2^{\ell-1}}$. In the specific case of $\ell = maxlevel = \ell_{max}$, this invariant holds for the non-dummy run-records rr^i, where $current \leq i \leq W_{top} - 1$ of the list $rr.inputs$.
11. Consider the ordered list S_ℓ of run-records obtained as follows: First concatenate together the lists $rr^i.inputs$ of the run-records rr^i, where $current \leq i \leq m-1$, in increasing order of i to obtain the list S'. Then list S_ℓ is obtained by appending the ordered list $rr^0, rr^1, \ldots, rr^{current-1}$ of run-records to the tail of list S'. Let the elements, in order, of list S_ℓ be $s_0, s_1, \ldots, s_{|S_\ell|}$, where $|S_\ell|$ denotes the number of run-records in S_ℓ.

(a) At most $\ell - 1$ run-records of S_ℓ either have their *flag* field set to *NotDone*.

(b) Consider a run-record $s \in S_\ell$, with $s.flag = NotDone$. Then there is precisely one level ℓ', where $0 < \ell' < \ell$, such that $lr[\ell'].rr$ points to s.

(c) Consider any pair s_{i_1} and s_{i_2} of elements of list S_ℓ such that $s_{i_1}.flag = s_{i_2}.flag = NotDone$ and $i_1 < i_2$ with $lr[\ell_1].rr$ pointing to s_{i_1} and $lr[\ell_2].rr$ pointing to s_{i_2}. Then $\ell_1 < \ell_2$.

We present a couple of useful observations as lemmas based on the above invariants, using the same notation.

Lemma 9 *Unless $\ell = maxlevel$ and $current = m$, given level-record $lr[\ell]$, we can obtain a run-record rr' using a single I/O operation, where rr' is such that $rr'.flag = NotDone$ and the merge operation associated with rr' is appropriate for level ℓ.*

Proof: The invariants 5, 6, 7, 10 and 9 together imply that the active run-record rr_ℓ of level ℓ has $rr_\ell.Order = 2^{2^{\ell-1}}$, unless $\ell = maxlevel$ and $current = m$. By definition of run-records and Definition 12, the merge operation affiliated with rr_ℓ is appropriate for level ℓ, unless $\ell = maxlevel$ and $current = m$. Since level-record $lr[\ell].active$ is a pointer to rr_ℓ, the lemma is true. □

When the allocation level is ℓ, **MAMerge** carries out computation producing blocks belonging to the run associated with the active run-record rr_ℓ of level ℓ by merging the runs associated with the run-records in the list $rr_\ell.inputs$: Lemma 9 ensures us that this amounts to making optimal utilization of resources. However, as will be seen later, it is possible for a run-record rr' in the list $rr_\ell.inputs$ to have its *flag* field set to *NotDone*, meaning that the formation of the run associated with rr' is not logically complete: This is a potential problem since it means that in order to produce blocks of the run associated with rr_ℓ, **MAMerge** would inherently have to also carry out the merge linked to each such run-record rr' and so producing blocks of the run associated to rr_ℓ may require **MAMerge** to have more memory than originally expected. However the invariants 11a and 11b avert this potential problem. Invariant 11a implies that there are at most $\ell - 1$ run-records in list $rr_\ell.inputs$ with their *flag* field set to *NotDone* and invariant 11b implies that the total number of extra run-records necessitated by these run-records is no more than $O(2^{2^{\ell-1}})$. This number is within a constant factor of the number $2^{2^{\ell-1}}$ of run-records originally expected to be involved in the merge producing blocks of the run associated with rr_ℓ, thus averting the potential problem.

It is easy to prove that the above invariants are all true immediately after the preprocessing computation and initialization of level-records is completed.

8.6 Low-level Merge Computation

We now describe the procedure **llmerge**(rr) used by **MAMerge** to carry out the merge computation affiliated to run-record rr. Whenever the allo-

cation level is $\ell \leq maxlevel$, and level-record $lr[\ell]$ is not *nil*, **MAMerge** executes the procedure **llmerge**(rr_ℓ) described below, producing blocks of the run associated with the active run-record rr_ℓ of level ℓ. Lemma 9 ensures us that rr_ℓ is affiliated to a merge operation appropriate for level ℓ and so level ℓ allocation phases are optimal, unless possibly when $\ell = maxlevel$ and $lr[\ell].current = lr[\ell].rr.Order$.

Whenever the allocation level ℓ is greater than $maxlevel$ or when $\ell = maxlevel$ and $lr[\ell].current = 2^{2^{\ell-1}}$, **MAMerge** executes the procedure **llmerge**(rr_{global}): By invariants 4 and 8, such a phase can be nonoptimal even if $\Theta(mB)$ elements are appended to the physical sequence of the run associated with rr_{global} during that phase. However, we argue in Section 9 that the total amount of resource consumption over all executions of **llmerge**(rr_{global}) is $O(n'' \lg m_{max})$, where n'' is the sum of the number of disk blocks of all the ρ runs merged by **MAMerge**.

8.6.1 Invariants for llmerge( )

In our description of the procedure **llmerge**( ), we use rr to denote the run-record passed as an argument to **llmerge**( ). We ensure that the following invariants are always satisfied whenever **MAMerge** executes procedure **llmerge**(rr) and the allocation level is ℓ:

1. $rr.flag = NotDone$.
2. If $\ell \leq maxlevel$, then
   (a) $rr = rr_\ell$, the active run-record of level ℓ.
   (b) During the level ℓ allocation phase at the instant when **MAMerge** makes the call to execute procedure **llmerge**(rr), the following condition is satisfied: Either the number *left* of I/O operations in the ongoing allocation phase is such that $left \geq \kappa_{llm} \cdot 2^{2^{\ell-1}}$, where κ_{llm} is a positive constant defined below, or, the immediately following allocation phase[10] of size *next* is at level $level(next) = \ell$.
   (c) The level-records $lr[1]$ through $lr[\ell]$ are already in memory at the time the call to execute **llmerge**(rr) is made.
3. If $\ell > maxlevel$, then
   (a) $rr = rr_{global}$.
   (b) During the level ℓ allocation phase, at the instant when **MAMerge** makes the call to execute procedure **llmerge**(rr), the following condition is satisfied: Either the number *left* of I/O operations in the ongoing allocation phase is such that $left \geq \kappa_{llm} \cdot 2^{2^{maxlevel}}$, where κ_{llm} is a positive constant defined below, or the immediately following allocation phase of size *next* is at allocation level $level(next) > maxlevel$.

[10] As mentioned before, it is possible to modify our strategy to make do without the information corresponding to *next* with no loss of efficiency, asymptotically.

(c) The level-records $lr[1]$ through $lr[maxlevel]$ are already in memory at the time the call to execute **llmerge**(rr) is made.

4. When the execution of **llmerge**(rr) is completed, the *flag* field of run-record rr is set to *Done* if and only if the formation of the run associated with rr is logically complete, as in Definition 6.

The following definition is used during our discussion on **llmerge**( ).

Definition 22 We define $\overline{k}$ to be the quantity $\min\{\ell, maxlevel + 1\}$ and $\overline{m}$ to be the quantity $2^{2^{\overline{k}-1}}$.

The execution of procedure **llmerge**(rr), producing blocks of the run associated to run-record rr, is split into two parts, called the *merging part* and the *state-saving* part respectively. During the merging part, **llmerge**(rr) performs I/O related to the merging process whereas, during the state-saving part, **llmerge**(rr) performs I/O in which relevant data structures, updated so as to maintain invariants (of Section 8.5), and partially full buffers of runs are committed to disk before the allocation level changes. Invariants 2b and 3b above ensure that **llmerge**(rr) is executed only when the ongoing allocation level is guaranteed to last for a number of I/Os large enough to accomodate both the merging part and the state-saving part, as the invariants 2b and 3b above indicates.

During the merging part, **llmerge**(rr) loads into memory the run-records in the list $rr.inputs$ and then starts merging the physical sequences of the runs associated with these run-records, appending the merge output to the physical sequence of the run associated with rr. If the physical sequence of any run associated with a run-record rr' of $rr.inputs$ becomes empty during the merge and $rr'.flag = NotDone$, **llmerge**(rr) now has to include the physical sequences of the runs associated with run-records in the list $rr'.inputs$ in the merge operation. In order to do this, **llmerge**(rr) first has to loads run-records of the list $rr'.inputs$ into memory. Similar steps result if some other run-record has a *flag* field value of *NotDone* when the physical sequence of the associated run becomes empty. The algorithm executed during the merging part of **llmerge**(rr) is therefore as follows:

1. The set T is initialized to contain all run-records in $rr.inputs$.

2. Ensure that each run-record of T is allocated one internal memory block to buffer the leading block of the physical sequence of the run it is associated to.

3. While the physical sequence of the run associated with every run-record in set T is non-empty, merge the physical sequences corresponding to run-records of T into the physical sequence of the run associated with rr.

4. If the physical sequence of the run associated to run-record $rr' \in T$ becomes empty, then $T = T - \{rr'\}$. If $rr'.flag = NotDone$, then add all run-records of list $rr'.inputs$ to T. Go to Step 2.

Definition 23 Any run-record that becomes an element of set T at some point during the merging part is said to have been *touched* by **llmerge**(rr). Let $y(\overline{k})$ and $z(\overline{k})$ respectively denote the minimum and maximum number of run-records touched during the merging part of **llmerge**(rr).

The number of I/Os required to load data blocks and data structures necessary to begin the merging part of **llmerge**(rr) is proportional to the number of run-records it touches. So we first obtain a handle on this quantity by means of the following two lemmas.

Lemma 10 *The maximum number $z(\overline{k})$ of run-records touched during the execution of* **llmerge**(rr) *is no more than $O(2^{2^{\overline{k}-1}})$.*

Proof: We first consider the case when $\overline{k} = \ell \leq maxlevel$ and $rr = rr_\ell$. In this case, by invariants 8 and 10 of Section 8.5, we know that the number of run-records in list $rr.inputs$ is $2^{2^{\overline{k}-1}}$ and all these are touched. Now we will consider run-records other than those in list $rr.inputs$ that get touched. For a given touched run-record to cause more run-records to be touched, that run-record necessarily must have a *flag* field value of *NotDone*.

We say that a run-record rr' belongs to level ℓ' if rr' is either in the *inputs* list of the run-record $lr[\ell'].rr$ or in the *inputs* list of some run-record in the *inputs* list of the run-record $lr[\ell'].rr$.

As a result of invariant 11 of Section 8.5 and the fact that any run-record not included in list $rr.inputs$ is touched only if it lies in a depleted run-record's *inputs* list, each run-record touched by **llmerge**(rr) must belong to a level $\ell' < \overline{k} = \ell$. By invariants 10 and 9 of Section 8.5, the maximum number of run-records belonging to to level ℓ' is $2^{2^{\ell'}}$. Hence the maximum number $z(\overline{k})$ of run-records touched by **llmerge**(rr) when $\overline{k} = \ell$ is

$$2^{2^{\overline{k}-1}} + \sum_{\ell'=1}^{\overline{k}-1} 2^{2^{\ell'}} = O(2^{2^{\overline{k}-1}})$$

In the case when $\overline{k} = maxlevel + 1$ and rr is rr_{global}, every touched run-record belongs to a level $\ell' < \overline{k}$ so the maximum number $z(\overline{k})$ of touched run-records is no more than

$$\sum_{\ell'=1}^{\overline{k}-1} 2^{2^{\ell'}} = O(2^{2^{\overline{k}-1}})$$

Hence the lemma is proved. □

The following lemma can be easily proved.

Lemma 11 *If the run-record rr is not the run-record rr_{global}, then we have $y(\overline{k}) \geq 2^{2^{\overline{k}-1}}$, where $y(\overline{k})$ is the smallest possible number of run-records that* **llmerge**(rr) *touches.*

If the runs being merged during **llmerge**(rr) are long enough, the merging part of **llmerge**(rr) can proceed for an indefinitely long time, unless we *preempt* **llmerge**(rr). The need to preempt stems from the fact that if the allocation level changes to $\ell' \leq maxlevel$, **MAMerge** would then execute

llmerge($rr_{\ell'}$), where $rr_{\ell'}$ is the active run-record of level ℓ'. In order to be able to resume a merge operation at some later stage, during the state-saving part of **llmerge**(rr) we commit partially empty buffer blocks of physical sequences being merged, updated touched run-records and pertinent updated level-records back to disk: Transferring these to disk obviously requires I/O operations so **llmerge**(rr) reserves a certain number of I/O operations from its total number, $\kappa_{llm} \cdot 2^{2^{\overline{k}-1}}$, of I/O operations, specifically for this purpose. We now describe the state-saving part of **llmerge**(rr) and determine how many I/O operations it requires.

During its state-saving part, **llmerge**(rr) executes the following steps:

1. The *flag* field of any run-record rr' such that rr' is either rr or a touched run-record, is set to *Done* if it was previously *NotDone* and the formation of the run associated with rr' is logically complete, as in Definition 6. (By virtue of invariant 4 of Section 8.6.1, it is enough to set $rr'.flag = Done$ whenever, after recursively determining the values of the *flag* fields of all the run-records in list $rr'.inputs$, it is found that they all have their *flag* fields set to *Done*.[11])

2. If a touched run-record whose *flag* field changes value from *NotDone* to *Done* is the run-record whose location is stored in $lr[\ell'].rr$ or $lr[\ell'].active$, for some ℓ' such that $1 \leq \ell' \leq \min\{\ell, maxlevel\}$, we need to accordingly update the level-record $lr[\ell']$ in order to ensure that the invariants of Section 8.5 remain true: It is not hard to see that these invariants can be maintained easily for all relevant ℓ's.

3. Write out to disk the internal memory blocks that buffer the physical sequences being merged. Write out all touched run-records and the run-record rr back to disk. Write out the level-records $lr[1]$ through $lr[\min\{\ell, maxlevel\}]$ back to disk.

The state-saving part basically ensures that the following lemma is true.

Lemma 12 *The invariants of Section 8.5 and the invariant 4 of Section 8.6.1 remains true after the execution of* **llmerge**(rr) *is completed.*

By Lemma 10, since **llmerge**(rr) has at most one internal memory block corresponding to each touched run-record, we have the following lemma.

Lemma 13 *There exists a constant κ_{save} such that the total number of I/O operations required to implement* **llmerge**(rr)*'s state-saving part, in which touched run-records, pertinent level-records and partially filled blocks of physical sequences are written to disk, is no more than $\kappa_{save} \cdot 2^{2^{\overline{k}-1}}$.*

In order to complete the description of **llmerge**(rr), we need to define the constant κ_{llm} used in invariants 2b and 3b. As mentioned above $\kappa_{llm} \cdot 2^{2^{\overline{k}-1}}$ should take into account the $\kappa_{save} \cdot 2^{2^{\overline{k}-1}}$ required to complete the state-saving part of **llmerge**(rr). Additionally, κ_{llm} should also be large enough for a useful amount of "work" to get done during the merging part of **llmerge**(rr). Below we quantify the notion of useful amount of work.

[11] Dummy run-records introduced in the preprocessing stage are all to be treated as run-records with their *flag* fields set to *Done*.

Definition 24 Consider the physical sequence $q(r,t)$ of the run r associated with run-record rr at time t just before the execution of **llmerge**(rr) begins. We call a particular execution of **llmerge**(rr) a *good call* if at least $2^{2^{\overline{k}-1}} \cdot B$ items are appended to the physical sequence $q(r,t)$ during that execution of **llmerge**(rr). Any execution of **llmerge**(rr) that is not a good call is a *bad call.*

We consider the work involved in appending $2^{2^{\overline{k}-1}} \cdot B$ items to the physical sequence of the run associated with **llmerge**(rr) a useful amount of work during the merging part of **llmerge**(rr). The following lemma bounds the total number of I/O operations incurred in carrying out this useful amount of work.

Lemma 14 *Let $q_{\max}$ be the total number of items that need to be appended to the physical sequence of the run r associated with run-record rr for the formation of r to be logically complete. Then the total number of I/O operations incurred by the the merging part of* **llmerge**(rr) *to carry out enough merging computation to append $\min\{2^{2^{\overline{k}-1}} \cdot B, q_{\max}\}$ items to the physical sequence of run r is no more than $\kappa_{load} \cdot 2^{2^{\overline{k}-1}}$, where κ_{load} is a small positive constant.*

Sketch of Proof: Suppose that the merging technique used during the merging part is the "standard" external memory $|T|$-way merge technique, where T is the set defined in the description of the merging part. The lemma then follows from the fact that there are at most $O(2^{2^{\overline{k}-1}})$ touched run-records during **llmerge**(rr) and after incurring $O(1)$ I/O operations corresponding to each touched run-record as "start-up overhead", the merging process results in 1 block of items being appended to the physical sequence of run r every $O(1)$ I/O operations. □

The execution of **llmerge**(rr) requires one internal memory block to buffer the physical sequence corresponding to each touched run-record and $O(1/B)$ internal memory blocks to store each touched run-record or each level-record loaded by **llmerge**(rr). Thus by Lemma 14, we have the following lemma.

Lemma 15 *The total number of internal memory blocks required by* **llmerge**(rr) *is no more than $\frac{\kappa_{load}}{2} \cdot 2^{2^{\overline{k}-1}}$.*

We now define the constant κ_{llm} used in invariants 2b and 3b.

Definition 25 We define the constant κ_{llm} to be $\kappa_{load} + \kappa_{save}$, where κ_{load} and κ_{save} are respectively defined in Lemma 13 and Lemma 14. We define the constant κ'_{llm} to be $\kappa_{llm} + \delta$ such that for $1 \leq \ell \leq \ell_{max}$, the total number of I/O operations required to load the level-records $lr[1]$ through $lr[\ell]$ is no more than $\delta \cdot 2^{2^{\ell-1}}$ and δ is as small as possible.

The definition of κ'_{llm} is for minor technical reasons. Suppose that the invariants mentioned earlier are satisfied at the time **MAMerge** makes the call to execute **llmerge**(rr) at allocation level ℓ. Then the complete description of **llmerge**(rr), based on the merging and state-saving parts, described above is as follows.

1. Execute the merging part until a time t such that at least one of the following conditions is violated:

   (a) If $\overline{k} = \ell < \mathit{maxlevel} + 1$, then either the number *left* of I/O operations in the ongoing allocation phase is such that $\mathit{left} \geq \kappa_{save} \cdot 2^{2^{\ell-1}}$ or the immediately following allocation phase of size *next* is at level $\mathit{level}(\mathit{next}) = \ell$. If $\overline{k} = \mathit{maxlevel} + 1$, then either the number *left* of I/O operations in the ongoing allocation phase is such that $\mathit{left} \geq \kappa_{save} \cdot 2^{2^{\overline{k}-1}}$ or the immediately following allocation phase of size *next* is at level $\mathit{level}(\mathit{next}) > \mathit{maxlevel}$.

   (b) The formation of the run associated with run-record rr is not logically complete at time t.

2. Execute the state-saving part described above.

The merging part can actually extend for many I/O operations more than $O(2^{2^{\overline{k}-1}})$, so long as the conditions in step 1a and 1b above are satisfied. Thus the number of blocks appended to the physical sequence of the run associated with rr can also exceed $O(2^{2^{\overline{k}-1}})$. The only situation that causes the execution of **llmerge**(rr) to be a bad call is when the condition in step 1b is violated during the execution of **llmerge**(rr): Due to invariants 2b and 3b of Section 8.6.1 and the definition of κ_{llm}, the execution of **llmerge**(rr) can never end up being a bad call on account of violation condition 1a during the execution of **llmerge**(rr). We have the following lemmas regarding good and bad **llmerge**(rr) calls relating the number of I/O operations incurred by **llmerge**(rr) and the number of blocks appended to the physical sequence of the run associated with run-record rr.

Lemma 16 *Suppose that the execution of* **llmerge**(rr) *is a good call in which g items are appended to the physical sequence of the run associated with rr. Then the total number of I/O operations incurred by the execution of* **llmerge**(rr) *is* $O(g/B)$.

Sketch of Proof: After the merging process has loaded into memory the leading B items of the runs being merged, roughly speaking, each time a block is brought into memory there is a corresponding block being written to the output run. The total number of touched run-records is $O(2^{2^{\overline{k}-1}})$. The total number of I/O operations required to append $2^{2^{\overline{k}-1}} \cdot B$ items to the concerned physical sequence is $O(2^{2^{\overline{k}-1}})$, by Lemma 14. Since the **llmerge**(rr) in question is a good call, we have $g \geq 2^{2^{\overline{k}-1}} \cdot B$. Hence the lemma follows. □

The following lemma follows directly from Lemma 14.

Lemma 17 *Suppose that the execution of* **llmerge**(rr) *is a bad call. Then the total number of I/O operations incurred by the execution of* **llmerge**(rr) *is* $O(2^{2^{\overline{k}-1}})$

In order to test that the condition of step 1b above remains true, at any time during its execution, **llmerge**(rr) needs to maintain the invariant that

the *flag* field of rr is *Done* if and only if the formation of the run associated with rr is logically complete: By virtue of Step 1 of the state-saving part and the fact that all touched run-records reside in internal memory at the time the formation of the run associated with rr does become logically complete, this invariant can be achieved easily by **llmerge**(rr).

8.7 Downloading Work for Adaptivity

If $lr[\ell]$ is not *nil*, whenever the allocation level is ℓ, **MAMerge** executes the merging operation affiliated with rr_ℓ. If **MAMerge**'s allocation level remains ℓ or smaller for extended periods of time, **MAMerge** runs out of all the $\Omega(2^{2^{\ell-1}})$-way merging work associated with $lr[\ell]$ and level-record $lr[\ell]$ becomes *nil*. When the level-record $lr[\ell]$ becomes *nil*, we need to associate level ℓ with "fresh" appropriate merging work. We refer to the process of establishing such an association a "downloading" process since it requires "stealing" some of the merging work associated with a higher allocation level $\ell' > \ell$. Once the new association of level ℓ with such a merge operation is made, the run-record rr_ℓ is once more affiliated to a merge operation which is appropriate for level ℓ, and which can be executed when the allocation level next becomes ℓ.

8.7.1 Loading Work Down One Level

We now consider what exactly constitutes downloading work from level $\ell+1$ to level ℓ, assuming $lr[\ell]$ is *nil* but $lr[\ell+1]$ is not. If $m = 2^{2^\ell}$, downloading work from level $\ell+1$ to level ℓ involves reorganizing the m-way merge affiliated with the active run-record $rr_{\ell+1}$ of level $\ell+1$ into a $\sqrt{m}$-way merge of $\sqrt{m}$-way runs each the output of a $\sqrt{m}$-way merge. We use the procedure **construct**( ) in order to bring about such a reorganization.

Assuming that level-record $lr[\ell+1]$ is not *nil* and level-record $lr[\ell]$ is *nil*, we use the procedure **loadlevel**(ℓ), described below, to download the active run-record $rr_{\ell+1}$ from $lr[\ell+1]$ to $lr[\ell]$, appropriately updating $lr[\ell+1]$ in the process.

1. Let rr_{new} be a new run-record.
2. If $\ell + 1 = maxlevel$ and $lr[\ell+1].current = lr[\ell+1].rr.Order < 2^{2^\ell}$, then
   (a) Set rr_{new} to be the run-record $lr[\ell+1].rr$.
3. Else
   (a) Let $m = 2^{2^\ell}$ and let $L_{\sqrt{m}}$ be the blocked list containing $\sqrt{m}$ run-records, implicitly represented by the field $rr_{\ell+1}.splitters$ of the active run-record $rr_{\ell+1}$ of level $\ell+1$.
   (b) Discard the run-record $rr_{\ell+1}$, execute **construct**($rr_{new}, L_{\sqrt{m}}$) to appropriately initialize run-record rr_{new} and let rr_{new} now take the place of the old run-record $rr_{\ell+1}$ on disk.

4. Set $lr[\ell].rr$ to store the disk location of run-record rr_{new}, set $lr[\ell].current = 0$ and set $lr[\ell].active = rr_{new}.inputs$ so it points to the first run-record of the list represented by $rr_{new}.inputs$.

5. If $lr[\ell+1].current \leq lr[\ell+1].rr.Order - 1$ then

   (a) If $\ell + 1 = \ell_{max}$ and $lr[\ell+1].current = W_{top} - 1$, then

      i. Set $lr[\ell+1].current$ to the value $lr[\ell+1].rr.Order$.

      ii. Set $lr[\ell+1].active$ to store the location of run-record $lr[\ell+1].rr$.

   (b) Otherwise, increment the value of $lr[\ell+1].current$ by 1. Then, if $lr[\ell+1].current < lr[\ell+1].rr.Order - 1$, set $lr[\ell+1].active$ to store the location of the $lr[\ell+1].current$-th run-record of list $lr[\ell+1].rr.inputs$; else set $lr[\ell+1].active$ to store the location of run-record $lr[\ell+1].rr$.

6. Else (that is, $lr[\ell+1].current = lr[\ell+1].rr.Order$)

   (a) Set $lr[\ell+1]$ to *nil*.

   (b) If $maxlevel = \ell + 1$, set $maxlevel$ to ℓ and set rr_{global} to be the run-record pointed by $lr[maxlevel].rr$.

The following lemma follows from Lemma 9, Lemma 6 and the observation that the list involved in the **construct**( ) operation of Step 3b contains $\sqrt{m}$ run-records.

Lemma 18 *Suppose that $m = 2^{2^\ell}$, $lr[\ell] = nil$ and $lr[\ell+1] \neq nil$, where $1 \leq \ell < maxlevel$, at some time during our algorithm. Then the execution of procedure* **loadlevel**(ℓ) *described above takes no more than $O(\sqrt{m})$ I/O operations and can be implemented using $O(1)$ blocks of internal memory.*

The significance of the upper bound on internal memory above is that procedure **loadlevel**(ℓ) can be executed over $O(\sqrt{m}/B + 1)$ I/O operations possibly spanning several arbitrary allocation phases, since the constant κ_{model} will be set to a value larger than the number of blocks required in **loadlevel**(ℓ) for any ℓ.

In **MAMerge**, the only time new run-records are created is during **loadlevel**( ) operations. Using Lemma 7 and the fact that the list $L_{\sqrt{m}}$ passed as argument to **construct**( ) during **loadlevel**( ) contains $\sqrt{m}$ run-records, we have the following lemma.

Lemma 19 *Suppose that $m = 2^{2^\ell}$, $lr[\ell] = nil$ and $lr[\ell+1] \neq nil$, where $1 \leq \ell < maxlevel$, at some time during our algorithm. Then the total number of new run-records created during* **loadlevel**(ℓ) *is no more than $O(\sqrt{m})$.*

We also make the crucial observation that **loadlevel**(ℓ) maintains the invariants proposed in Section 8.5.

Lemma 20 *The invariants of Section 8.5 remain true after the execution of* **loadlevel**(ℓ).

Proof: Since $lr[\ell] = nil$ and since $lr[\ell+1]$ satisfies invariant 11a prior to **loadlevel**(ℓ), the list $S_{\ell+1}$ defined in invariant 11 contains at most $\ell - 1$ run-records with *flag* fields set to *NotDone*.

First we will prove that $lr[\ell]$ satisfies all the invariants after the execution of **loadlevel**(ℓ). Consider the active run-record $rr_{\ell+1}$ of level $\ell+1$ just before **loadlevel**(ℓ) is executed. In the case when $\ell+1 = maxlevel$ and $lr[\ell+1].current = lr[\ell+1].rr.Order < 2^{2^\ell}$, $rr_{\ell+1}$ happens to be the run-record pointed by $lr[\ell+1].rr$. Invariant 8 implies that $rr_{\ell+1}.Order = 2^{2^x}$ for an integer x such that $x \leq \ell - 1$. Thus, the assignment making $lr[\ell].rr$ store the location of the run-record $rr_{\ell+1}$, the other assignments in Step 4, and the fact that *maxlevel* decreases to ℓ, ensure that $lr[\ell]$ satisfies the invariants after the execution of **loadlevel**(ℓ). When any one of the conditions $\ell+1 = maxlevel$ and $lr[\ell+1].current = lr[\ell+1].rr.Order < 2^{2^\ell}$ are not satisfied, the run-record $rr_{\ell+1}$ necessarily has $rr_{\ell+1}.Order = 2^{2^\ell}$. In Step 3b, we replace this run-record on disk with an appropriately constructed new run-record rr_{new} with $rr_{new}.Order = 2^{2^{\ell-1}}$. Thus, the assignment making $lr[\ell].rr$ store the location of this newly constructed run-record and the other assignments in Step 4 ensure that $lr[\ell]$ satisfies the invariants after the execution of **loadlevel**(ℓ).

Now we will prove that $lr[\ell+1]$ satisfies the invariants after the execution of **loadlevel**(ℓ). If $lr[\ell+1].current$ was $lr[\ell+1].Order$ prior to **loadlevel**(ℓ), then $lr[\ell+1]$ becomes *nil* trivially satisfying all invariants. Step 5 ensures that invariants 6 and 7 remain true even after **loadlevel**(ℓ). In the case when $lr[\ell+1]$ is not *nil* after the execution of **loadlevel**( ), we now consider the parts of invariant 11. After the execution of **loadlevel**(ℓ), the set of run-records in list $S_{\ell+1}$ has at most one run-record with *flag* set to *NotDone* *more* than just before **loadlevel**(ℓ). But since the maximum number of run-records in $S_{\ell+1}$ prior to **loadlevel**(ℓ) is $\ell - 1$, invariant 11a remains true even after **loadlevel**(ℓ). The invariant 11b remains true of $S_{\ell+1}$ even after **loadlevel**(ℓ) because the only change among level-records of levels lower than $\ell+1$ is that $lr[\ell].rr$ now stores the location of the run-record rr_{new} which lies in the list $lr[\ell+1].inputs$. The invariant 11c remains true of ordered list $S_{\ell+1}$ after **loadlevel**(ℓ) since the last element of $S_{\ell+1}$ is the run-record pointed by $lr[\ell].rr$, which is at the highest level lower than level $\ell+1$.

In case *maxlevel* decreases during **loadlevel**( ), the run associated to the run-record pointed by $lr[maxlevel].rr$ is logically the same as the one associated with the run-record pointed by the $lr[maxlevel].rr$ with the old value of *maxlevel*, so invariant 4 remains true.

It can be easily verified that the other invariants remain true after the execution of **loadlevel**(ℓ) as well. Thus the lemma is proved. □

8.7.2 Loading Work Down Several Levels

Consider the problem of downloading work to level k when several levels $k, k+1, \cdots, \ell$ have level-records set to *nil* and only the level-record $lr[\ell+1]$ of level $\ell+1$ is not *nil*. In such a situation, **MAMerge** has to download work to level k from level $\ell+1$ and it does so using the procedure **download**(k) described below.

1. Traverse the (blocked) list of level-records beginning with $lr[k+1]$ until

a level-record $lr[\ell+1]$ such that $lr[\ell+1] \neq nil$ is found.

2. Execute **loadlevel**(i) for i going from ℓ down to k.

Thus, during the execution of **download**(k), **MAMerge** ends up downloading work to the other levels $k+1, \cdots, \ell$ that had their level-records set to *nil* as well.

Definition 26 We say that the execution of **download**(k) *loads* levels k through ℓ, where ℓ is as defined above. The highest level *loaded* by **download**(k) is level ℓ.

Using Lemma 18 and the invariants of Section 8.5, we have the following lemma regarding resource consumption during the execution of **download**(k).

Lemma 21 *Suppose $m = 2^{2^\ell}$ where ℓ is the highest level that is loaded by* **download**(k). *The total number of I/O operations involved in first traversing the list of level-records $lr[1]$ through $lr[k]$ and then executing* **download**(k) *is $O(\sqrt{m})$. The* **download**(k) *operation can be implemented such that it requires no more than $O(1)$ internal memory blocks.*

Proof: The total number of I/O operations required to traverse the list $lr[1]$ through $lr[k]$ of pointers before executing **download**(k) and then traversing level-records $lr[k]$ through $lr[\ell+1]$ during the initial part of **download**(k) requires no more than $O((\lg\lg m)/B+1)$ I/O operations. Using Lemma 18 and the invariants of Section 8.5, the $\ell-k+1$ calls to **loadlevel**( ) totally incur

$$O(\sum_{i=0}^{\ell-k}\sqrt{m^{1/2^i}}) = O(\sum_{i=0}^{\ell-k}(\sqrt{m})^{1/2^i}) = O(\sqrt{m})$$

I/O operations. Thus the total number of I/O operations is $O(\sqrt{m})$. As in the case of **loadlevel**( ) we require only $O(1)$ memory blocks to complete the operation. □

In the above lemma we also counted the I/O operations required to load level-record $lr[k]$ into memory assuming we have to follow the blocked list of pointers beginning with $lr[1]$ to do so.

The following lemma follows from Lemma 21 and is used later, among other things, in setting the constant κ_{level} to an appropriate value.

Lemma 22 *Consider a sequence of d operations such that:*

1. *In the ith operation, where $0 \leq i \leq d-1$, we traverse the blocked linked list of level-records to access $lr[\ell_i]$ and then execute* **download**(ℓ_i).

2. *Over the entire sequence of d operations, for no level ℓ', where $1 \leq \ell' \leq \ell_{max}-1$, is* **loadlevel**$(\ell')$ *executed more than once. Moreover, let ℓ be the highest level such that* **loadlevel**(ℓ) *is executed during the sequence of d operations and let $m = 2^{2^\ell}$.*

Then there exists a small positive constant κ_{dl} such that the number of I/O operations over the entire sequence of d actions is no more than $\kappa_{dl}\sqrt{m}$.

Using Lemma 19, we can easily prove the following lemma.

Lemma 23 *The total number of new run-records during the sequence of* d **download**( ) *operations described in Lemma 22 is no more than* $O(\sqrt{m})$.

We also make the following simple observation regarding the invariants of Section 8.5.

Lemma 24 *The invariants of Section 8.5 remain true after the execution of* **download**(k).

Proof: During **download**(k), run-records and level-records are manipulated only during calls to **loadlevel**( ). The lemma here follows from Lemma 20 and the fact that any **download**( ) execution calls **loadlevel**(ℓ') only when $lr[\ell']$ is *nil* and $lr[\ell'+1] \neq nil$. □

8.8 Putting download( ) and llmerge( ) Together

We describe how to combine the **download**( ) procedure, used to dynamically reorganize merging computation, and the **llmerge**( ) procedure, used to merge physical sequences in appropriate merge operations, together with the run-record and level-record data structures to realize the memory-adaptive merging routine **MAMerge** that merges together ρ runs. By using **MAMerge** in our framework of Section 5.2 as described at the beginning of Section 8, we obtain a memory-adaptive mergesort.

In order to complete the description of our memory-adaptive sorting algorithm we need to define the constant κ_{level} used in our definition of allocation levels, and the constant κ_{model} in the context of our memory-adaptive mergesort.

Definition 27 The constant κ_{level} is defined to be $\kappa_{level} = (\kappa'_{llm} + \kappa_{dl})/2$. The constant κ_{model} is defined to be $\kappa_{model} = \lceil 2\kappa_{level} \rceil$.

If m is of the form 2^{2^ℓ}, where $1 \leq \ell \leq \ell_{max}$, then the number $2\kappa_{level}\sqrt{m}$ of I/O operations in a level ℓ allocation phase of size $\kappa_{level}\sqrt{m}$ is large enough to first accomodate $\kappa_{dl}\sqrt{m}$ I/O operations (corresponding to the sequence of **download**( ) operations in Lemma 22) "loading" level ℓ and then permit the smallest number $\kappa_{llm}m$ of I/O operations involved in a good **llmerge**(rr) call, where $rr = rr_\ell$, the active run-record of level ℓ. The smallest size κ_{model} of an allocation phase is such as to permit a binary merge during a phase of that size.

Next we define some useful abbreviations used in our description of **MAMerge**.

Definition 28 We use the symbol *clevel* to mean the current allocation level $level(mem)$. If $\ell \leq maxlevel$, we define the predicate $enough(\ell)$ to be true whenever the invariant 2b of Section 8.6.1 is satisfied and false otherwise. If $\ell > maxlevel$, we define the predicate $enough(\ell)$ to be true whenever the invariant 3b of Section 8.6.1 is satisfied and false otherwise.

Algorithm MAMerge

The algorithm **MAMerge** can be summarized as follows:

1. If the allocation level is $\ell \leq maxlevel$ and there is work associated with level ℓ (meaning $lr[\ell] \neq nil$), then execute **llmerge**(rr). If $\ell > maxlevel$, execute **llmerge**(rr_{global}).

2. If the allocation level is ℓ and $lr[\ell] = nil$, then download some work to level ℓ by executing **download**(ℓ).

A more precise description is as follows:

1. Execute the preprocessing stage of Section 8.3 and then the initialization of level-records as indicated in Section 8.4.

2. Until $rr_{global}.flag \neq Done$, do:

   (a) Let $\hat{\ell} = \min\{clevel, maxlevel\}$.

   (b) Load level-records in a blocked manner beginning with $lr[1]$ until $lr[\hat{\ell}]$ into memory. .

   (c) If $clevel > maxlevel$ then

      i. Execute **llmerge**(rr_{global}).

      ii. When the call **llmerge**(rr_{global}) returns control, if $rr_{global}.flag = Done$, then **MAMerge** is completed.

      iii. If the call **llmerge**(rr_{global}) returns control at a time when the allocation level $level(next)$ of the following phase is $maxlevel$ or smaller, then reliniquish the remaining portion of the current phase, whose allocation level must necessarily be greater than $maxlevel$. Otherwise proceed immediately to the following step

      iv. GoTo Step 2a.

   (d) (Invariant: $clevel \leq maxlevel$.)

   (e) If ($lr[clevel] = nil$), then

      i. If $clevel = maxlevel$, then **MAMerge** is completed; otherwise execute **download**($clevel$).

      ii. When the call **download**( ) returns control, the level of allocation $clevel$ may have changed since the time the call is made. If $enough(clevel)$ is false for the new value of $clevel$, then relinquish the remaining portion of the allocation phase ongoing when **download**( ) returns. If $enough(clevel)$ is true, then proceed immediately to the following step.

      iii. GoTo Step 2a.

   (f) (Invariant: $lr[clevel] \neq nil$ and $clevel \leq maxlevel$.)

   (g) Else

      i. While $enough(clevel)$AND$lr[clevel] \neq NIL$, execute **llmerge**(rr_{clevel}), where rr_{clevel} is the active run-record of level $clevel$.

ii. When the last call **llmerge**(rr_{clevel}) of the while loop returns, if $enough(clevel)$ is false, then relinquish the remaining portion of the ongoing allocation phase; otherwise proceed immediately to the following step.

iii. GoTo Step 2a.

8.8.1 Relinquished I/O Operations

Since invariant 2b (respectively invariant 3b) has to be met at the time the call to **llmerge**(rr_ℓ) (respectively **llmerge**(rr_{global})) is made, **MAMerge** sometimes relinquishes a portion of an allocation phase. We count the relinquished I/O operations among I/O operations incurred by **MAMerge** by using the following charging scheme.

Definition 29 Whenever **MAMerge** relinquishes part of an allocation immediately after executing a **download**( ) operation, we charge the relinquished I/O operations to that particular **download**( ) operation. Whenever **MAMerge** relinquishes part of an allocation immediately after executing an **llmerge**( ) operation, we charge the relinquished I/O operations to that particular **llmerge**( ) operation.

First we account for I/O operations charged to **download**( ) operations by using the notion of ℓ, d-sequences, which are sequences of d consecutive **download**( ) operations, possibly followed by a relinquish operation.

Definition 30 Consider a sequence of d consecutive **download**(ℓ_i) operations, where $0 \leq i \leq d-1$ and $d \geq 1$, that satisfy the conditions mentioned in Lemma 22. and let ℓ be as defined in Lemma 22. We call the above sequence of d **download**( ) operations an ℓ, d-sequence if the first time **MAMerge** either relinquishes I/O operations (in Step 2(e)ii above) or executes an **llmerge**( ) operation after it executes **download**(ℓ_0) is only immediately after it executes **download**(ℓ_{d-1}).

We make a crucial observation regarding ℓ, d-sequences.

Lemma 25 *Consider any ℓ, d-sequence* **download**(ℓ_0), ..., **download**(ℓ_{d-1}) *and let $m = 2^{2^\ell}$. If, at any time after* **download**(ℓ_0) *begins and before* **download**(ℓ_{d-1}) *ends,* **MAMerge** *is subjected to an allocation phase of size m_h such that $level(m_h) \geq \ell$, then the execution of* **download**(ℓ_{d-1}) *is immediately followed (that is, without relinquishing any I/Os) by the execution of* **llmerge**(rr)*, where $\ell_h = level(m_h)$ and rr is the active run-record rr_{ℓ_h} of level ℓ_h if $\ell_h \leq maxlevel$ and the run-record rr_{global} otherwise.*

Proof: From Lemma 22, we know that the total number of I/O operations required over the entire ℓ, d-sequence is no more than $\kappa_{dl}\sqrt{m}$. Suppose that after **download**(ℓ_0) begins execution and before **download**(ℓ_{d-1}) completes execution, **MAMultiply** gets an allocation phase of size m_h where m_h is as defined above. Then even if all $\kappa_{dl}\sqrt{m}$ I/O operations corresponding to the ℓ, d-sequence occurred during the phase of size m_h, that phase is still left with $\kappa_{llm} \cdot \sqrt{m_h}$ pending I/O operations that would cause $enough(level(m_h))$ to evaluate to true: This follows from the definition of κ_{level} above. Furthermore, by definition of ℓ, d-sequence, the dth **download**( ) operation

download(ℓ_{d-1}) of the ℓ, d-sequence cannot be immediately followed by another **download**( ) operation so **llmerge**(rr) is the next operation executed by **MAMerge**. Thus the lemma is proved. □

If I/O operations corresponding to a portion of an allocation are relinquished by **MAMerge** after an ℓ, d-sequence, then the level of the allocation is necessarily smaller than ℓ, by virtue of the above lemms. This also means that the maximum number of I/O operations relinquished by **MAMerge**, that can be charged to an ℓ, d-sequence is no more than $O(2^{2^{\ell-1}})$.

We state now a simple consequence of the above lemma, Lemma 22 and definition 29 to bound the number of I/O operations that can be charged to an ℓ, d-sequence.

Lemma 26 *The number of I/O operations that can be charged to an ℓ, d-sequence is no more than $O(\sqrt{m})$, where $m = 2^{2^\ell}$.*

When **MAMerge** relinquishes I/O operations of a portion of a level ℓ allocation phase immediately after executing an **llmerge**( ) call (Step 2(g)ii or Step 2(c)iii), the number of I/O operations charged to that **llmerge**( ) call remains within a constant factor of the I/O operations incurred by that **llmerge**( ) call, as per Lemma 16 or Lemma 17.

Lemma 27 *The total number of I/O operations charged to an* **llmerge**( ) *operation, including I/Os relinquished by* **MAMerge**, *is given by Lemma 16 if* **llmerge**( ) *is a good call and Lemma 17 if it is a bad call.*

9 Analysis of resource consumption

In this section we show that in merging together ρ runs, totally consisting of n' blocks, our MA algorithm consumes only $O(n' \lg \rho + n' \lg m_{max})$ resources and results in an optimal sorting algorithm.

We already showed that the total number of I/O operations incurred while preprocessing is no more than $O(\rho)$ I/O operations, which means a resource consumption of no more than $O(\rho \lg m_{max})$. Next we show that the resource consumption incurred by the downloading activity, which makes **MAMerge** memory-adaptive, is only $O(\rho \lg m_{max})$. Then by charging the I/O operations incurred by a bad **llmerge**( ) call to the run-records touched by that call, we argue that the total number of I/O operations incurred by bad **llmerge**( ) calls throughout the execution of **MAMerge** is no more than $O(\rho)$. Then we argue that the total number of I/O operations incurred by **llmerge**(rr_{global}) calls is no more than $O(n')$. Finally, we employ the notion of merge potential to show that **MAMerge** makes optimal utilization of available resources during good **llmerge**( ) calls. We do so by showing that the resource consumption of each good **llmerge**( ) call is always within a constant factor of the increase in merge potential it registers. A corollary of the above fact is that the net resource consumption charged to all good **llmerge**( ) calls cannot exceed $O(n' \lg \rho)$.

9.1 Resource Consumption of download( ) calls, bad llmerge( ) calls and llmerge(rr_{global}) calls

By definition, allocation phases utilized by **MAMerge** for the execution of **download**( ) calls, bad **llmerge**( ) calls, and **llmerge**(rr_{global}) calls, can be nonoptimal phases. Here we show that the net resource consumption over all such activity during **MAMerge** is no more than $O(n' \lg m_{max})$, where n' is the total number of blocks in the the ρ runs being merged by **MAMerge**.

9.1.1 Resource consumption during download( ) computation

As we noted earlier, **download**( ) computation is memory-oblivious: Since it never requires more than κ_{model} blocks it can be carried out during allocation phase(s) at arbitrary levels. Every ℓ, d-sequence involves a **loadlevel**(ℓ) operation. We bound the resource consumption of all the **download**( ) operations that occur in course of the algorithm by bounding the maximum possible number of **loadlevel**(ℓ) operations in course of the algorithm, for $1 \leq \ell \leq \ell_{max} - 1$, and then using Lemma 26.

First we make an observation that bounds the maximum number of times level ℓ can be involved in a **loadlevel**($\ell - 1$) operation at a stretch, without the occurrence of a **loadlevel**(ℓ) operation.

Lemma 28 *Suppose that $1 \leq \ell < \ell_{max}$. Then, if $\ell+1 < \ell_{max}$, the maximum number of* **loadlevel**(ℓ) *operations the level-record $lr[\ell+1]$ can be involved in before it becomes nil is $lr[\ell+1].rr.Order + 1$ if $\ell + 1 < \ell_{max}$; if $\ell + 1 = \ell_{max}$, the maximum number of* **loadlevel**(ℓ) *operations the level-record $lr[\ell + 1]$ can be involved in before it becomes nil is $W_{top} + 1$.*

Since each ℓ, d-sequence necessarily includes a **loadlevel**(ℓ) operation, we have the following lemma.

Lemma 29 *Consider any level $\ell_{max} - j$, where $1 \leq j \leq \ell_{max} - 1$. The total number of $\ell_{max} - j, d$-sequences possible in course of* **MAMerge** *is no more than*

$$\left(\lceil \frac{\rho}{2^{2^{\ell_{max}-1}}} \rceil + 1 \right) \prod_{i=2}^{j} (\overline{\rho}^{1/2^i} + 1)$$

where $\overline{\rho} = 2^{2^{\ell_{max}}}$ and the product $\prod$ above is defined to be 1 when $j = 1$. Moreover, the above quantity is always $O(\rho/\overline{\rho}^{1/2^j})$.

Proof: Firstly, by Lemma 28, the number of times **loadlevel**($\ell_{max} - 1$) can be called is $\lceil \rho/2^{2^{\ell_{max}-1}} \rceil + 1$, by the definition of the **loadlevel**( ) procedure, from invariant 6 of Section 8.5 and the fact that the number W_{top} of non-dummy run-records immediately after the preprocessing stage is no more than $\lceil \rho/2^{2^{\ell_{max}-1}} \rceil$.

By invariant 9 of Section 8.5, for all ℓ such that $1 \leq \ell \leq \ell_{max} - 1$, we have $lr[\ell].Order = 2^{2^{\ell-1}}$.

We prove the lemma by induction on j. From Lemma 28 and the observation above, we know that that **loadlevel**($\ell_{max} - (J+1)$) can be called at most $2^{2^{\ell_{max}-(J+1)}} + 1 = \overline{\rho}^{1/2^{J+1}}$ times each time a **loadlevel**($\ell_{max} - J$) operation completes. The fact that the above expression is $O(\rho/\overline{\rho}^{1/2^j})$ also

follows by induction on j. In fact we can show that the above expression is no more than

$$\left(1 + \frac{2}{\overline{\rho}^{1/2^j}}\right) \frac{2\rho}{\overline{\rho}^{1/2^j}}$$

The lemma follows from the observation that each $\ell_{max} - j, d$-sequence must involve a **loadlevel**$(\ell_{max} - j)$ operation. □

Lemma 30 *Suppose that allocation level $\ell_{max} - j$, where $1 \leq j \leq \ell_{max} - 1$, is charged all the I/O operations that charged to any $\ell_{max} - j, d$-sequence. Then the total number of I/O operations charged to level $\ell_{max} - j$, where $1 \leq j \leq \ell_{max} - 1$, is no more than $O(\rho/\overline{\rho}^{1/2^{j+1}})$.*

Proof: Consider any $\ell_{max} - j, d$-sequence, where $j \geq 1$. By Lemma 29, the maximum number of $\ell_{max} - j, d$-sequences is no more than $O(\rho/\overline{\rho}^{1/2^j})$. Since Lemma 26 proves that the maximum number of I/O operations that can be charged to any $\ell_{max} - j, d$-sequence is $O(2^{2^{\ell_{max}-j-1}}) = O(\overline{\rho}^{1/2^{j+1}})$, the maximum number of I/O operations that can be charged to level $\ell_{max} - j$ is

$$O(\rho/\overline{\rho}^{1/2^j}) \cdot O(\overline{\rho}^{1/2^{j+1}})$$

which is $O(\rho/\overline{\rho}^{1/2^{j+1}})$. Hence the lemma is proved. □

We finally bound the total number of I/O operations and the total amount of resource consumption charged to any **download**( ) call.

Theorem 6 *The total number of I/O operations charged to all* **download**( ) *calls operation is $O(\rho)$. The total resource consumption over all these I/O operations is $O(\rho \lg m_{max})$.*

Proof: Each **download**( ) is, by definition, part of an ℓ, d-sequence. So it is enough to bound the I/O operations charged to ℓ, d-sequences. The total number of I/O operations to be bounded is the sum of the number of I/O operations charged to any level, over all levels. Using Lemma 30, this number is

$$O(\sum_{j=0}^{\ell_{max}-1} \frac{\rho}{\overline{\rho}^{1/2^{j+1}}})$$

which can be simplified as

$$O(\rho(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{16} + \frac{1}{256} + \cdots + \frac{1}{\overline{\rho}^{1/4}} + \frac{1}{\overline{\rho}^{1/2}})).$$

The above expression is $O(\rho)$.

The bound on the total resource consumption is obtained by assuming that the amount of memory available during all the $O(\rho)$ I/O operations counted above is m_{max} and using the definition of resource consumption for memory-adaptive sorting. □

9.1.2 Resource Consumption during bad llmerge( ) calls

A bad **llmerge**( ) call can consume a whole lot of resources while doing very little "work". Such a situation occurs when an **llmerge**(rr_ℓ) execution is preempted at a time when very few items remain to be added to the physical sequence of the run associated with rr_ℓ: If the level of allocation becomes ℓ once more soon after this preemption, just setting up the merge operation **llmerge**(rr_ℓ) can incur $O(2^{2^{\ell-1}})$ I/O operations belonging to an allocation phase of size $O(2^{2^\ell})$.

We now bound the total resource consumption of all bad **llmerge**( ) calls over the execution of **MAMerge** by proving that the total number of I/O operations charged to bad **llmerge**( ) calls over the execution of **MAMerge** is $O(\rho)$. We do so by charging the $O(2^{2^{\ell-1}})$ I/O operations incurred by a bad **llmerge**(rr_ℓ) call to the run-records it touches, and then observing on the one hand that no run-record can ever be touched by more than one bad **llmerge**( ) call and on the other that the total number of run-records created during **MAMerge** is $O(\rho)$.

By invariant 4 of Section 8.6.1 and invariants 8 and 9 of Section 8.5, we have the following lemma.

Lemma 31 *If an execution of* **llmerge**(rr_{global}) *is a bad call, then the algorithm* **MAMerge** *terminates after that call. The total number of I/O operations that can be charged to that* **llmerge**(rr_{global}) *call is* $O(\rho)$.

Due to the above lemma we now consider only bad **llmerge**(rr_ℓ) calls executed when the allocation level is ℓ. We use the following charging scheme to count the total number of I/O operations over all such bad **llmerge**( ) calls.

Definition 31 We charge the total number $O(2^{2^{\ell-1}})$ of I/O operations charged to a bad **llmerge**(rr_ℓ) call (see Lemma 27 and Lemma 17) to the $\Omega(2^{2^{\ell-1}})$ run-records touched (see Lemma 11) by that **llmerge**(rr_ℓ) call.

We now make a useful observation regarding bad **llmerge**(rr_ℓ) calls.

Lemma 32 *When the execution of a bad* **llmerge**(rr_ℓ) *call completes, we have* $rr_\ell.flag = Done$.

Proof: This follows from the definition of the constant κ_{llm} in Definition 25, constant κ_{load} in Lemma 14, invariants 2b and 4 of Section 8.6.1, and the fact that the only reason for **llmerge**(rr_ℓ) to end up being a bad call is that the formation of the run associated with rr_ℓ is logically complete. □

Based on Lemma 32, we make the following useful observation.

Lemma 33 *Any run-record can be touched by at most one bad* **llmerge**( ) *call.*

Proof: Consider any run-record rr' other than run-record rr_{global}. Consider the first bad **llmerge**(rr_ℓ) call that touches run-record rr' and let t denote the time at which that **llmerge**(rr_ℓ) call completes its execution. By Lemma 32, we have $rr_\ell.flag = Done$, when the execution of

llmerge(rr_ℓ) completes. Since rr' is touched by **llmerge**(rr_ℓ), by definition of the merging part of **llmerge**(), there must exist a "path" $rr_\ell = rr(0), rr(1), rr(2), \ldots, rr(d) = rr'$ such that $rr(i)$ is in list $rr(i-1).inputs$. It can be e proved via induction on the length of the path that no **llmerge**() call after time t can touch run-record rr'. This proves the lemma. □

From the charging scheme of Definition 31, it is clear that we can charge each run-record touched by a bad **llmerge**( ) call at most $O(1)$ I/O operations of that **llmerge**( ) call and so, by Lemma 33 above, the total number of I/O operations incurred by bad **llmerge**( ) calls throughout the execution of **MAMerge** is bounded by the total number of run-records created during **MAMerge**. Below we prove that the total number of run-records created during **MAMerge** is $O(\rho)$.

Lemma 34 *The total number of run-records created during* **MAMerge** *is* $O(\rho)$.

Proof: The proof follows by observing that the total number $O(\sqrt{m})$ of run-records created during an ℓ, d-sequence (Lemma 23) is roughly the same as the number of I/O operations incurred by the ℓ, d-sequence (Lemma 22), where $m = 2^{2^\ell}$. Hence, using the same techniques as in Lemma 29, Lemma 30 and Theorem 6, we can argue that the total number of run-records created during **MAMerge** is no more than $O(\rho)$. □

Thus we have proved the following theorem.

Theorem 7 *The total number of I/O operations that can be charged to bad* **llmerge**( ) *calls made during the execution of* **MAMerge** *is* $O(\rho)$. *The total resource consumption that can be charged to bad* **llmerge**( ) *calls made throughout* **MAMerge** *is no more than* $O(\rho \lg m_{max})$.

9.1.3 Resource Consumption of llmerge(rr_{global}) calls

Each **llmerge**(rr_{global}) call appends blocks of items to the physical sequence of the run associated with the "global" run-record. By definition, when **MAMerge** merges ρ run totally consisting of n' blocks, the number of blocks in the run associated with rr_{global} is no more than n'. Blocks are appended to the physical sequence corresponding to rr_{global} in an efficient manner such that the total number of I/Os charged to **llmerge**(rr_{global}) calls is no more than $O(n')$.

We have the following theorem regarding the I/O operations incurred by **llmerge**(rr_{global}) calls.

Theorem 8 *Suppose that the* ρ *runs input to* **MAMerge** *totally consist of* n' *blocks. The total number of I/O operations that can be charged to* **llmerge**(rr_{global}) *calls made during the execution of* **MAMerge** *is* $O(n')$. *The total resource consumption that can be charged to* **llmerge**(rr_{global}) *calls made during the execution of* **MAMerge** *is* $O(n' \lg m_{max})$.

Proof: By Lemma 31, we know that there can be at most one bad **llmerge**(rr_{global}) call during **MAMerge** and it incurs no more than $O(\rho)$ I/O operations. By Lemma 27 and Lemma 16, we know that if a good

llmerge(rr_{global}) call appends g' blocks of items to the run associated with rr_{global}, then that **llmerge**(rr_{global}) call can be charged at most $O(g')$ I/O operations. From invariant 2 of Section 8.5, we know that the total number of blocks that can get appended to the run associated with rr_{global} is no more than n'. The theorem follows. □

9.2 Potential Function Argument

We have shown that the number of I/O operations charged to **download**( ) operations and bad **llmerge**( ) calls is $O(\rho) = O(n')$ whereas the number of I/O operations charged to **llmerge**(rr_{global}) calls is $O(n')$. Since the net number of I/O operations in these activities is small, even if we assume conservatively that throughout these $O(n')$ I/O operations the allocation level was at its maximum value ℓ_{max}, the net resource consumption of these activities remains $O(n' \lg m_{max})$.

On the other hand the number of I/O operations charged to good **llmerge**( ) calls in general may be superlinear in the number of blocks n' output by **MAMerge**. The number of I/O operations charged to good **llmerge**( ) calls in general may be as high as $O(n' \lg \rho)$, which means we cannot adopt the counting strategy mentioned in the above paragraph: Assuming that the allocation level throughout these $O(n' \lg \rho)$ I/O operations is ℓ_{max} would lead to an $O(n'(\lg \rho)(\lg m_{max}))$ bound on resource consumption of good **llmerge**( ) calls, which is clearly not acceptable.

In order to demonstrate that the resource consumption of good **llmerge**( ) calls is bounded by $O(n' \lg m_{max})$ as well, we use the notion of merge potential defined in Section 6. Informally speaking, when a good **llmerge**(rr_ℓ) call is charged g' I/O operations, its resource consumption is $O(g' \cdot 2^\ell)$; but the good **llmerge**(rr_ℓ) call also raises the rank of each one of the $\Omega(g'B)$ items it appends to the physical sequence of the run associated with rr_ℓ by an additive value of $\Omega(2^\ell)$, so that the potential of the merge increases by an amount of $\Omega(g' \cdot 2^\ell)$. Since the maximum value the potential of the merge assumes is $O(n' \lg m_{max})$, it follows that the net resource consumption is $O(n' \lg m_{max})$.

We now formally prove our claims. Each of the run-records of the ρ runs input to **MAMerge** have a unique run assigned to them. It is convenient to logically assign sets containing "dummy input runs" to dummy run-records that may possibly be introduced in course of our preprocessing. The assignment of sets containing dummy input runs to dummy run-records is *only performed for the sake of analysis* and does not alter the computation in any way.

Definition 32 Let U denote the set of ρ runs being merged by **MAMerge**. For the sake of convenience in analysis, we assign a *dummy set* to each dummy run-record created during **MAMerge**'s preprocessing stage. Each dummy set is a a set of one or more *dummy input runs*. Each dummy input run is a run distinct from any of the ρ runs of U and is defined to satisfy the following conditions:

1. Each dummy input run is contained in at most one dummy set.
2. The physical sequence of each dummy input run $e_j(y)$ is always empty.

3. The rank of each dummy input run is 1.

We assign appropriately sized dummy sets to each of the dummy run-records introduced in Step 2 and Step 4 of the preprocessing stage.

Definition 33 Consider the D_1 dummy run-records $d_0, d_1, \ldots, d_{D_1-1}$ introduced in Step 2 of the preprocessing stage. Each dummy run-record d_j is assigned a dummy-set containing a single dummy input run, for $0 \leq j \leq D_1-1$. Consider the D_2 dummy run-records $d'_0, d'_1, \ldots, d'_{D_2-1}$ introduced in Step 4 of the preprocessing stage. Each dummy run-record d'_j is assigned a dummy set containing $2^{2^{\ell_{max}-1}}$ dummy input runs, for $0 \leq j \leq D_2 - 1$.

Just as there is a run associated with a non-dummy run-record, we can now associate runs with dummy run-records.

Definition 34 The run associated with a dummy run-record assigned a set s of dummy input runs is defined to be the (vacuous) merge of the dummy input runs of s. The rank of this run is $|s|$.

A simple lemma that can be proved by induction is the following.

Lemma 35 *Consider a run-record rr which is neither the run-record associated with an input run nor a dummy run-record. Let rr' and rr'' be any two run-records in the list rr.inputs. Let r, r' and r'' respectively be the runs associated with run-records rr, rr' and rr'' respectively. Then we have*

$$p(r') = p(r'') = \frac{p(r)}{rr.Order}$$

The following lemma follows easily from the definition of the potential assigned to a run-record and the potential of the state of the merge.

Lemma 36 *The rank of the run r_{global} associated with run-record rr_{global} does not change with time and is larger than the rank of the run associated with any other run-record in* **MAMerge**. *Moreover, we have $p(r_{global}) < 2\lg\rho = O(\lg m_{max})$. The potential of the merge when* **MAMerge** *completes execution is no more than $2n'\lg\rho$, where n' is the total number of blocks merged by* **MAMerge**.

Now we consider any good **llmerge**(rr_ℓ) operation, where rr_ℓ is the active run-record of level ℓ and such that $rr_\ell \neq rr_{global}$, executed when the allocation level is ℓ. The following lemma proves that the total resource consumption charged to the **llmerge**(rr_ℓ) is within a constant factor of the increase in the potential of the state of the merge brought about by **llmerge**(rr_ℓ).

Lemma 37 *Suppose that an* **llmerge**( ) *call's resource consumption is defined to be the resource consumption over all the I/O operations charged that* **llmerge**( ) *call. Consider a good* **llmerge**(rr_ℓ) *call in which G' items are added to the run associated with the active run-record rr_ℓ of level ℓ, where $rr_\ell \neq rr_{global}$. Then*

1. *The resource consumption charged to* **llmerge**(rr_ℓ) *is $O(G'2^\ell/B)$.*

2. *The increase in the potential of the state of the merge is at least* $G'2^{\ell}/2B$

Proof: By invariant 2b and the description of the **llmerge**( ) procedure we know that all I/O operations charged to **llmerge**(rr_ℓ) are ones in which the allocation phase is of size at most $O(2^{2^\ell})$. By Lemma 27 and Lemma 16 we know that the total number of I/O operations charged to the **llmerge**(rr_ℓ) call is $O(G'/B)$. Thus the net resource consumption charged to **llmerge**(rr_ℓ) is $O((G'/B)\lg 2^{2^\ell}) = O((G'2^\ell)/B)$, by definition of resource consumption.

Consider any item x appended to the physical sequence of the run r_ℓ, associated with rr_ℓ, by **llmerge**(rr_ℓ). From the procedure **llmerge**( ) and the definition of the potential assigned to an item at any time, we know that the maximum possible rank x could have had prior to **llmerge**(rr_ℓ) is $p(r')$, where r' is the run associated with a run-record in the list $rr.inputs$. By invariant 10, 9 and the definition of the active run-record rr_ℓ of level ℓ, we know that $rr_\ell.Order = 2^{2^{\ell-1}}$. Thus by Lemma 35, the rank of element x increases by a factor of at least $2^{2^{\ell-1}}$ during **llmerge**(rr_ℓ). Since **llmerge**(rr_ℓ) adds G' items to the physical sequence of the run associated with rr_ℓ, the net increase in merge potential is at least

$$G' \times \frac{1}{B} \times \lg(2^{2^{\ell-1}}) = G'2^{\ell}/2B.$$

Thus the lemma is proved. □

Lemma 37 and Lemma 36 can now be used to prove the following theorem.

Theorem 9 *The total resource consumption charged to all good* **llmerge**(rr) *calls during* **MAMerge** *in which rr is not the run-record rr_{global} is $O(n' \lg \rho)$, where n' is the total number of blocks of items among the ρ runs input to* **MAMerge**.

Proof: By Lemma 37, we have the condition that whenever any of the good **llmerge**( ) calls in question are charged a resource consumption of R, the increase in the potential of the state of the merge is $\Omega(R)$: Thus at any time the total amount of resource consumption charged to all all good **llmerge**(rr) calls in which $rr \neq rr_{global}$ is within a constant factor of the potential of the state of the merge. Since, by Lemma 36, the maximum value for the potential of the state of the merge is $2n' \lg \rho$, the theorem is proved. □

9.3 Optimality of Resource Consumption for Sorting

From Lemma 8, Theorem 6, Theorem 7, Theorem 9 and the definition of ρ, the number of runs input to **MAMerge**, we have the following theorem.

Theorem 10 *Consider our algorithm* **MAMerge** *used to merge ρ runs each of length at least κ_{model} blocks, totally comprising n' blocks. The total amount of resource consumption incurred by* **MAMerge** *is $O(n' \lg m_{max})$.*

Applying Theorem 10 to Lemma 4 and Corollary 1, we have the following theorem.

Theorem 11 *Consider the memory-adaptive sorting algorithm based on our mergesort framework of Section 5.2 and using* **MAMerge** *as the the memory-adaptive merging routine, as indicated at the beginning of Section 8. When used to sort an input file of n blocks, the resource consumption of our sorting algorithm is $O(n \lg n)$ and the algorithm is dynamically optimal.*

10 Some notes on the potential function

Our potential argument implies that in order to attain dynamic optimality it is necessary for the memory-adaptive mergesort to pump up the merge potential by at least $\Omega(m \lg m)$ during a typical allocation phase of size m. In this context, consider a memory-adaptive merge $\mathcal{M}'$ that tries to attain efficient memory utilization by using the following technique during allocation phases of size m: Algorithm $\mathcal{M}'$ carries out $\Theta(m)$ binary merge operations "in parallel" by dividing the I/Os and the memory blocks of the allocation phase among the $\Theta(m)$ binary merges. This way $\mathcal{M}'$ makes use of all the m memory blocks of the allocation. However, in terms of our potential function, the increase in potential during an allocation phase of size m using algorithm $\mathcal{M}'$ is only $1/B$ for each one of $\Theta(mB)$ items (that is; one block of items for each one of $\Theta(m)$ binary merges), resulting in a net potential increase of $\Theta(m)$, in contrast to a desired increase of $\Omega(m \lg m)$. In fact, the trivial memory-adaptive merging algorithm that consists of always executing a single binary merge merge oblivious of the allocation level also attains the same increase $\Theta(m)$ in merge potential during an allocation phase of size m even though it uses only $O(1)$ blocks of the m blocks allocated to it. This indicates that the strategy $\mathcal{M}'$ is as bad as simple binary merging, notwithstanding the fact that it uses all blocks of the allocation phase.

A much more subtle issue related to our particular potential function is that it allows for the maximum potential and hence the resource consumption to be up to $O(n \lg |S|)$ when the runs of set S totally consist of n blocks. Now this resource consumption is optimal for *memory-adaptive merging* in the case when all input runs are equal in length; however, it is *nonoptimal* for memory-adaptive merging when the input runs can be of arbitrary lengths: To see this point, consider a set S of runs such that one run is $n-|S|-1$ blocks long and all other runs are one block long each. Suppose that $n > |S| \lg |S|$. This set of runs can be merged by first merging together the $|S| - 1$ one-block runs using $O(|S| \lg |S|)$ I/O operations in a binary merging process and then merging the $|S| - 1$-block run with the $n - |S| - 1$-block run to get the n block output run. The net number of I/O operations is $O(n)$. Since all merges here were binary, they can all be performed over a sequence of arbitrary allocation phases. When the allocation sequence consists of $O(1)$ sized allocation phases, the net resource consumption of this strategy is $O(n)$. The $O(n \lg |S|)$ resource consumption permitted by our potential function is rendered nonoptimal in the above example. The question of obtaining a dynamically optimal general purpose algorithm for merging is an interesting open question. In case of our mergesort, although the runs being merged by

algorithm **MAMerge** are, in general, unequal in length and so its resource consumption $O(n \lg m_{max})$ is nonoptimal with respect to the general problem of memory-adaptive merging. Yet our **MAMerge**-based framework is dynamically optimal for *sorting*: Even though **MAMerge** is not dynamically optimal for the general problem of memory-adaptive merging, it does yield a dynamically optimal sorting algorithm.

Yet another question pertains to the manner in which **MAMerge** reorganizes merge computation. Algorithm **MAMerge** splits an $m = 2^{2^{\ell-1}}$-way merge into $\sqrt{m}$-way merges when it needs to reorganize the merge to assign to allocation levels smaller than ℓ. The potential increase registered by **MAMerge** during a typical phase of size m is optimal (with respect to sorting) because this potential increase corresponds to that attained by $2m$ I/O operations of a "traditional" mergesort algorithm A' that repeatedly merges together the fixed number $\sqrt{m}$ runs, which is optimal for a static memory allocation of m blocks. However, for a static memory of m blocks, the mergesort algorithm A'' that repeatedly merges together $\Theta(m)$ runs is more efficient by a constant factor. One interesting question is whether or not there exists a memory-adaptive mergesort which registers a potential increase comparable to that of A'' during a phase of size m.

11 Dynamically Optimal Permuting, FFT, and Permutation Networks

We now consider dynamically optimal algorithms for problems related to sorting. We use the memory-adaptive merging and sorting algorithms as subroutines for developing memory-adaptive permuting and FFT algorithms: The memory-adaptive permutation network follows from the FFT algorithm since a series of at most three approprialtely designed FFT circuits can simulate any permutation network [AV88].

11.1 Permuting

A permuting problem can be solved by first attaching with each item to be permuted its destination address and then sorting the items to be permuted using the address field as key. Our sorting algorithm can be used for this purpose. However, as pointed out in Theorem 2, there are certain circumstances under which the sorting lower bound does not hold for the permuting problem. In the text immediately following Theorem 2 in Section 3.1, we argued that when the sorting lower bound does not hold for permuting, a naive internal memory algorithm using $O(N)$ I/Os and $O(1)$ blocks of internal memory to permute a file of N items is dynamically optimal. Based on this observation we can develop a dynamically optimal permuting algorithm.

Theorem 12 *There exists a simple dynamically optimal permuting algorithm.*

Sketch of Proof: We can run "in parallel", the naive permuting algorithm, which at any time takes only $O(1)$ memory blocks, and the dynamically optimal sorting algorithm. The sorting algorithm gets to use all but $O(1)$ blocks

of memory in each allocation phase and the naive permuting algorithm gets to use one I/O operation for every $O(1)$ I/O operations the sorting algorithm gets so that the number of I/Os spent in total is within a constant factor of I/Os spent on permuting. It can be verified that if we terminate the above algorithm as soon as one of the two algorithms running "in parallel" completes execution, the resource consumption is within a constant factor of optimal. Thus our algorithm is dynamically optimal. □

11.2 Dynamically Optimal FFT and Permutation Networks

We now develop a memory-adaptive version of the FFT algorithm of Vitter and Shriver [VS94], which is based on a series of *shuffle-merge* operations. A shuffle-merge is a merge in which all input runs are equal in length and the output run consists of a perfect item-wise interleaving of the input runs. We define a shuffle merge as follows.

Definition 35 An f-way *shuffle merge* is a merge of f runs each consisting of an equal number, say, p blocks of items such that if the sequence of items of the ith run are denoted by

$$x_{i,0}, x_{i,1}, x_{i,2}, \ldots, x_{i,pB-1},$$

for each i satisfying $0 \leq i \leq f-1$, then the output of the merge is

$$x_{0,0}, x_{1,0}, x_{2,0}, \ldots, x_{f-1,0}, x_{0,1}, x_{1,1}, \ldots, x_{f-1,1}, \ldots\ldots, x_{0,pB-1}, x_{1,pB-1}, \ldots, x_{f-1,pB-1}.$$

The following lemma follows from the definition above and Theorem 5.

Lemma 38 *An m_{max}-way shuffle merge such that each run consists of n'/m_{max} blocks of items can be performed in a memory-adaptive manner by algorithm* **MAMerge** *with a resource consumption of $O(n' \lg m_{max})$.*

In our memory-adaptive simulation of the FFT algorithm of Vitter and Shriver, we chop up the N-input $\lg N$-level FFT digraph into $(\lg N)/\lg M'_{max}$ "layers", where each layer consists of $\lg M'_{max}$ levels and $M'_{max} = 2^{2^{\ell_{max}-1}}B$. Below we describe how to process each layer. In order to route the outputs of one layer to appropriate input destinations of the next layer we need to perform a series of $\max\{1, \log_{m_{max}}(\min\{M'_{max}, N/M'_{max}\})\}$ m_{max}-way shuffle-merge operations each involving n blocks of items [VS94]. By Lemma 38, we can use **MAMerge** to implement all the shuffle-merge operations required to route the outputs of one layer to the inputs of the next. The net resource consumption of all such m_{max}-way shuffle-merge operations, summed over all $(\lg N)/\lg M'_{max}$ layers is

$$O\left(n(\lg m_{max})\frac{\lg N}{\lg M'_{max}}\left(1+\log_{m_{max}}\left(\min\left\{M'_{max}, \frac{N}{M'_{max}}\right\}\right)\right)\right), \qquad (12)$$

which can be simplified to $O(n \lg n)$ [VS94].

Each layer consists of $\lg M'_{max}$ levels and a total of N input items. We can split each layer into N/M'_{max} "groups" such that each group itself is an independent $M'_{max} = 2^{2^{\ell_{max}-1}}B$ input FFT graph. While processing a particular layer, whenever our algorithm has an allocation phase of size

$y \cdot 2^{2^{\ell max - 1}}$ for $y \geq 1$, we compute the outputs of $\lfloor y \rfloor$ groups during that phase. In order to function efficiently during smaller allocation phases, we make the crucial observation that pebbling through an an FFT graph with $M'_{max} = 2^{2^{\ell max - 1}}$ inputs is equivalent to first pebbling through $2^{2^{\ell max - 2}}$ independent FFT graphs each with $2^{2^{\ell max - 2}}B$ input nodes, then executing a $2^{2^{\ell max - 2}}$-way shuffle-merge totally involving $2^{2^{\ell max - 2}}$ blocks, and then pebbling through $2^{2^{\ell max - 2}}$ independent FFT graphs each with $2^{2^{\ell max - 2}}B$ input nodes once more. This decomposition of an $m = 2^{2^{\ell max - 1}}$ block FFT digraph into $\sqrt{m} = 2^{2^{\ell max - 2}}$ equally sized, independent FFT digraphs is analogous to the splitting of an m-way merge into $\sqrt{m}$ merges, each one itself a $\sqrt{m}$-way merge. Each of these steps can be implemented in a memory-adaptive manner by using modifications of the data structures and online reorganization techniques that we developed for **MAMerge** so that the net resource consumption incurred in processing a single group is $O((M'_{max}/B)\lg(M'_{max}/B)) = O((M'_{max}/B)\lg m_{max})$. As a result each layer can be processed with a resource consumption of $O(n \lg m_{max})$.

Thus the net resource consumption of our FFT algorithm is $O(n \lg n)$ and we have a dynamically optimal FFT algorithm. It follows that we also have a dynamically optimal algorithm for arbitrary permutation networks since it is well known that any N input permutation network can be simulated by at most three appropriate N input FFT digraphs placed one after the other; see [AV88] for references.

12 Extending adaptability to other applications

In this section we present a memory-adaptive version of the buffer tree data structure introduced by Arge [Arg94]. The buffer tree is a general technique to efficiently externalize many internal memory data structures. The most appealing aspect of the buffer tree is that it isolates the I/O specific parts of data structures so that a specific set of I/O efficient techniques can be applied to several different internal memory data structures that have a "buffer tree wrapper", resulting in I/O optimal (in an amortized sense [Arg94]) algorithms for several applications consisting of *batched dynamic problems*. These include improved graph algorithms, ordered binary decision disgrams, external heaps and string sorting, among other applications. (See [Arg96] and [Vit98] for details.) More recent applications for the buffer tree include "bulk loading" operations on R-trees and B-trees [AHVV98].

In order to present a concise description of our memory-adaptive buffer tree, it is convenient to informally differentiate between two types of external memory problems: External memory problems that are *memory-oblivious* and external memory problems that are *memory-intensive*. Memory-oblivious problems include operations such as scanning a file, partitioning a file into a constant number of buckets, merging together a constant number of runs: Assuming that the input size consists of n blocks, all the above problems can be solved in an optimal, linear number $O(n)$ number of I/O operations regardless of internal memory size and are unaffected positively or negatively, by memory fluctuations. The **download**( ) operation of **MAMerge** is an example of a memory-oblivious operation. On the other

hand, memory-intensive problems include sorting a file, merging a set of $\omega(1)$ runs together, distributing an unsorted file into $\omega(1)$ buckets. In such applications it is essential to mimic an optimal algorithm designed for the static, m-block memory version of the problem during allocation phases of size m; not doing so results in nonoptimal utilization of allocations and accompanying overhead. Memory-adaptive algorithms for such applications are affected by memory fluctuations and changes in the allocation level: When such allocation levels drop they have to do more I/O to do the same amount of work and when allocation levels rise they can do the same work with a small number of I/O operations.

We identify the *buffer-emptying* operation to be the only memory-intensive operation involved in implementing buffer trees and use our dynamically optimal sorting technique to render an optimal, memory-adaptive version of the buffer emptying operation. Since all other operations involved in the buffer tree technique are memory-oblivious, this is enough to guarantee a dynamically optimal buffer tree (in an amortized sense) by invoking the same arguments as in [Arg94].

Definition 36 A buffer tree of fanout parameter m' consists of an (a, b)-tree extended with m' buffer blocks per internal node where $a = m'/4, b = m'$, as defined in [Arg94]. A *memory-adaptive buffer tree* is a buffer tree of fanout parameter m'_{max}, where m'_{max} is a polynomial in m_{max}, with memory-adaptive performance; that is, the buffer tree operations have to be performed over an allocation sequence as per our dynamic memory model.

Insert and Delete operations on the buffer tree may involve the execution of buffer emptying computations and splitting and merging nodes in course of rebalancing operations. We refer the reader to [Arg94] for details on the buffer tree. An examination of buffer tree operations and the techniques used to implement them reveal that the buffer emptying operation is the only memory-intensive buffer tree operation, all other operations being memory-oblivious.

12.1 Memory-adaptive buffer emptying of internal nodes

In the original buffer tree [Arg94], the buffer emptying process at internal node v consists of using $\Theta(m')$ blocks of memory to empty the contents associated with node v as follows:

1. Load into internal memory and sort at most $m'B/2$ timestamped items of the buffer associated with node v.

2. Distribute these sorted items, after cancelling out "annihilating pairs", into at most $O(m')$ nodes corresponding to the children of node v using the $\Theta(m')$ partitioning elements associated with v.

The above procedure takes $O(m')$ I/O operations using $\Theta(m')$ internal memory blocks. We show how to implement the above buffer emptying process in our dynamic memory model using no more than $O(m'_{max} \lg m_{max})$ resource consumption in our memory-adaptive buffer tree.

1. The $m'_{max}B/2$ or fewer timestamped items of node v are sorting using our memory-sorting algorithm using no more than $O(m'_{max} \lg m_{max})$ resource consumption. This implements Step 1 of the above buffer emptying operation.

2. Since the set of $m'_{max}B/2$ or fewer items have been sorted, we can carry out the distribution of these items to the children of v in a memory-oblivious manner: stream through these records and the sorted list of partitioning elements of node v: this takes no more than $O(m'_{max})$ I/O operations. The net resource consumption over this implementation of Step 2 of the buffer emptying process cannot be more than $O(m'_{max} \lg m_{max})$.

Hence we have the following lemma.

Lemma 39 *The total amount of resource consumption during a buffer emptying operation on an internal node of our buffer tree with fanout parameter m'_{max} is $O(m'_{max} \lg m_{max})$.*

12.2 Memory-adaptive buffer emptying of other nodes

We use the buffer emptying process of Section 12.1 even in the case of tree nodes just above the level of leaf nodes, which are not considered internal nodes [Arg94]. However, while the buffer tree guarantees that the buffer emptying computation at each internal node never involves more than $O(m'_{max})$ blocks of records to be emptied, there is no such guarantee while emptying the buffer of a tree node just above the level of leaf nodes. So, as far as nodes just above the level of leaf nodes are concerned, the buffer emptying process described above can involve more than $m'_{max}B/2$ items. If the buffer tree is being used to carry out a sequence of $N = nB$ arbitrary insert/delete operations, the number of items involved in a buffer emptying operation at a node just above the tree node could be as high as $O(nB)$. Fortunately, it can be shown [Arg94] that each item[12] can be involved at most once in a buffer-emptying computation at any node on the level just above the leaf level. This means that the number of items involved in buffer emptying operations summed over all the nodes on the level just above leaf nodes is $O(nB)$.

Using the same technique used to implement the buffer emptying computation for internal nodes we have the following lemma.

Lemma 40 *Consider an initially empty buffer tree with fanout parameter m_{max} that evolves over an arbitrary sequence of nB insert/delete operations. Consider all the buffer emptying operations that occur at nodes on the level just above the leaf nodes of the tree. Let the ith such buffer emptying operation occur at node v_i. Let the total number of blocks of the items associated with node v_i's buffer be n_i. Then*

1. $\sum_i n_i \leq n$.

[12] Each item here has a unique timestamp. The ith insert/delete operation on the buffer tree generates a record with timestamp i.

2. *The total amount of resource consumption R_i of the buffer emptying operation on node v_i is no more than $O(n_i \lg n_i)$.*

3. *The total amount $\sum_i R_i$ of resource consumption over all buffer emptying operations at nodes at the level just above the leaf nodes over the entire sequence of nB insert/delete operations is $\sum_i R_i = O(n \lg n_i)$.*

12.2.1 Main result on buffer tree

Based on Lemma 39 and 40 and the amortization arguments of Theorem 1 of [Arg94], we have the following theorem regarding resource consumption of our memory-adaptive buffer tree.

Theorem 13 *The total resource consumption over an arbitrary sequence of nB insert and delete operation on an initially empty memory-adaptive buffer tree is $O(n \lg n)$; the amortized resource consumption of each operation is $O((1/B) \lg n)$. Our memory-adaptive buffer tree is dynamically optimal.*

Proof: The total number of times each of the N items can be involved in buffer empting operations at "full" internal nodes is $O(\lg n / \lg m'_{max})$ [Arg94]. Since each such buffer emptying process involves $\Theta(m'_{max} B)$ items and incurs a resource consumption of $O(m'_{max} \lg m'_{max})$ resource consumption (by Lemma 39) , the total resource consumption for all buffer emptying operations involving "full" internal nodes is

$$N \times \frac{\lg n}{\lg m'_{max}} \times \frac{m'_{max} \lg m'_{max}}{m'_{max} B} = O(n \lg n)$$

Each rebalance operation[13] can incur no more than $O(m'_{max} \lg m'_{max})$ resource consumption (by Lemma 39), and there are no more than $O(n/m'_{max})$ rebalancing operations over the entire sequence of nB insert/delete operations [Arg94]. Thus rebalancing cannot cost more than $O(n \lg m_{max}) = O(n \lg n)$ resource consumption. Lemma 40 accounts for the resource consumption during buffer emptying operations at non-internal nodes just above the level of leaves. This proves the theorem. □

13 Dynamically Optimal Memory-adaptive Matrix Arithmetic

In this section we consider the problem of multiplying two $\hat{N} \times \hat{N}$ matrices in a dynamically optimal manner. It turns out that the techniques developed here for matrix multiplication also apply to the problem of LU factorization of an n block matrix, as implied by results in [WGWR93]. We first consider issues related to disk block layout of matrices that arise during the algorithm.

[13] In each rebalance operation, either a node is split into two or two nodes are fused into one; the latter operation may require emptying some "non-full" buffer containing $O(m'_{max})$ blocks. See [Arg94] for details.

13.1 Transformation between different blocking orders

Consider the multiplication of two $\hat{N} \times \hat{N}$ matrices A and B each consisting of $N = \hat{N}^2$ elements spread over $n = N/B$[14] elements disk blocks. The product matrix $C = AB$ also consists of N elements. For convenience, we assume in this section that n is a power of 4: If this condition is not met each matrix can be padded without changing the asymptotic running time of our algorithm.

Very often, matrices are stored in *row-major* order, defined below, on disk. However, in many external memory matrix algorithms it is more convenient to store matrices in *two-dimensional blocked* order, defined below, on disk.

Definition 37 An $N = \hat{N} \times \hat{N}$ sized matrix A is said to be stored in row-major order on disk if its elements are on n blocks $b_{i,j}$, where $0 \leq j \leq \sqrt{N}/B - 1$ and $0 \leq i \leq \sqrt{N} - 1$, such that block $b_{i,j}$ contains elements $A[i, jB]$ through $A[i, jB + B - 1]$.

An $N = \hat{N} \times \hat{N}$ sized matrix A is said to be stored in two-dimensional blocked order on disk if its elements are on n blocks $b_{i,j}$, where $0 \leq i, j \leq j$, such that $b_{i,j}$ contains the $\sqrt{B}$ elements $A[i', j\sqrt{B}]$ through $A[i', j\sqrt{B}+\sqrt{B}-1]$ of row i', for each i' such that $i\sqrt{B} \leq i' \leq i\sqrt{B} + \sqrt{B} - 1$. We assume that each block $b_{i,j}$ of the two-dimensional blocked order has pointers to the (at most two) blocks adjacent to it in the row i of blocks and the (at most two) blocks adjacent to it in the column j of blocks.

We can transform an $\hat{N} \times \hat{N}$ matrix's storage format from row-major order to two-dimensional blocked order as follows: If $\hat{N} \leq \sqrt{B}$, the matrix is already in two-dimensional blocked order. Assuming $\hat{N} > \sqrt{B}$, in a linear pass over the matrix we add dummy elements if necessary to ensure that the number of rows and columns in the matrix is a multiple of $\sqrt{B}$. With some abuse of notation, we use the same symbol $\hat{N}$ to denote the resulting number of rows and columns. If $r = \min\{\sqrt{B}, \hat{N}/\sqrt{B}\}$, we can perform the desired transformation by performing a series of r-way merges. Each of the r runs are one block long and the (implicit) keys of the merged items are determined by their position in the row-major order.

During allocation phases of size xr, where $\lfloor x \rfloor \geq 1$, $\lfloor x \rfloor r$ row-major order blocks can be trivially transformed into $\lfloor x \rfloor r$ two-dimensional order blocks: The resource consumption of the transformation algorithm during any such allocation phase is $O((xr)^{3/2})$.

On the other hand, during allocation phases smaller than r, we can employ the memory-adaptive merging algorithm **MAMerge** to appropriately implement r-way merges realizing the required transformation. If $m_1, m_2, \ldots, m_t$ are the sizes of the allocation phases involved in a single such r-way memory-adaptive merge computation, we know from Theorem 10 that

$$\sum_{i=1}^{t} m_i \lg m_i \leq cr \lg r$$

[14] In contrast to previous discussions, in this Section we use N to denote the size of the output, as opposed to the input, which here is of size $2N$.

where c is a positive constant. It follows that

$$\sum_{i=1}^{t}(m_i \lg m_i)\left(\frac{\sqrt{m_i}}{\lg m_i}\right) \leq (cr \lg r) \max_{1\leq i\leq t}\left\{\frac{\sqrt{m_i}}{\lg m_i}\right\}$$

which implies that the net resource consumption over the phases $m_1, m_2, \ldots, m_t$ is no more than $O(r^{3/2})$, since $\max_{1\leq i\leq t}\{\sqrt{m_i}/\lg m_i\} \leq \sqrt{r}/\lg r$. Thus we have the following lemma.

Lemma 41 *Any $\hat{N} \times \hat{N}$ matrix consisting of $N = nB = \hat{N}^2$ elements can be transformed from row-major order to two-dimensional blocked order and vice-versa without incurring a resource consumption of more than $O(n^{3/2})$.*

Since we are interested in an $O(n^{3/2})$ bound on the resource consumption of our memory-adaptive matrix multiplication algorithm, by Lemma 41, we can assume without loss of generality that all matrices are stored in two-dimensional blocked order on disk.

13.2 Memory-adaptive Matrix Multiplication

The in-memory matrix multiplication $AB = C$, where each of A,B and C consist n blocks, can be executed using approximately $3n$ blocks of internal memory and $\Theta(n)$ I/O operations. Thus we set the parameter m_{max} of the dynamic memory model to be $m_{max} = \min\{3n, phy_{max}\}$.

The multiplication of large matrices can be carried out by a series of multiplications of smaller matrices as explained below. We first chop the matrices A,B and C into submatrices each consisting of an appropriate number $\hat{m}_{max} = \Theta(m_{max})$ of disk blocks. We organize the computation $AB = C$ to proceed in $(n/\hat{m}_{max})^{3/2}$ steps such that each step is guaranteed to incur resource consumption of $O(\hat{m}_{max}^{3/2})$, thus obtaining a dynamically optimal memory-adaptive algorithm.

Suppose that matrix A is partitioned into $(n/\hat{m}_{max})$ square submatrices $A_{i,j}$, where $0 \leq i, j \leq \sqrt{n/\hat{m}_{max}} - 1$, such that each square submatrix consists of $\sqrt{\hat{m}_{max}} \times \sqrt{\hat{m}_{max}} = \hat{m}_{max}$ blocks. Suppose that B and C are similarly partitioned into square submatrices each consisting of $\hat{m}_{max}$ blocks. Then, we organize out computation of $AB = C$ to proceed as follows:

1. For $0 \leq i, j \leq \sqrt{n/\hat{m}_{max}} - 1$, $C_{i,j} := 0$. (Set each submatrix $C_{i,j}$ of matrix C to zero.)

2. For $0 \leq i, j, k \leq \sqrt{n/\hat{m}_{max}} - 1$, $C_{i,j} := C_{i,j} + A_{i,k}B_{k,j}$. (Compute $A_{i,k}B_{k,j}$ and add the resulting $\hat{m}_{max}B$ elements to the corresponding existing $\hat{m}_{max}B$ elements of $C_{i,j}$.)

The following lemma can be proved easily.

Lemma 42 *The computation indicated above in Steps 1 and 2 correctly outputs the product matrix $C = AB$.*

The computation indicated in Step 1 above can easily be carried out implicitly. In the remainder of this Section we show how to carry out the

computation $C_{i,j} := C_{i,j} + A_{i,k}B_{k,j}$ indicated in Step 2 above in a memory-adaptive manner incurring only $O(\hat{m}_{max}^{3/2})$ resource consumption, thus yielding a dynamically optimal matrix multiplication algorithm. In order to mimic the standard, I/O-optimal matrix multiplication algorithm for static m-block internal memory, we need to carry out a matrix multiplication operation involving $\Theta(m)$ blocks during an allocation phase of size m: Due to the nature of the resource consumption, we need to "pebble" $\Theta((mB)^{3/2})$ in a "typical" allocation phase of size m, in order to achieve optimality. This suggests that the matrix multiplication computation should be organized in such a manner that whenever the allocation phase size m' is in the range $[m, cm]$, for a constant $c > 1$, we should carry out a a multiplication of two square sub-matrices each containing $\Theta(c^{\log_c m})$ disk blocks: This ensures that the number of DAG nodes pebbled is $\Theta((c^{\log_c m}B)^{3/2}) = \Theta((m'B)^{3/2})$.

Definition 38 In our memory-adaptive matrix multiplication algorithm an allocation phase of size m is said to be at allocation level $level(m)$, where $level(m)$ is defined (for our matrix multiplication algorithm) to be $\lceil \log_4 \frac{m}{c_{level}} \rceil$, where c_{level} is an appropriately chosen positive constant. Thus each allocation phase is at some level ℓ where $1 \leq \ell \leq \ell_{max}$ and $\ell_{max} = level(m_{max})$. We define $\hat{m}_{max}$ to be the number $4^{\ell_{max}-1}$.

13.3 Mop Records and Level-Records

We focus now on the problem of implementing the computation of Step 2 in a memory-adaptive manner. In our scheme we have to deal with square submatrices consisting of $2^\ell \times 2^\ell$ blocks, where $1 \leq \ell \leq \ell_{max}$. Recall that we assume that matrices are stored in two-dimensional blocked order on disk. Thus, given a pointer to any one block of a submatrix, we can easily access all other blocks of the submatrix in order to to load the disk blocks submatrix into memory. Without loss of generality, we choose the pointer $p(\hat{A})$ to a specific block of a matrix $\hat{A}$ to act as the handle for matrix $\hat{A}$.

Definition 39 Given a square matrix $\hat{A}$ stored in two-dimensional blocked order on disk, the *lt-ptr* $p(\hat{A})$ of matrix $\hat{A}$ is the pointer to that block of $\hat{A}$ that is the intersection of $\hat{A}$'s first row of blocks with its first column of blocks.

We are now in a position to describe *mop* (matrix operation) records, each of which corresponds to a multiplication of submatrices consisting of $2^\ell \times 2^\ell$ blocks, for some ℓ such that $1 \leq \ell \leq \ell_{max}$.

Definition 40 Consider the mop record mr corresponding to the matrix-multiplication $\hat{C} := \hat{C} + \hat{A}\hat{B}$, where each one of $\hat{A}, \hat{B}$ and $\hat{C}$ consist of $2^\ell \times 2^\ell$ blocks. We denote by $\hat{A}_{i,j}$, where $0 \leq i, j \leq 1$, the four $2^{\ell-1} \times 2^{\ell-1}$-block non-overlapping square submatrices, that $\hat{A}$ can be decomposed into. Similarly, we denote by $\hat{B}_{i,j}$ and $\hat{C}_{i,j}$, where $0 \leq i, j \leq 1$, respectively the square submatrices resulting from a similar decomposition of B and C respectively. The mop record mr then consists of the following fields:

1. *ltptrs*: This field is assigned the triple $p(\hat{A}), p(\hat{B})$, and $p(\hat{C})$ of lt-ptrs of matrices $\hat{A}, \hat{B}$ and $\hat{C}$ respectively.

2. *lsize*: This field assigned the number ℓ.

3. *split*: This field is assigned the twelve pointers $p(\hat{A}_{i,j}), p(\hat{B}_{i,j}), p(\hat{C}_{i,j})$ where $0 \leq i, j \leq 1$, which are respectively the lt-ptrs of the twelve submatrices $\hat{A}_{i,j}, \hat{B}_{i,j}$, and $\hat{C}_{i,j}$ defined above.

The twelve pointers assigned to the field *split* of a mop record are used to further split the matrix multiplication operation if needed, as follows: If matrix $\hat{A}$ (respectively $\hat{B}$ and $\hat{C}$) is broken down into four square submatrices $\hat{A}_{i,j}$ (respectively $\hat{B}_{i,j}$ and $\hat{C}_{i,j}$) where $0 \leq i, j \leq 1$, then the product $\hat{C} := \hat{C} + \hat{A}\hat{B}$ can be computed by computing the eight products, $\hat{C}_{i,j} := \hat{C}_{i,j} + \hat{A}_{i,k}\hat{B}_{k,j}$, where $0 \leq i, j, k \leq 1$.

Definition 41 Consider the mop record mr corresponding to the operation $\hat{C} := \hat{C} + \hat{A}\hat{B}$, with $\hat{A}_{i,j}, \hat{B}_{i,j}$, and $\hat{C}_{i,j}$ for $0 \leq i, j \leq 1$ as defined above. Suppose q is an integer such that $0 \leq q \leq 7$. Then by " the qth $0-1$ triple", we refer to the triple (i', j', k') that is the qth triple in a lexicographic ordering of the eight triples $\{(i, j, k) : 0 \leq i, j, k \leq 1\}$. And by "the qth subproduct of mop record mr", we refer to the product $\hat{C}_{i',j'} := \hat{C}_{i',j'} + \hat{A}_{i',k'}\hat{B}_{k',j'}$, where (i', j', k') is the qth $0-1$ triple.

We are now in a position to describe level-records for matrix-multiplication, which perform the same role here that they played in our memory-adaptive sorting algorithm: That is, given an allocation phase at level ℓ, we can simply look up $lr[\ell]$, the level-record corresponding to level ℓ, to decide what computation to carry out during that phase.

Definition 42 Consider ℓ such that $1 \leq \ell \leq \ell_{max}$. The level-record $lr[\ell]$ corresponding to allocation level ℓ is either set to *nil* or consists of the following fields:

1. mr: This is assigned a mop-record mr such that $mr.lsize = \ell$.

2. The integer *ctriple* such that $0 \leq ctriple \leq 7$.

All level-records $lr[\ell]$, where $1 \leq \ell \leq \ell_{max}$, are stored as as blocked linked list.

When our algorithm is subjected to a phase at level ℓ, our algorithm looks up $lr[\ell]$ and then executes computation corresponding to the $lr[\ell].ctriple$-th subproduct of mop record $lr[\ell].mr$, incrementing $lr[\ell].ctriple$.

13.4 The loadlevel( ), download( ), and llmult( ) Subroutines

We now describe the **loadlevel**( ) and **download**( ) functions that provide the same functionality they did during memory-adaptive sorting.

The algorithm maintains a variable called *maxlevel* such that $1 \leq maxlevel \leq \ell_{max}$ is always true. Intuitively, *maxlevel* is such that any time our algorithm is subjected to an allocation phase at level $maxlevel + 1$ or higher, it completes the entire computation involved in an instance of Step 2 of Section 13.2: If, on any given instance, some computation pertaining to

that instance has already been completed before receiving the phase at allocation level greater than *maxlevel*, we simply execute the remaining computation required to finish processing that instance, during that phase. Even if the allocation level is never *maxlevel* + 1, our algorithm completes the computation of each instance of Step 2 of Section 13.2 efficiently. Until the processing of a given instance of Step 2 of Section 13.2 is not completed, we have $lr[maxlevel] \neq nil$. Thus the variable *maxlevel* is updated appropriately during the functions **loadlevel**( ) and **download**( ).

The subroutine loadlevel(ℓ)

As mentioned earlier, our goal is to execute a subproduct of $lr[\ell].mr$ when the allocation level is ℓ. When all such subproducts of a given mop record $lr[\ell].mr$ are completed, $lr[\ell]$ is set to *nil*. When this happens, we need to assign computation work to $lr[\ell]$ appropriately in a dynamic manner in order to use future phases at level ℓ effectively. The procedure **loadlevel**(ℓ), where $1 \leq \ell \leq \ell_{max} - 1$, given below, is executed to assign work from $lr[\ell + 1]$ to $lr[\ell]$ and is executed only when $lr[\ell + 1] \neq nil$:

1. Suppose that $lr[\ell + 1].mr$ corresponds to the matrix operation $\hat{C} := \hat{C} + \hat{A}\hat{B}$. Let $q = lr[\ell+1].ctriple$ and let (i, j, k) be such that $\hat{C}_{i,j} + \leftarrow \hat{A}_{i,k}\hat{B}_{k,j}$ is the qth subproduct of $lr[\ell+1].mr$. Suppose x is a new mop record to be appropriately initialized.

2. Set $x.ltptrs$ to the triple $p(\hat{A}_{i,k}), p(\hat{B}_{k,j}), p(\hat{C}_{i,j})$ of lt-ptrs.

3. Set $x.lsize$ to ℓ.

4. If $\ell \geq 1$, then compute the four lt-ptrs $p(X_{i',j'})$, where $0 \leq i', j' \leq 1$, corresponding to the four $2^{\ell-1} \times 2^{\ell-1}$ block submatrices $X_{i',j'}$, for each one of $X = \hat{A}_{i,k}, X = \hat{B}_{k,j}$, and $X = \hat{C}_{i,j}$. These pointers can be computed by traversing appropriately the boundary blocks of X, for a given value of X. Set $x.split$ to the twelve pointers so obtained.

5. Set $lr[\ell].mr$ to x and $lr[\ell].ctriple = 0$.

6. If $lr[\ell+1].ctriple < 7$, increment $lr[\ell+1].ctriple$. Otherwise, set $lr[\ell+1]$ to *nil* and If $maxlevel = \ell + 1$ set *maxlevel* to ℓ.

The following lemma bounds the total number of I/Os and the total number of internal memory blocks required to execute **loadlevel**(ℓ).

Lemma 43 *The total number of internal memory blocks required during the execution of* **loadlevel**(ℓ) *is* $O(1)$. *The total number of I/O operations incurred during* **loadlevel**(ℓ) *is* $O(2^{\ell-1})$.

Proof: It is easy to see that no more than a constant amount of internal memory is required during **loadlevel**(ℓ). As regards the number of I/O operations, it can be seen that for each one of the three instances of X, obtaining the lt-ptrs of $X_{i',j'}$, where $0 \leq i', j' \leq 1$, takes no more than $O(2^{\ell-1})$ I/O operations. No other activity during **loadlevel**(ℓ) incurs any I/O. Thus the lemma is proved. □

The subroutine download(ℓ')

Whenever $lr[\ell']$ is *nil* we may need to assign some new work to $lr[\ell']$ from some level-record at a higher level $\ell+1$, where $\ell' \leq \ell$, via a series of applications of **loadlevel**(ℓ'') for $\ell' \leq \ell'' \leq \ell$. We present below the steps involved in **download**(ℓ'), which is only executed when $\ell' < maxlevel$:

1. Set $\ell'' = \ell'$.
2. While $lr[\ell''] = nil$, $\ell'' = \ell'' + 1$.
3. Set $\ell = \ell'' - 1$.
4. For ℓ'' going from ℓ down to ℓ', execute **loadlevel**(ℓ'').

Definition 43 The levels ℓ' through ℓ are said to have been loaded by the **download**(ℓ') call described above. Level ℓ is said to be the highest level to be loaded.

The following lemma bounds the total memory and I/O requirement of **download**(ℓ'). The proof follows easily from Lemma 43.

Lemma 44 *Suppose ℓ is as defined above; that is, ℓ is the highest level to get loaded during* **download**(ℓ'). *Then the total number of I/Os incurred in first accessing $lr[\ell']$ by following the blocked linked list of level-records and then executing* **download**(ℓ') *is $O(2^{\ell-1})$. The total number of internal memory blocks required is $O(1)$.*

The following lemma is useful while accounting for the resource consumption during **download**( ) operations.

Lemma 45 *Consider a sequence of d operations such that:*

1. *In the ith operation, where $0 \leq i \leq d-1$, we traverse the blocked linked list of level-records to access $lr[\ell_i]$ and then execute* **download**(ℓ_i).
2. *Over the entire sequence of d operations, for no level ℓ', where $1 \leq \ell' \leq \ell_{max} - 1$, is* **loadlevel**($\ell'$) *executed more than once. Moreover, let ℓ denoted the highest level ℓ' for which* **loadlevel**(ℓ') *was executed over the sequence of d operations.*

Then there exists a small positive constant c_{dl} such that the number of I/O operations over the entire sequence of d operations is no more than $c_{dl} \cdot 2^{\ell-1}$.

llmult( )

We now describe the simple matrix multiplication routine **llmult**(ℓ) executed when the allocation level is ℓ and $lr[\ell]$ is not *nil*. Basically this routine simply reads in the blocks of the submatrices involved in the qth subproduct of $lr[\ell].mr$, where $q = lr[\ell].ctriple$, carries out the multiplication and addition, and then writes the blocks back to disk.

1. Suppose $lr[\ell].ltptrs$ contains the lt-ptrs of submatrices $\hat{A}, \hat{B}$, and $\hat{C}$ respectively, each consisting of $2^\ell \times 2^\ell$ blocks. Suppose $lr[\ell].ctriple$ is q and that (i, j, k) is the qth $0-1$ triple.

2. Use the lt-ptrs $p(\hat{A}_{i,k}), p(\hat{B}_{j,k})$ and $p(\hat{C}_{i,j})$ stored in the field $lr[\ell].mr.split$ to respectively read in blocks of the three $2^{\ell-1} \times 2^{\ell-1}$-block submatrices $\hat{A}_{i,k}, \hat{B}_{j,k}$, and $\hat{C}_{i,j}$.

3. Perform the internal memory computation $\hat{C}_{i,j} := \hat{C}_{i,j} + \hat{A}_{i,k}\hat{B}_{k,j}$.

4. Write the disk blocks of $\hat{C}_{i,j}$ back to disk. [15]

5. If $lr[\ell].ctriple < 7$, it is incremented. Otherwise $lr[\ell]$ is set to *nil*. Level-record $lr[\ell]$ is written to disk.

We will now bound the toal number of I/Os and the total memory requirement of **llmult**(ℓ).

Lemma 46 *The number of I/O operations incurred in accessing $lr[\ell]$ using the blocked linked list of level-records is no more than $\ell/B+1$. The number of I/O operations incurred during* **llmult**(ℓ) *is no more than $4\times 4^{\ell-1}$. The total number of internal memory blocks required is no more than $4 \times 4^{\ell-1}$. There exists a small constant c'_{llm} such that the total number of I/O operations incurred in first accessing $lr[\ell]$ and then executing* **llmult**(ℓ) *is bounded by $c'_{llm} \cdot 4^{\ell-1}$ and the total number of memory blocks required is bounded by $c'_{llm} \cdot 4^{\ell-1}/2$.*

Proof: The proof is trivial since the number of level-records in the accessed portion of the blocked list of level-records is ℓ and the number of blocks of $\hat{A}_{i,k}, \hat{B}_{k,j}$, and $\hat{C}_{i,j}$ each is $4^{\ell-1}$; blocks of $\hat{A}_{i,k}$ and $\hat{B}_{k,j}$ are only read in whereas blocks of $\hat{C}_{i,j}$ are input and then output after computation. □

We present a useful lemma that will come in handy while accounting for resource consumption.

Lemma 47 *Suppose c'_{llm} is as defined in Lemma 46. Then there exists a small positive constant c_{llm} such that $\sum_{\ell'=1}^{\ell} c'_{llm} \cdot 4^{\ell'-1} \leq c_{llm} \cdot 4^{\ell-1}$.*

13.5 Algorithm MAMultiply

The procedure **download**( ) described above is memory oblivious in the sense it can function with some constant number c of internal memory blocks and since we ensure that the smallest allocation phase has size c, it can execute in any allocation phase. The procedure **llmult**(ℓ) on the other hand requires $O(4^{\ell-1})$ internal memory blocks over a sequence of $O(4^{\ell-1})$ I/O operations, so it is executed when the allocation level is ℓ. Now we show how to sew these two procedures together to obtain a memory-adaptive matrix multiplication algorithm.

Consider some point of time at which the allocation level is ℓ and we could start execution on **llmult**(ℓ): It is appropriate to actually go through with the call to **llmult**(ℓ) only when either the current allocation phase, say of size m, has $O(4^{\ell-1})$ I/O operations remaining in it or if we know that

[15] In practice the level ℓ matrix multiplication operations can be ordered such that the next level ℓ operation is $\hat{C}_{i,j} \ +\leftarrow \hat{A}_{i,k+1}\hat{B}_{k+1,j}$ so that disk blocks of $\hat{C}_{i,j}$ would not be written back to disk if if the allocation level remains ℓ long enough for this next operation to immediately follow the just completed operation.

the next allocation phase is also a level ℓ allocation phase. We define the following predicate $enough(m)$ to guide this decision of the memory-adaptive algorithm

Definition 44 During an ongoing allocation phase of size m such that $level(m) = \ell$, the boolean predicate $enough(m)$ is true if and only if $left \geq c_{llm} \cdot 4^{\ell-1}$ or $level(next) = \ell$.

Now we define the constant c_{level} appropriately, which is instrumental in the classificastion of allocation phase sizes into different allocation levels.

Definition 45 We define the constant c_{level} to be the smallest constant such that $2c_{level} \geq c_{llm} + c_{dl}$.

We now present the memory-adaptive matrix multiplication that carries out the computation $\hat{C}+ \leftarrow \hat{A} \cdot \hat{B}$, where each of $\hat{A}, \hat{B}$, and $\hat{C}$ consist of $\sqrt{\hat{m}_{max}} \times \sqrt{\hat{m}_{max}}$ blocks, thus yielding a memory-adaptive implementation of Step 2. We use *clevel* to mean $level(mem)$ in the following description.

1. Initialize all fields of a new mop record x corresponding to the matrix-multiplication operation $\hat{C}+ \leftarrow \hat{A} \cdot \hat{B}$. Then set $lr[\ell_{max}].mr$ to x and $lr[\ell_{max}].ctriple$ to 0. Set *maxlevel* to ℓ_{max}.

2. While $(lr[maxlevel] \neq nil)$ execute the following:

   (a) Walk through level-records until $lr[\min\{clevel, maxlevel\}]$ is in memory.

   (b) If $(clevel > maxlevel)$, complete the operation loading in appropriate blocks into internal memory, performing required operations and then writing them out to disk. Set $lr[\ell]$ to *nil* for all ℓ.

   (c) Otherwise; that is, if $(clevel \leq maxlevel)$, then

   (d) If $(lr[clevel] = nil)$, then

      i. Execute **download**$(clevel)$.

      ii. (Here, *mem* may have changed from its value at the beginning of Step 2(d)i.) If $enough(mem)$ is false then relinquish what's left of the ongoing allocation phase.

      iii. GoTo Step 2a.

   (e) Otherwise; that is, if $(lr[clevel] \neq nil)$, then

      i. While $(enough(mem)$AND$lr[clevel] \neq nil$, execute **llmult**$(clevel)$.

      ii. If $enough(mem)$ is false then relinquish what's left of the ongoing allocation phase.

      iii. GoTo Step 2a.

For analysis, it is convenient to define the call **llmult**$(maxlevel + 1)$, although **llmult**(ℓ) is only define for the case $\ell \leq maxlevel$.

Definition 46 We define the computation involved in the execution of Step 2b above to be the computation corresponding to **llmult**$(maxlevel+1)$. Thus, **llmult**$(maxlevel + 1)$ is said to be executed when (and if) Step 2b above is executed.

Abusing notation, we call this procedure **llmult**$(maxlevel + 1)$.

13.6 Resource Consumption Analysis of MAMultiply

We will now prove that the algorithm **MAMultiply** computes the matrix multiplication operation $\hat{C}+ \leftarrow \hat{A}\hat{B}$, where each of the three submatrices consist of $\sqrt{\hat{m}_{max}} \times \sqrt{\hat{m}_{max}}$ blocks each, incurring a resource consumption of no more than $O(\hat{m}_{max}^{3/2})$. We first prove that the resource consumption in the **download**( ) expense account is $O(\hat{m}_{max}^{3/2})$ by combining bounds on the number of **download**( ) operations with bounds on the amount of resource consumption of individual **download**( ) operations. The fact that the resource consumption of all **llmult**( ) operations is $O(\hat{m}_{max}^{3/2})$ follows from the observation that the $O(4^{3\ell/2})$ resource consumption during the execution of **llmult**(ℓ) is charged to the $\Omega((4^{\ell}B)^{3/2})$ pebbling operations performed during **llmult**(ℓ).

13.6.1 Resource consumption of download( ) operations

We begin with the definition of a certain type of **download**( ) operation sequence followed by a couple of key lemmas about such sequences.

Definition 47 Consider a sequence of d consecutive **download**(ℓ_i) operations, where $0 \leq i \leq d-1$ and $d \geq 1$, that satisfy the conditions mentioned in Lemma 45 and let ℓ be as defined in Lemma 45. We call the above sequence of d **download**( ) operations an ℓ, d-sequence if the first time **MAMultiply** either relinquishes I/O operations (in Step 2(d)ii above) or executes an **llmult**( ) operation after it executes **download**(ℓ_0) is only immediately after it executes **download**(ℓ_{d-1}).

Lemma 48 *Consider any ℓ, d-sequence* **download**(ℓ_0), ..., **download**(ℓ_{d-1}). *If, at any time after* **download**(ℓ_0) *begins and before* **download**(ℓ_{d-1}) *ends,* **MAMultiply** *is subjected to an allocation phase of size m_h such that $level(m_h) \geq \ell/2 + 1$, then the execution of* **download**(ℓ_{d-1}) *is immediately followed (that is, without relinquishing any I/Os) by the execution of* **llmult**(ℓ_h)*, where $\ell_h = \min\{level(m_h), maxlevel + 1\}$.*

Proof: From Lemma 45, we know that the total number of I/O operations required over the entire ℓ, d-sequence is no more than $c_{dl} \cdot 2^{\ell-1}$. Suppose that after **download**(ℓ_0) begins execution and before **download**(ℓ_{d-1}) completes execution, **MAMultiply** gets an allocation phase of size m_h where m_h is as defined above. Then even if all the $c_{dl} \cdot 2^{\ell-1} \leq c_{dl} \cdot 4^{\ell/2}$ I/O operations incurred during the ℓ, d-sequence occurred during the phase of size m_h, that allocation phase is still left with $c_{llm} \cdot 4^{level(m_h)-1}$ pending I/O operations. Hence on completion of the ℓ, d-sequence, $enough(level(m_h))$ evaluates to true: This follows from the definition of c_{level} above. Furthermore, by definition of ℓ, d-sequence, the dth **download**( ) operation **download**(ℓ_{d-1}) of the ℓ, d-sequence cannot be immediately followed by another **download**( ) operation so **llmult**(ℓ_h) is the next operation executed by **MAMultiply**. Thus the lemma is proved. □

We state now a simple corollary of the above lemma.

Corollary 4 *If an ℓ, d-sequence is followed by the execution of Step 2(d)ii, the total number of I/O operations relinquished is no more than $O(4^{l/2})$.*

13.6.2 Charging Scheme for download( ) Operations

Each ℓ, d-sequence incurs a certain amount of resource consumption. We use the following charging scheme to account for the of resource consumption of ℓ, d-sequences:

1. In the event that the ℓ, d-sequence is followed by **llmult**(ℓ_h), where $\ell_h > l/2$, we charge the resource consumption of the ℓ, d-sequence to the **llmult**(ℓ_h) operation.

2. In the event that the ℓ, d-sequence is followed by the execution of Step 2(d)ii or by the execution of an **llmult**(ℓ'_h) operation, where $\ell'_h \leq \ell/2$, we account for its resource consumption using Lemma 48 and Corollary 4.

We first count the maximum number of ℓ, d-sequences that can occur during **MAMultiply**.

Lemma 49 *The total number of times an ℓ, d-sequence can occur during the entire execution of* **MAMultiply** *is $8^{\ell_{max}-\ell}$, where $1 \leq \ell \leq \ell_{max} - 1$.*

Proof: This follows from the fact that each time $lr[\ell+1]$ is set to a non-*nil* value, the maximum number of ℓ, d-sequences that can occur before $lr[\ell+1]$ becomes *nil* is 8. □

We will now bound the total resource consumption of all **download**( ) operations, barring those involved in ℓ, d-sequences whose resource consumption is charged to **llmult**( ) operations.

Theorem 14 *Suppose that the resource consumption of an an ℓ, d-sequence is the resource consumption during I/O operations incurred during the ℓ, d-sequence* and *the resource consumption corresponding to the I/O operations relinquished by the (possible) execution of Step 2(d)ii immediately following the ℓ, d-sequence. The total resource consumption of all ℓ, d-sequences, except any ℓ, d-sequence whose resource consumption we charge in Step 1 of Section 13.6.2 to an* **llmult**(ℓ_h) *operation with $\ell_h > l/2$, is no more than $O(\hat{m}_{max}^{3/2})$.*

Proof: By Lemma 49, the total number of ℓ, d-sequences that can occur is $8^{\ell_{max}-\ell}$. Including the I/O operations that are possibly relinquished on account of executing Step 2(d)ii immediately after the ℓ, d-sequence, the total number of I/O operations charged to the ℓ, d-sequence is no more than $O(4^{\ell/2})$, by Lemma 45 and Corollary 4. Also, by Lemma 48, the maximum allocation level during any of these $O(4^{l/2})$ I/O operations is no more than $\ell/2$, implying that the maximum resource consumption of each ℓ, d-sequence relevant to this theorem is no more than $O((4^{\ell/2})^{3/2})$.

Hence the total amount of resource consumption that can be charged to all ℓ, d-sequences is

$$\begin{aligned}
\sum_{l=1}^{\ell_{max}-1} 8^{\ell_{max}-\ell} \cdot O((2^{\ell})^{3/2}) &= O(8^{\ell_{max}} \sum_{l=1}^{\ell_{max}-1} 2^{3\ell/2}/2^{3\ell}) \\
&= O(8^{\ell_{max}} \sum_{l=1}^{\ell_{max}-1} 1/2^{3\ell/2}) \\
&= O(8^{\ell_{max}}),
\end{aligned}$$

which is $O(\hat{m}_{max})$ since $8^{\ell_{max}} = (4^{3/2})^{\ell_{max}} = (4^{\ell_{max}})^{3/2} = \hat{m}_{max}^{3/2}$. □

13.6.3 Resource Consumption of llmult( ) Operations

We argue here that the total amount of resource consumption that can be charged to an **llmult**(ℓ) operation is no more than $O((4^\ell)^{3/2})$ while the number of pebbling operations accomplished is at least $\Omega((4^\ell B)^{3/2})$.

We first have the following lemma implying that our reorganization of the operation $\hat{C}' := \hat{C}' + \hat{A}'\hat{B}'$, where each of $\hat{A}'$, $\hat{B}'$, and $\hat{C}'$ are submatrices consisting of $2^\ell \times 2^\ell$ blocks, into 8 subproduct operations $\hat{C}'_{i,j} := \hat{C}'_{i,j} + \hat{A}'_{i,k}\hat{B}'_{j,k}$ corresponding to the 8 $0-1$-triples is correct.

Lemma 50 *The matrix operation $\hat{C}'+ \leftarrow \hat{A}'\hat{B}'$ is correctly implemented by the 8 operations $\hat{C}'_{i,j} := \hat{C}'_{i,j} + \hat{A}'_{i,k}\hat{B}'_{j,k}$ corresponding to the 8 (i,j,k) $0-1$-triples.*

It can be inductively proved that the number of pebbling operations performed by **MAMultiply** using the above approach is no more than $(\hat{m}_{max}B)^{3/2}$. Now we consider the maximum resource consumption that can be charged to a single **llmult**(ℓ) operation.

Lemma 51 *The maximum resource consumption that can be charged to a single* **llmult**(ℓ) *operation, where $1 \le \ell \le$ maxlevel, is $O((4^\ell)^{3/2})$.*

Proof: By Lemmas 46 and 47, the total number of I/O operations incurred by an **llmult**(ℓ) operation is $O(4^\ell)$ for any ℓ, including $\ell = maxlevel + 1$. The total number of I/O operations relinquished due to possibly executing Step 2(e)ii immediately after the **llmult**(ℓ) operation is $O(4^\ell)$. If $1 \le \ell \le$ *maxlevel* the allocation level throughout the above I/O operations is ℓ so that the resource consumption is $O((4^\ell)^{3/2})$. On the other hand, the total number of I/O operations incurred by the ℓ', d-sequence, where $\ell' \le 2\ell - 2$, whose resource consumption we possibly charge in Step 1 of Section 13.6.2 to **llmult**(ℓ) is no more than $O(4^\ell)$, by Lemma 45. The maximum allocation level during these $O(4^\ell)$ I/O operations is ℓ, so their resource consumption is $O((4^\ell)^{3/2})$. This proves the lemma. □

We now bound the total resource consumption charged to all the **llmult**( ) operations incurred during *mamultiply* by $O(\hat{m}_{max}^{3/2})$.

Theorem 15 *The total resource consumption charged to all the* **llmult**( ) *operations incurred during mamultiply by $O(\hat{m}_{max}^{3/2})$.*

Proof: First we consider all **llmult**(ℓ) operations with $\ell < maxlevel + 1$. By Lemma 51, we know that each **llmult**(ℓ) operation can be charged a resource consumption of no more than $O((4^\ell)^{3/2})$. By definition, each **llmult**(ℓ) operation performs $\Omega((4^\ell B)^{3/2})$ pebbling operations. Thus at any time during **MAMultiply**, $B^{3/2}$ times the total resource consumption charged to **llmult**( ) operations up to that point of time is always of the order of the total number of pebbling operations performed by **MAMultiply** up to that point. Since **MAMultiply** performs no more than $(\hat{m}_{max}B)^{3/2}$ pebbling operations, the lemma holds for all **llmult**(ℓ) operations with $\ell < maxlevel + 1$.

There can be at most one **llmult**(ℓ) operations with $\ell \geq maxlevel$ and such an operation incurs $O(\hat{m}_{max})$ I/O operations so its resource consumption is also $O(\hat{m}_{max}^{3/2})$. This proves the lemma. □

13.7 Proving Optimality

Theorem 14 and Theorem15 together imply the following lemma bounding the total resource consumption of **MAMultiply**.

Theorem 16 *The total resource consumption of* **MAMultiply** *is no more than* $O(\hat{m}_{max}^{3/2})$.

Since the total number of **MAMultiply** operations involved is the same as the number of times Step 2, which is $(n/\hat{m}_{max})^{3/2}$, Theorem 16, Corollary 1 and the definition of dynamic optimality implies the following theorem.

Theorem 17 *The total amount of resource consumption of our memory-adaptive matrix multiplication algorithm when used to multply two n block matrices is* $O(n^{3/2})$. *Our memory-adaptive matrix multiplication algorithm is dynamically optimal.*

14 Conclusions and Future Work

In this paper we have presented a simple and reasonable dynamic memory allocation model that enables database and operating systems to dynamically change the amount of memory that external memory algorithms are allocated. We have defined what it means for memory-adaptive external memory algorithms designed to work in this model to be dynamically optimal. We have presented dynamically optimal memory-adaptive algorithms for fundamental problems such as sorting, problems related to sorting, permuting, other problems related to sorting and matrix multiplication. We have also presented a dynamically optimal (in an amortized sense) version of the buffer tree, which has a large number of batched dynamic applications and applications such as "bulk-loading" of external memory data structures. We have shown that a previously devised approach to memory-adaptive external mergesort is provably nonoptimal due to funadmental drawbacks. The lower bound proof techniques for sorting and matrix multiplication are the two fundamentally distinct proof techniques invoked by most other external memory lower bounds and hence we anticipate that the techniques presented here will apply to many external memory problems.

In the case of mergesorting and matrix multiplication, we have shown how to sow together a conventional external memory algorithm with an appropriate "memory-adaptivity" data structure that balances work across "levels of allocation " to obtain a memory-adaptive external memory algorithm. Our proof techniques deal with the interesting constraints faced while proving optimality of resource consumption in our dynamic memory model.

We believe that our techniques for memory-adaptive merging apply to memory-adaptive distribution and thus to a dynamically optimal distribution sort. Since the $BatchMerge_K(\ )$ operation used in [BK98] can be performed using a modification of our memory-adaptive merging technique, we

conjecture that we can design a dynamically optimal memory-adaptive version of the worst-case optimal external priority queue of [BK98]. It would be fruitful to extend our approach to other domains and applications. An interesting question is whether or not we can devise a general technique that takes any external memory algorithm that is optimal for static memory and convert it into a dynamically optimal memory-adaptive algorithm.

References

[AHVV98] Lars Arge, Klaus Hinrichs, Jan Vahrenhold, and Jeffirey S. Vitter. Efficient bulk operations on dynamic r-trees. *Submitted*, 1998.

[AKL93] L. Arge, M. Knudsen, and K. Larsen. A general lower bound on the I/O-complexity of comparison-based algorithms. In *Proceedings of the 3rd Workshop on Algorithms and Data Structures*, volume 709, pages 83–94. Lecture Notes in Computer Science, Springer-Verlag, 1993.

[Arg94] Lars Arge. The buffer tree: A new technique for optimal I/O-algorithms. Technical Report RS-94-16, BRICS, Univ. of Aarhus, Denmark, 1994.

[Arg96] Lars Arge. *Efficient External-Memory Data Structures and Applications*. PhD thesis, Department of Computer Science, University of Aarhus, 1996.

[AV88] Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[AV96] L. Arge and J. S. Vitter. Optimal interval management in external memory. *Proc. of the 37th Annual IEEE Symposium on Foundations of computer Science (FOCS '96)*, pages 560–569, October 1996. Also appeared in Abstracts of the First CGC Workshop on Computational Geometry, Center for Geometric Computing, Johns Hopkins university, Baltimore, MD, October 1996.

[BK98] G. S. Brodal and J. Katajainen. Worst-case efficient external-memory priority queues. *Scandinavian Workshop on Algorithmic Theory*, 1998.

[CGG+95] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1995.

[GTVV93] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *IEEE Foundations of Computer Science*, pages 714–723, 1993.

[HK81] J. W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. *Proc. 13th Annual ACM Symp. on Theory of Computation*, pages 326–333, may 1981.

[Knu97] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading Ma., third edition, 1997.

[Knu98] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, second edition, 1998.

[PCL93] H. Pang, M. Carey, and M. Livny. Memory-adaptive external sorts. *Proc. Nineteenth International Conf. on Very Large Data Bases*, 1993.

[SV87] J. E. Savage and J. S. Vitter. Parallelism in space-time tradeoffs. In F. P. Preparata, editor, *Advances in Computing Research, Volume 4*, pages 117–146. JAI Press, 1987.

[Vit98] Jeffrey S. Vitter. External memory algorithms. *Proceedings of the 17th Annual ACM Symposium on Principles of Database Systems (PODS '98)*, pages 119–178, 1998.

[VS94] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.

[VV95] Darren Erik Vengroff and Jeffrey Scott Vitter. I/O-efficient scientific computation using TPIE. Technical Report CS–1995–18, Duke University Dept. of Computer Science, 1995.

[WGWR93] D. Womble, D. Greenberg, S. Wheat, and R. Riesen. Making parallel computer i/o practical. *Proceedings of the 1993 DAGS/PC Symposium*, pages 56–63, June 1993.

[ZL97] W. Zhang and P.-A. Larson. Dynamic memory adjustment for external mergesort. *Proc. Twenty-third International Conf. on Very Large Data Bases*, 1997.