Cache-Adaptive Algorithms

Michael A. Bender[†‡] Roozbeh Ebrahimi[†] Jeremy T. Fineman[§] Golnaz Ghasemiesfeh[†]
Rob Johnson[†] Samuel McCauley[†]

Abstract

We introduce the cache-adaptive model, which generalizes the external-memory model to apply to environments in which the amount of memory available to an algorithm can fluctuate. The cache-adaptive model applies to operating systems, databases, and other systems where the allocation of memory to processes changes over time.

We prove that if an optimal cache-oblivious algorithm has a particular recursive structure, then it is also an optimal cache-adaptive algorithm. Cache-oblivious algorithms having this form include Floyd-Warshall all pairs shortest paths, naïve recursive matrix multiplication, matrix transpose, and Gaussian elimination. While the cache-oblivious sorting algorithm Lazy Funnel Sort does not have this recursive structure, we prove that it is nonetheless optimally cache-adaptive. We also establish that if a cache-oblivious algorithm is optimal on "square" (well-behaved) memory profiles then, given resource augmentation it is optimal on all memory profiles.

We give paging algorithms for the case where the cache size changes dynamically. We prove that LRU with 4-memory and 4-speed augmentation is competitive with optimal. Moreover, Belady's algorithm remains optimal even when the cache size changes.

Cache-obliviousness is distinct from cache-adaptivity. We exhibit a cache-oblivious algorithm that is not cache-adaptive and a cache-adaptive algorithm for a problem having no optimal cache-oblivious solution.

Keywords and phrases cache adaptive, cache oblivious, external memory, memory adaptive, paging algorithms.

This research was supported in part by NSF grants CCF 1114809, CCF 1217708, CCF 1218188, IIS 1247726, and IIS 1251137, and by Sandia National Laboratories.

[†]Department of Computer Science, Stony Brook University, Stony Brook, NY 11794-4400, USA. Email: {bender,rebrahimi,gghasemiesfe,rob,smccauley}@cs.stonybrook.edu.

[‡]Tokutek, Inc. www.tokutek.com.

[§]Department of Computer Science, Georgetown University, 37th and O Streets, N.W., Washington D.C. 20057, USA. Email: jfineman@cs.georgetown.edu.

1 Introduction

For over two decades, practitioners have recognized the desirability of algorithms that adapt as the availability of RAM changes [3,4,15,34,35]. Such changes naturally occur as processes come and go on the same system and compete with each other for memory resources. Database researchers, in particular, have published empirically efficient adaptive sorting and join algorithms [24,36,37,47–49].

There exists a theoretical framework for designing algorithms that adapt to memory fluctuations [3,4], but there has been essentially no theoretical followup work. Moreover, the empirically efficient adaptive algorithms do not have theoretical performance bounds and are not commonly used in today DBMSs, even though the practical need for such algorithms has, if anything, increased. We attribute this lacuna to the difficulty of designing, analyzing, and implementing adaptive algorithms.

One of the main goals of this paper is to show that adaptivity is sometimes achievable via more manageable algorithms, by leveraging cache-oblivious technology [23, 40]. ***Cache-oblivious*** algorithms are not parameterized by the memory hierarchy, yet they often achieve provably optimal performance for any *static* hierarchy. The question is how these algorithms *adapt* when the memory changes dynamically. We prove that there exist a general class of cache-oblivious algorithms that are optimally ***cache-adaptive***.[1]

We define the ***cache-adaptive*** (CA) model, an extension of the DAM [1][2] and ***ideal cache*** [23, 40] models. The CA model describes systems in which the available memory can change dynamically.

Barve and Vitter [3] write, "Allowing an algorithm to be aware of its internal memory allocation is obviously necessary to develop MA [memory adaptive] algorithms. . . ." This quote rings true if the algorithm and page-replacement policies are inextricably linked. But if we can separate the algorithm and the page replacement, as is typical with cache-oblivious analysis [23], this claim no longer holds.

The cache-adaptive model does, in fact, untangle the paging policy from the algorithm. We assume an *automatic*

[1]We use "cache" to refer to the smaller level in any two-level hierarchy. Because this paper emphasizes RAM and disk, we use the terms "internal memory," "RAM," and "cache" interchangeably.

[2]Also called the external-memory (EM) or I/O model.

optimal page replacement and justify it by proving competitiveness of LRU in the CA model. This separation helps us achieve CA optimality for CO algorithms, which are unaware of their memory allocations.

Memory Models. In the DAM model [1], the system consists of a two-level memory hierarchy, comprising an internal memory of size M and an external memory that is arbitrarily large. Data is transferred between the memory and disk in chunks of fixed size B. A DAM algorithm manages its own page replacement, and the internal memory is fully associative. In the DAM model, in-memory computation comes for free, and performance is measured by the number of I/Os or block transfers, which serves as a proxy for running time. Thus, in each ***time step***[3] of the algorithm one block is transferred between memory and disk.

Barve and Vitter generalize the DAM model to allow the memory size M to change periodically.[4] Their optimal algorithms (sorting, FFT, matrix multiplication, LU decomposition, permutation, and Buffer trees) are impressive and technically sophisticated. The present paper considers a similar model, although we allow M to change more rapidly and unpredictably.

A cache-oblivious algorithm [23] is analyzed using the ***ideal-cache model*** [23, 40], which is the DAM model augmented to include *automatic, optimal* page replacement. The algorithm is not parameterized by M or B, but it is analyzed in terms of M and B. The beauty of this restriction is that an optimal cache-oblivious algorithm is simultaneously optimal for any fixed choice of M and B.

Automatic replacement is necessary because the algorithm has no knowledge of B or M. The optimality assumption is justified since the Least Recently Used (LRU) page replacement policy is constant-competitive with optimal with resource augmentation [43].

Results. We prove that a class of recursive cache-oblivious algorithms are optimally cache-adaptive (Section 4). This class includes cache-oblivious naïve matrix multiplication, matrix transpose, Gaussian elimination, and Floyd-Warshall all-pairs shortest paths algorithms. We also establish that the ***lazy funnel sort*** algorithm [12], which falls outside this class, is optimally cache-adaptive (Subsection 4.4).

We prove that LRU with 4-memory and 4-speed augmentation is competitive with the optimal page replacement policy in the cache-adaptive model (Section 5). We show that Belady's Longest Forward Distance policy is an optimal offline page replacement policy in the cache-adaptive model (Section 5).

We establish a separation between cache-adaptivity and cache-obliviousness by showing that: (1) There exists a cache-adaptive algorithm for permutation, for which no cache-oblivious algorithm can be devised [13], and (2) there exist cache-oblivious algorithms that are not cache-adaptive (Section 6).

2 Cache-Adaptive Model

The ***cache-adaptive*** (***CA***) model extends the DAM and ideal-cache models. As with the DAM model, computation is free and the performance of algorithms is measured in terms of I/Os. As in the ideal-cache model, the system manages the content of the cache automatically and optimally. But in the CA model, the cache size is not fixed; it can change during an algorithm's execution.

The ***memory profile*** is the function $m : \mathbb{N} \to \mathbb{N}$, which indicates the size, in blocks, of the cache at the time of the tth I/O.[5] The ***memory profile in bytes*** is the function $M(t) = Bm(t)$.

We place no restrictions on how the memory-size changes from one I/O to the next (unlike previous approaches [3]). However, since at most one block can be transferred at each step, we assume without loss of generality that the increases are bounded by

$$m(t+1) \leq m(t) + 1. \tag{2.1}$$

When the size of memory decreases, a large number of blocks may need to be written back to disk, depending on whether the blocks are dirty. In this paper, we do not charge these write-backs to the application, but we believe that the model can be extended to do so.

Optimality in the CA Model. Analyses in the DAM and ideal cache models often use the notion of ***competitiveness***. Roughly speaking, algorithm A is competitive with A' if A's running time is always within a constant factor of A''s [43]. In the DAM model, memory cannot change in size. So to achieve competitiveness, it is equivalent to give algorithm A extra time to complete or extra speed, so that it can complete at the same time as A'.

In contrast, in the CA model, not all timesteps have the same memory size, which means that we need to refine what we mean by competitive. We disambiguate by giving extra speed. That is, if A has ***c-speed augmentation*** compared to A', then when A performs its ctth I/O, A' performs its tth I/O. (The alternative notion of giving A extra time appears to be a dead end. If the available memory drops precipitously right after A' completes, then A may finish arbitrarily later than A'; see Lemma 5.2.)

[3] Often when we write external-memory and cache-oblivious algorithms, we avoid the word "time" and express performance in terms of I/Os. In the cache-adaptive context, we prefer to use the word "time step" explicitly.

[4] They require that memory stays constant for enough I/Os to refill the cache, after which it can change arbitrarily.

[5] Throughout, we use the terms *block* and *page* interchangeably.

DEFINITION 2.1. (***speed augmented profiles***) *If m is any memory profile, under* $\mathbf{c_1}$***-speed augmentation*** *m is scaled into the profile $m'(t) = m(\lfloor t/c_1 \rfloor)$.*

DEFINITION 2.2. *An algorithm A that solves problem P is* ***optimal*** *in the cache-adaptive model if there exist a constant c such that on all memory profiles and all sufficiently large input size N, the worst-case running time of a c-speed augmented A is better than the worst-case running time of any other (non-augmented) algorithm.*

An algorithm with $\mathbf{c_2}$***-memory augmentation*** compared to A' has c_2 times more memory than A'.

DEFINITION 2.3. (***memory augmented profiles***) *If m is any memory profile, under* $\mathbf{c_2}$***-memory augmentation*** *m is scaled into the profile $m'(t) = c_2 m(t)$.*

Justification for the CA Model. The CA model is intended to capture performance on systems in which memory size changes asynchronously in response to external events, such as the start or end of other tasks. Since these events are asynchronous, changes in memory size should be pegged to wall-clock time.

We construct the CA model as an extension of the DAM model, i.e., the DAM model is the special case that $m(t)$ is static. The DAM model, however, has no explicit notion of time. Instead, performance is measured by the number of I/Os. This I/O counting can be reinterpreted as time, with an I/O taking unit time and computation taking 0 time. On real systems, I/Os dominate computation, so the number of I/Os is a good approximation to wall-clock time for I/O-bound algorithms.

In the CA model, we explicitly measure time in terms of I/Os. This is the same approach adopted by Barve and Vitter [3]. For example, the arrival of a new process at a particular time can be modeled by having memory drop after a certain number of I/O steps.

Peserico [39] uses an alternative model for a cache whose size changes dynamically. In this model, the changes to memory size are specified by '+' or '-' symbols in a page-request sequence, representing an increase or decrease to cache size, respectively. In contrast to our model, these changes are not linked to time, but to specific page requests. For example, in Peserico's model, one can specify that the cache size should change after instruction 52 of the program, whereas in our model, one can specify that the cache size should change after time step 10. While both models are natural, the CA model is more applicable to our motivating application, i.e., size changing due to external events.

Optimal Page Replacement in the CA Model. Analogous to the justification of optimal page replacement in the ideal-cache model [23], we show that LRU in the CA model can simulate OPT. We prove that LRU with 4-memory and 4-speed augmentation always completes sooner than OPT (Theorem 5.1). Although Peserico [39] proves similar results, the model is drastically different, and hence those results do not apply here.

3 Toolbox for Cache-Adaptive Analysis

We first introduce a class of memory profiles called ***square profiles***. Proving cache-adaptive optimality on square profiles is cleaner. Moreover, optimality on square profiles is extendable to all profiles.

DEFINITION 3.1. *A memory profile m is* ***square*** *if there exist boundaries $0 = t_0 < t_1 < \dots$ such that for all $t \in [t_i, t_{i+1})$, $m(t) = t_{i+1} - t_i$. In other words, a square memory profile is a step function where each step is exactly as long as it is tall (see Figure 1).*

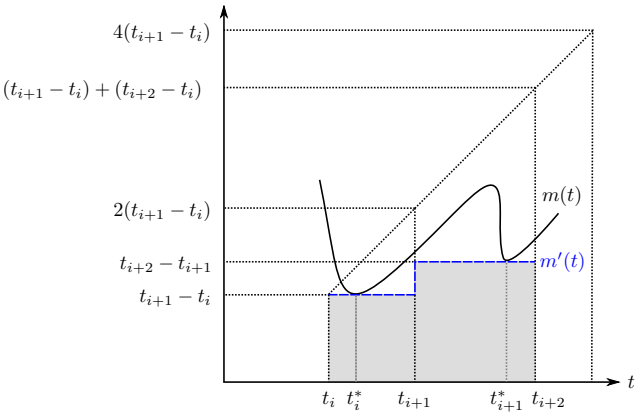

Figure 1: The inner square profile of the memory profile $m(t)$. The inner square boundaries comprise t_i, t_{i+1}, and t_{i+2}. The figure also illustrates the proof of Lemma 3.1.

We define an ***inner square profile*** m' for a profile m by placing maximal squares below m, proceeding left to right, as illustrated in Figure 1. The ***inner square boundaries*** are the left/right boundaries of the squares in m'.

DEFINITION 3.2. *For a memory profile m, the* ***inner square boundaries*** *$t_0 < t_1 < t_2 < \dots$ of m are defined as follows: Let $t_0 = 0$. Recursively define t_{i+1} as the largest integer such that $t_{i+1} - t_i \le m(t)$ for all $t \in [t_i, t_{i+1})$. The* ***inner square profile*** *of m is the profile m' defined by $m'(t) = t_{i+1} - t_i$ for all $t \in [t_i, t_{i+1})$.*

The following lemma enables us to analyze algorithms even in profiles where the memory size drops precipitously. Intuitively, the lemma states that the $(i+1)$st interval in the inner square profile is at most twice as long as the ith interval, and the available memory in the original memory

profile during the $(i+1)$st interval is at most four times the available memory in the ith interval of the inner square profile.

LEMMA 3.1. *Let m be a memory profile where $m(t+1) \leq m(t)+1$ for all t. Let $t_0 < t_1 < \ldots$ be the inner square boundaries of m, and let m' be the inner square profile of m.*

1. *For all t, $m'(t) \leq m(t)$.*
2. *For all i, $t_{i+2} - t_{i+1} \leq 2(t_{i+1} - t_i)$.*
3. *For all i and $t \in [t_{i+1}, t_{i+2})$, $m(t) \leq 4(t_{i+1} - t_i)$.*

Proof.
1. By construction of m' in Definition 3.2.

2. By construction of the inner square boundaries in Definition 3.2, for each i there exists a $t_i^* \in [t_i, t_{i+1}]$ such that $m(t_i^*) = m'(t_i) = t_{i+1} - t_i$. Since m can only increase by one in each time step, $m(t_{i+1}) \leq m(t_i^*) + (t_{i+1} - t_i^*)$. Substituting for $m(t_i^*)$ and because $t_i^* \geq t_i$, we obtain $m(t_{i+1}) \leq 2(t_{i+1} - t_i)$. Also by construction the inner square boundaries, $t_{i+2} - t_{i+1} \leq m(t_{i+1})$, we must have $t_{i+2} - t_{i+1} \leq 2(t_{i+1} - t_i)$.

3. Similarly, for all $t \in [t_{i+1}, t_{i+2})$, $m(t) \leq m(t_i^*) + (t - t_i^*) \leq (t_{i+1} - t_i) + (t_{i+2} - t_i) = (t_{i+1} - t_i) + (t_{i+2} - t_{i+1}) + (t_{i+1} - t_i)$. Hence we get $m(t) \leq 4(t_{i+1} - t_i)$. ∎

3.1 Lower Bounds in Cache-Adaptive Model. The following definition explains how we treat lower bounds in the CA model.

DEFINITION 3.3. (***piecewise lower bound for problem P***) *A problem P has a piecewise lower bound if there exists **work** function W and **progress bound** U as follows:*

1. Work function $W(N) : \mathbb{N} \to \mathbb{N}$ is the minimum amount of work needed to solve a worst case instance of size N on problem P, where work is a problem-specific notion of what it means for P to be solved.

2. Progress function $U(M, B, N) : \mathbb{N}^3 \to \mathbb{N}$ is an upper bound on the amount of work that can be completed using a memory of size M and M/B I/Os.

3. Let m be any square profile with interval boundaries $t_0 < t_1 < \ldots < t_n$. If

$$\sum_{i=0}^{n-1} U(M(t_i), B, N) < W(N), \tag{3.2}$$

then there does not exist an algorithm that solves all instances of P of size at most N under profile m.

Definition 3.3 serves as a tool to port lower bounds in the DAM model to the CA model. A lower bound in the DAM model can be used in Definition 3.3 if it is ***memoryless***, which means that an upper bound on the progress in an interval does not depend on the progress in previous intervals.

We give an example. The lower bound for naïve matrix multiplication [42] uses a version of the *red-blue pebble game* [26] on the matrix multiplication DAG. There are $W(N) = \Theta(N^{3/2})$ nodes in the DAG to be pebbled. Regardless of the progress in the past, the maximum number of DAG nodes that can be pebbled using M/B I/Os when memory has size M is $U(M, B, N) = \mathrm{O}(\mathrm{M}^{3/2})$. Thus, naïve matrix multiplication has a piecewise lower-bound in the CA model.[6]

3.2 Recursion in the Cache-Adaptive Model. Recursive cache-oblivious algorithms have base cases of constant size. In contrast, their I/O complexity is expressed by a recurrence, where the base case is a function of M or B.

The recurrence "bottoms out" at nodes in the recursion tree with input size at most M. This is because once a subproblem is brought fully into cache, subsequent recursive calls do not incur I/Os. We refer to such nodes as ***bottomed-out nodes***; see Figure 2. The number of block transfers needed to complete a bottomed-out node is usually linear in the input size of the node.

Bottomed out nodes in a recursion tree are at the same depth (as long as the tree has a regular structure) since M is fixed. In contrast, in the CA model, where the cache-size changes over time, the height of bottomed-out nodes can vary; see Figure 3. The running time of a recursive algorithm in the CA model is influenced by the height of these bottomed-out nodes in different periods of time.

3.3 Fractional Progress. We analyze algorithms in terms of their ***progress*** towards completion. The lower bound and upper bound on progress often have different units.[7] We remove the units from the progress measures and use ***fractional progress (denoted by FP)*** instead, defined as the percentage of the task completed in a period of time.

Upper bound on fractional progress of any algorithm. We describe the maximum possible fractional progress of any algorithm on an interval $\mathcal{I}$ of length M/B I/Os using M memory. By Definition 3.3, in M/B I/Os, any algorithm $\mathcal{C}$ can make at most $U(M, B, N)$ progress and the total work is $\geq W(N)$. Hence,

$$\mathrm{FP}_{\mathcal{C}}(\mathcal{I}) \leq \frac{U(M, B, N)}{W(N)}. \tag{3.3}$$

Lower bound on fractional progress of a recursive algorithm. We bound the fractional progress of a recursive al-

[6] Definition 3.3 is stated in additive form, where the progress U should at least add up to W. If a lower bound is stated in multiplicative form, then take logarithms.

[7] For example, the sorting lower bound measures the reduction in permutations in the input. An algorithm, such as merge sort, does not.

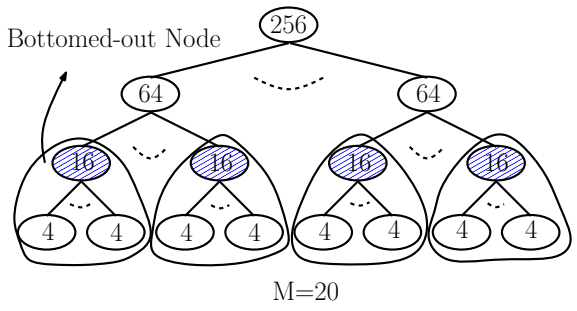

Figure 2: Bottomed-out nodes in the cache-oblivious analysis.

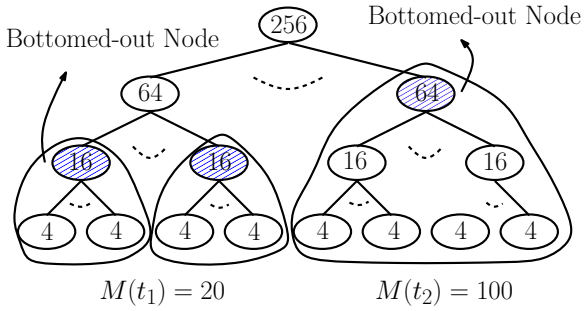

Figure 3: Bottomed-out nodes in the cache-adaptive model.

gorithm in an interval $\mathcal{I}$ in terms of completion of bottomed-out nodes. Let the I/O complexity of the recursive algorithm $\mathcal{A}$ be upper bounded by the function $T(N, M, B)$ in the DAM model. Assume that $M(t)_{t\in\mathcal{I}} \geq M_1$ and the ***exact*** cost of a bottomed-out node of size N_1 is $\text{Cost}(N_1, M, B)$. Hence,

$$\text{FP}_{\mathcal{A} \text{ completing } N_1}(\mathcal{I}) \geq \frac{\text{Cost}(N_1, M_1, B)}{T(N, M_1, B)}. \tag{3.4}$$

4 A Class of Optimal Cache-Adaptive, Cache-Oblivious Algorithms

In this section, we define a class of cache-oblivious algorithms that are optimal in the cache-adaptive model. We present the proof of optimality in two steps. First, we prove that this class is optimal for all *square profiles*; then we show that we can extend the optimality to all memory profiles.

DEFINITION 4.1. *A recursive algorithm $\mathcal{A}$ is in **constant overhead recursive (COR)** form, if there exist monotone functions $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$ satisfying $f(N) \geq 2$ and $1 \leq g(N) < N$ and positive constants $\alpha < 1$ and L_B (depending on B), such that the following holds:*

1. On input of size N, $\mathcal{A}$ recursively calls itself $f(N)$ times on subproblems of size at most $g(N)$. Besides calling itself, the algorithm accesses only $O(1)$ pages. For sufficiently small N, the algorithm accesses only $O(1)$ pages.

2. Let $M \geq L_B$, where L_B is a problem-specific function of B. The I/O complexity of the $\mathcal{A}$ in the DAM model can be bounded by

$$T(N, M, B) = \begin{cases} f(N)T\big(g(N), M, B\big) + O(1) & N > \alpha M \\ \Theta\left(N/B + 1\right) & N \leq \alpha M. \end{cases} \tag{4.5}$$

The term L_B captures tall-cache assumptions needed by some algorithms. For example, for matrix multiply, $L_B = B^2$, but for linear scan, which needs no tall-cache assumption, $L_B = 1$.

Optimality in the DAM model. In the DAM model, an algorithm is optimal with respect to work lower bound $W(N)$ and progress upper bound $U(M, B, N)$ if there exists a constant c such that for all $M \geq L_B$ and $N \geq M$,

$$T(N, M, B) \leq c \left\lceil \frac{M}{B} \right\rceil \left\lceil \frac{W(N)}{U(M, B, N)} \right\rceil. \tag{4.6}$$

To prove that cache-oblivious algorithms in COR form are optimal in the cache-adaptive model, we first work on square profiles, where we can apply the above idea on each square individually.

4.1 COR Optimality on Square Profiles.

To prove optimality of algorithms in COR form, we use the following lemma, which is proved in Appendix A.

LEMMA 4.1. *Assume a tree T of height H and branching factor greater than or equal to 2. Given any node x, consider the set of next z leaves, S_z, that appear after x in the DFS order of the tree. Let I_z be the set of internal nodes in the DFS order between x and the last node of S_z. Then $|I_z| \leq 3H + z$.*

THEOREM 4.1. *Given a problem P and a **piecewise lower bound** on P, consider an optimal cache-oblivious algorithm $\mathcal{A}$ in **COR** form. Let H be the depth of recursion of $T(N, L_B, B)$ and d be the additive $O(1)$ term in Eq. 4.5. The algorithm $\mathcal{A}$ is cache-adaptively optimal on all **square profiles**, $m(t)$, that satisfy*

$$\forall t,\ m(t) \geq \max\{L_B, 3dH\}.$$

Proof. Algorithm $\mathcal{A}$'s I/O complexity can be bounded by function T in the DAM model. In order to make the analysis simpler, we assume that the cost of each *bottomed-out* node for $\mathcal{A}$ is ***exactly equal*** to T. More formally, we assume $\mathcal{A}$ is implemented on a machine with a (sub-optimal) paging policy that flushes the cache before the execution of each bottomed-out node. Such an assumption would only make $\mathcal{A}$ slower and would not affect the intended optimality outcome.

Furthermore, let β be the constant in the base case of Eq. 4.5, i.e. when $N \leq \alpha M$, $T(N, M, B) \leq \beta(N/B + 1)$.

We compare the fractional progress of algorithm $\mathcal{A}$ (upper bound) with the fractional progress specified by the lower bound in any given square of the profile $m(t)$. For simplicity we refer to the fractional progress made by the lower bound as the actions of an algorithm called OPT.

Let $\mathcal{J}$ be any square of the profile $m(t)$ and let $m_1 = M_1/B$ be the length of $\mathcal{J}$. By definition, the available memory in $\mathcal{J}$ is M_1. Assume that $\mathcal{A}$ is $(2q+1)$-speed augmented, where $q > \lceil \max(2dc, d\alpha\beta, 1) \rceil$. By definition, there are $(2q+1)m_1$ I/Os available to $\mathcal{A}$ in $\mathcal{J}$.

We are going to count the number of bottomed-out nodes completed by the algorithm $\mathcal{A}$ in the interval $\mathcal{J}$. By the definition of the recursion of $\mathcal{A}$, a node of input size N_1 is ***bottomed-out*** iff $N_1 \leq \alpha M_1 < g^{-1}(N_1)$.

LEMMA 4.2. *Algorithm $\mathcal{A}$ completes at least $y = \lfloor qm_1/dT(N_1, M_1, B) \rfloor$ bottomed-out nodes in $\mathcal{J}$. Here, $d \geq 1$ is the $O(1)$ overhead term in Definition 4.1.*

Proof. The recursion follows a DFS order to reach and complete bottomed-out nodes. Assume that at beginning of $\mathcal{J}$, the algorithms is operating on an arbitrary node in the recursion tree and completes y bottomed-out nodes by the end of $\mathcal{J}$. Lemma 4.1 states that in between these bottomed-out nodes, we visit at most $3H + y$ internal nodes. Each internal node incurs at most d I/Os. We refer to the overall cost of internal nodes as ***overhead*** cost.

The algorithm $\mathcal{A}$ has $(2q+1)m_1$ available I/Os in $\mathcal{J}$. Set $y = \lfloor qm_1/dT(N_1, M_1, B) \rfloor$. Since $q \geq d\alpha\beta$, $y \geq 1$. Sum the *overhead* cost and the cost of bottomed-out node completions.

$$\underbrace{yT(N_1, M_1, B)}_{\text{Bottomed-out nodes}} + \underbrace{dy + 3dH}_{\text{Overhead cost}} \leq \frac{qm_1}{d} + \frac{qm_1}{T(N_1, M_1, B)} + \underbrace{m_1}_{\text{because } 3dH \leq m_1} \leq \underbrace{(2q+1)m_1}_{\text{because } d, T(N_1, M_1, B) \geq 1}.$$

Since the sum is less $(2q+1)m_1$, at least y bottomed-out nodes will be completed by $\mathcal{A}'$ in $\mathcal{J}$. ∎

Now we show that the fractional progress of $\mathcal{A}$ is bigger than the fractional progress of OPT in $\mathcal{J}$. The two progress terms are now as follows:

$$\text{FP}_{\mathcal{A}}(\mathcal{J}) \geq \frac{yT(N_1, M_1, B)}{T(N, M_1, B)} \qquad \text{(using Eq. 3.4)} \tag{4.7}$$

$$\text{FP}_{\text{OPT}}(\mathcal{J}) = \frac{U(M_1, B, N)}{W(N)} \qquad \text{(using Eq. 3.3)} \tag{4.8}$$

We insert Eq. 4.6 in the denominator of Eq. 4.7 to compare it to Eq. 4.8.

$$\begin{aligned} \text{FP}_{\mathcal{A}}(\mathcal{J}) &\geq \frac{yT(N_1, M_1, B)}{T(N, M_1, B)} \\ &\geq \frac{yT(N_1, M_1, B)}{c\lceil M_1/B \rceil \lceil W(N)/U(M_1, B, N) \rceil} \\ &\geq \frac{yT(N_1, M_1, B)U(M_1, B, N)}{cm_1 W(N)} \\ &\geq \left\lfloor \frac{qm_1}{dT(N_1, M_1, B)} \right\rfloor \frac{T(N_1, M_1, B)U(M_1, B, N)}{cm_1 W(N)} \\ &\geq \frac{q}{2dc} \frac{U(M_1, B, N)}{W(N)}. \end{aligned} \tag{4.9}$$

Since $q > \max(2dc, 1)$, algorithm $\mathcal{A}$ makes more fractional progress in $\mathcal{J}$ than OPT in $\mathcal{J}$.

Definition 3.3 allows us to sum the maximum progress of OPT in different squares. Since in each square, $\mathcal{A}$ dominates OPT in terms of fractional progress, we get that $\mathcal{A}$ makes more fractional progress than OPT on the entirety of m. ∎

4.2 Inductive Charging and Optimality on All Memory Profiles. We introduce the inductive charging approach, which we use to extend Theorem 4.1 to all memory profiles.

THEOREM 4.2. *Theorem 4.1 is extendable to **all memory profiles** if either of the following is satisfied:*

- ***a.*** *$T(N, M, B)$ satisfies the **regularity condition** [23], i.e., $T(N, M, B) = O(T(N, 2M, B))$.*[8]
- ***b.*** *Progress function $U(M, B, N)$ is polynomial in M.*

Proof. Consider the inner square profile m' of memory profile m. Let $[S_i]$ denote the intervals defined by the boundaries of the squares of m'.

We structure the proof by charging $\text{FP}_{\text{OPT}}(S_{i+1})$ to $\text{FP}_{\mathcal{A}}(S_i)$ inductively. In the base case, we charge $\text{FP}_{\text{OPT}}(S_1 \cup S_2)$ to that of $\text{FP}_{\mathcal{A}}(S_1)$.

Base Case. In Lemma 3.1 we proved that S_2 is at most twice as long as S_1 and that $\forall t \in S_2,\ m(t) \leq 4m'(S_1) = 4\min\{m(t) | t \in S_1\}$. Hence, in the worst case, we can assume that OPT is working with an available memory of $4M$ in $S_1 \cup S_2$ and $\mathcal{A}$ is working with an available memory of M in S_1. Algorithm $\mathcal{A}$ is $(2p+1)$-speed augmented, so it has $(2p+1)M/B = (2p+1)m$ I/Os in S_1. Meanwhile, in $S_1 \cup S_2$ OPT has $\leq 3M/B = 3m$ I/Os. In order to be able to apply the piecewise lower bound easily, we treat OPT generously and allow it $4m$ I/Os instead of $3m$.

[8] Frigo et al. [23] showed that cache-oblivious algorithms that satisfy the regularity condition can be "ported" to systems with LRU page replacement instead of optimal page replacement. The regularity condition has become one of underpinnings of the ideal-cache model.

The inequality of the available memory to $\mathcal{A}$ and OPT makes their respective fractional progress terms incomparable if one wants to repeat the argument of Theorem 4.1. However, when (a) or (b) is true, we can compensate for this difference in memory by giving $\mathcal{A}$ a constant amount of additional speed augmentation.

For case (a), there exists a constant e_1, such that

$$T(N, M, B) = O(T(N, 4M, B)) \leq e_1 T(N, 4M, B).$$

We replace $T(N, M, B)$ with $e_1 T(N, 4M, B)$ in the denominator of the first term in Eq. 4.9. In Eq. 4.9, an additional e_1 appears:

$$\text{FP}_{\mathcal{A}}(\mathcal{J}) \geq \frac{p}{2e_1 dc} \cdot \frac{U(4M_1, B, N)}{W(N)}.$$

If $p > \lceil \max(2e_1 dc, d\alpha\beta, 1) \rceil$, then we get the base case:

$$\text{FP}_{\mathcal{A}}(S_1) > \text{FP}_{\text{OPT}}(S_1 \cup S_2).$$

For case (b), there exists a constant e_2, such that

$$U(4M, B, N) = O(U(M, B, N)) \leq e_2 U(M, B, N).$$

We replace $U(4M, B, N)$ with $e_2 U(M, B, N)$ in the numerator of the last term in Eq. 4.9, and establish the base case similarly to case (a).

Inductive Step. We charge $\text{FP}_{\text{OPT}}(S_{i+1})$ to $\text{FP}_{\mathcal{A}}(S_i)$. The accounting follows that of the base case.

Using Definition 3.3, we sum the fractional progress terms in all intervals and show that $\mathcal{A}$ finishes before OPT. ■

Alternatively, we can compensate for the difference in memory between $\mathcal{A}$ and OPT by simply giving $\mathcal{A}$ more memory, which yields the following theorem.

THEOREM 4.3. *If a cache-oblivious algorithm is optimal on square profiles then, given 4-memory and 4-speed augmentation, it is optimal on all memory profiles.*

4.3 Examples of COR Cache-Oblivious Algorithms. As a corollary of Theorem 4.2, a number of cache-oblivious algorithms are optimally cache-adaptive; see Table 1.

COROLLARY 4.1. *Cache-oblivious naïve matrix multiplication [23], Gaussian elimination paradigm (GEP) [18], Floyd-Warshall APSP [38], and matrix transpose [23] are all optimal cache-adaptive algorithms if* $m = \Omega(\max\{B, \log N\})$ *(see Table 1).*

Proof. We show that all these problems have piece-wise lower bounds. We have already stated the lower bound of naïve matrix multiplication from [26]. The matrix multiplication problem can be stated in the form of Gauss Elimination Paradigm (GEP), so the same piece-wise lower bound works for GEP as well. In [38] it is shown that the Floyd-Warshall algorithm reduces to matrix multiplication with respect to the processor-memory traffic. In other words, their DAGs have similar properties, and the *red-blue pebble game* of [26] can be used to derive a piece-wise lower bound. Finally, the lower bound on matrix transpose comes from the time required to read the input ($W(N) = N, U = 1/B$).

Notice that for matrix multiplication, transpose, and GEP, $M = \Omega(B^2)$. Therefore, the base case n_1 is $\Omega(M)$ and thus $O(\sqrt{n_1} + n_1/B) = O(n_1/B)$. Therefore, all these algorithms are in COR form. Furthermore, all these algorithms are known to have the *regularity condition*. Therefore by Theorem 4.2, we get that all these cache-oblivious algorithms are optimally cache-adaptive. ■

4.4 Lazy Funnel Sort is Optimally Cache-Adaptive. Lazy funnel sort (LFS) [12, 23] is cache oblivious and has a piecewise lower bound, but its running time cannot be written in COR form, so we cannot apply Theorem 4.2. However, by following the strategy of Theorem 4.1, Theorem 4.2 and the analysis of the LFS algorithm in [12] we prove the following theorem in Appendix B.

THEOREM 4.4. *LFS with k-mergers with output buffers of size k^d is optimally cache-adaptive if $d \geq 2$ and $M(t) \geq 2B^{2d/(d-1)}$ for all t.*

5 Page Replacement in the Cache-Adaptive Model

This section gives competitive analyses for page replacement when the size of cache changes over time. Since we measure time in terms of I/Os, two paging algorithms may become out-of-sync when one has a cache miss and the other has a cache hit. Thus, two algorithms may have wildly different amounts of memory available to them when they service the same request. This section shows how to deal with this alignment issue.

5.1 LRU with 4-Memory and 4-Speed Augmentation is Competitive with OPT. We utilize square profiles, Lemma 3.1 and an inductive charging approach similar to Theorem 4.2 to prove competitiveness [43] of LRU with a variable-sized cache. We show that LRU with 4-memory and 4-speed augmentation is competitive with the optimal page replacement algorithm. This shows that, given some resource augmentation, real LRU-based systems can implement the CA model.

NOTATION. *For a sequence of page accesses $\sigma = (\sigma_1, \ldots, \sigma_n)$, memory profile $m(t)$, and page replacement algorithm P, let $C_P(m, \sigma)$ be the number of I/Os required to process σ in m while using page replacement algorithm P. Let $C(m, \sigma)$ denote $C_{\text{OPT}}(m, \sigma)$.*

The following lemma enables us to convert simultane-

Cache-Oblivious Algorithm	Lower Bound	Recurrence Relation
Floyd-Warshall APSP [38], $m = \Omega(B)$	[26]	$Q(N) = \begin{cases} 8Q(N/2) + O(1) & N > \alpha M \\ O(N/B) & N \leq \alpha M \end{cases}$
Naive Matrix Multiplication [23], $m = \Omega(B)$	[26, 28]	$Q(N) = \begin{cases} 8Q(N/4) + O(1) & N > \alpha M \\ O(\sqrt{N} + N/B) & N \leq \alpha M \end{cases}$
Matrix Transposition [40], $m = \Omega(B)$	[23]	$Q(N) = \begin{cases} 2Q(N/2) + O(1) & N > \alpha M \\ O(\sqrt{N} + N/B) & N \leq \alpha M \end{cases}$
Gaussian Elimination Paradigm [18], $m = \Omega(B)$	[18, 26]	$Q(N) = \begin{cases} 8Q(N/4) + O(1) & N > \alpha M \\ O(\sqrt{N} + N/B) & N \leq \alpha M \end{cases}$

Table 1: Cache-oblivious algorithms in constant-overhead recursive (COR) form (Definition 4.1).

ously between LRU and OPT and square and general profiles.

LEMMA 5.1. *Let m be any memory profile. Let m' be the inner square profile of m. Let $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ be any sequence of page accesses. Then $C_{LRU}(m'_{4,4}, \sigma) \leq 4C_{OPT}(m, \sigma)$.*

Proof. Since OPT and LRU can never place more than one page in cache per time step, we may assume without loss of generality that $m(t+1) \leq m(t) + 1$ for all t. Let $0 = t_0 < t_1 < \ldots$ be the inner square boundaries of m.

We show by induction on i that for all σ, if $C_{\text{OPT}}(m, \sigma) \in [t_{i+1}, t_{i+2})$, then $C_{\text{LRU}}(m'_{4,4}, \sigma) \leq 4t_{i+1}$.

Base case $i = 0$**.** We argue that if OPT can process σ in the first two intervals of m, then LRU can process it in the first interval of $m'_{4,4}$. Suppose $C_{\text{OPT}}(m, \sigma) < t_2$. From Lemma 3.1, $t_2 = t_2 - t_0 = (t_2 - t_1) + (t_1 - t_0) \leq 2(t_1 - t_0) + (t_1 - t_0) = 3(t_1 - t_0) = 3t_1$. Since the cache is empty at time $t_0 = 0$, this implies that σ can refer to at most $t_2 \leq 3t_1$ distinct pages. Since, during interval $[0, 4t_1)$, profile $m'_{4,4}$ always has size at least $4t_1$ pages, LRU can load all the pages referenced by σ into memory, and this will require at most $3t_1$ I/Os. Thereafter, no further I/O will be required. Thus $C_{\text{LRU}}(m'_{4,4}, \sigma) < 4t_1$.

Inductive step. The inductive assumption is that for all σ', if $C_{\text{OPT}}(m, \sigma') \in [t_{i+1}, t_{i+2})$, then $C_{\text{LRU}}(m'_{4,4}, \sigma') \leq 4t_{i+1}$. Suppose that $C_{\text{OPT}}(m, \sigma) \in [t_{i+2}, t_{i+3})$. Let σ' be the prefix of σ that OPT services in $[t_0, t_{i+2})$. Consequently, $C_{\text{OPT}}(m, \sigma') < t_{i+2}$. By the inductive hypothesis, $C_{\text{LRU}}(m'_{4,4}, \sigma') \leq 4t_{i+1}$.

Let σ'' be the remainder of σ, i.e. $\sigma = \sigma' || \sigma''$. Observe that, since OPT can process σ'' in $t_{i+3} - t_{i+2} \leq 2(t_{i+2} - t_{i+1})$ time steps and with a cache whose initial size is at most $2(t_{i+2} - t_{i+1})$, σ'' can contain references to at most $4(t_{i+2} - t_{i+1})$ distinct pages.

We now only need to show that LRU can process any suffix of σ'' in interval $[4t_{i+1}, 4t_{i+2})$. The memory available during this interval in $m'_{4,4}$ is $4(t_{i+2} - t_{i+1})$. Thus LRU can load all the distinct pages referenced by σ'' into memory, which will require at most $4(t_{i+2} - t_{i+1})$ I/Os. Thereafter, it can serve all the page requests in σ'' with no further I/O. ∎

THEOREM 5.1. *LRU with 4-memory and 4-speed augmentation always completes sooner than OPT.*

Proof. Take σ, m, and m' as in Lemma 5.1. Since $m'_{4,4}$ always has less memory than $m_{4,4}$, $C_{\text{LRU}}(m_{4,4}, \sigma) \leq C_{\text{LRU}}(m'_{4,4}, \sigma)$. Thus, by Lemma 5.1, $C_{\text{LRU}}(m_{4,4}, \sigma) \leq 4C_{\text{OPT}}(m, \sigma)$. ∎

The following lemma shows that LRU requires speed augmentation to be competitive (proof deferred to the full version). LRU requires memory augmentation [43], so Theorem 5.1 is, up to constants, the best possible.

LEMMA 5.2. *There exists a memory profile $m(t)$ and a page request sequence σ, such that a non-speed augmented LRU performs arbitrarily worse than OPT.*

5.2 Belady's Algorithm is Optimal. Belady's algorithm (LFD) [5], which evicts the page whose next request is farthest in future, is optimal in the DAM model. The following theorem shows that it remains optimal when the size of cache changes over time. The complete proof is postponed to the full version of the paper. We give a summary here.

THEOREM 5.2. *LFD is an optimal page replacement algorithm for a variable-sized cache.*

First, we prove that one can always convert optimal paging policies into ***on-demand*** policies, i.e., the algorithm only loads a page when that page has been requested but is not currently in cache. Then, among all on-demand optimal policies, we consider the policy, OPT, that matches the behavior of LFD for the longest time and then diverges. We modify OPT to have the same cost and match LFD in one more step and get $\mathcal{B}$. However, the modification might make $\mathcal{B}$ *non-on-demand*. We then convert the modified $\mathcal{B}$ back to an on-demand policy, while preserving its agreement with LFD, to arrive at a contradiction.

6 Cache-Adaptive ≠ Cache-Oblivious

This section exhibits a separation between cache-adaptivity and cache-obliviousness by showing that: (1) There exists a cache-adaptive algorithm for permutation, for which no cache-oblivious algorithm can be devised [13], and (2) there exist cache-oblivious algorithms that are not cache-adaptive.

6.1 Cache-Adaptive Permutation.

By interleaving the cache-adaptive sorting algorithm from Subsection 4.4 and the naïve algorithm that puts each element in the correct place individually, we obtain a cache-adaptive permutation algorithm, $\mathcal{IA}$. However, no cache-oblivious algorithm [13] for permutation can be devised.

In Appendix C, we prove the following theorem. The proof is similar to that of Theorem 4.4.

THEOREM 6.1. *Let $d \geq 2$ be the same parameter as in Theorem 4.4. If for all t, $M(t) \geq 2B^{\frac{2d}{d-1}}$, the interleaving algorithm $\mathcal{IA}$, is an optimal cache-adaptive algorithm.*

6.2 Not All Optimal Cache-Oblivious Algorithms Are Optimally Cache-Adaptive.

THEOREM 6.2. *There exists an asymptotically optimal cache-oblivious algorithm that is not optimally cache adaptive, even with any constant-factor resource augmentation.*

Proof. Consider the following problem: given two $\sqrt{N_1} \times \sqrt{N_1}$ matrices and a size-N_2 array of numbers, compute both the matrix product and the sum of all elements in the array.

An algorithm $\mathcal{A}$ that first sums the elements of the array then computes the matrix product using a cache-oblivious matrix multiply is an optimal cache-oblivious algorithm for this problem. Specifically, for fixed memory size $M \geq B^2$, the number of I/Os performed by this algorithm is $\Theta(\textit{scan}(N_2) + \textit{matrix-multiply}(N_1)) = \Theta\left(\frac{N_2}{B} + \frac{N_1^{3/2}}{B\sqrt{M}}\right)$, which is asymptotically optimal.

To see that algorithm $\mathcal{A}$ is not cache adaptive, we consider a specific memory profile and input. For the memory profile, choose any maximum memory size of $m \gg c_1 B$ blocks (i.e., total space at least $M \gg c_1 B^2$), where c_1 is any constant memory augmentation. Use the following memory profile:

$$m(t) = \begin{cases} m & \text{if } t \leq m^{5/2} \\ 2m^{5/2} - t & \text{if } m^{5/2} < t < 2m^{5/2} - B \\ B & \text{if } t \geq 2m^{5/2} - B. \end{cases}$$

Specifically, the memory profile consists of three phases. In the first phase, lasting for $m^{5/2}$ time steps, the memory contains m blocks. In the second phase, consisting of less than $m^{5/2}$ time steps, the memory decreases by one block per time step. In the third phase, which lasts indefinitely, the memory consists of the minimum allowed B blocks.

Choose an input with $N_2/B = 2c_2 m^{5/2}$, where c_2 is the constant of speed augmentation. Choose $\sqrt{N_1}$ such that the cache-oblivious matrix multiply with memory size mB would complete in $m^{5/2}$ I/Os, i.e., with $\sqrt{N_1} = \Theta(m\sqrt{B})$.

Let us now analyze the algorithm $\mathcal{A}$ on this input and profile. The algorithm takes at least $2c_2 m^{5/2}$ I/Os to complete the array scan, ending during third phase even when sped up by a factor of c_2. The matrix multiply must then continue with fixed memory size B, giving an I/O bound of $\Theta(N_1^{3/2}/B^2) = \Theta(m^3/\sqrt{B})$. For $m \gg B$, the total number of I/Os is $\Theta(m^3/\sqrt{B})$.

To see that $\mathcal{A}$ is not optimal, consider an algorithm $\mathcal{A}'$ that first performs the matrix multiply then performs the array sum. Even without memory or speed augmentation, this algorithm completes the matrix multiply in phase 1 by assumption, and the scan completes during third phase, for a total of $\Theta(m^{5/2})$ I/Os. Algorithm $\mathcal{A}$ thus performs $\Theta(\sqrt{m/B})$ times as many I/Os as algorithm $\mathcal{A}'$, which can be increased arbitrarily by increasing $m \gg B$. ∎

7 Related Work

Barve and Vitter [3, 4] generalize the DAM model to allow for fluctuations in memory size. They give optimal sorting, matrix multiplication, LU decomposition, FFT, and permutation algorithms.

There also exist empirical studies of adaptivity. Zhang and Larson [48] and Pang, Carey, and Livny [36] both give memory-adaptive sorting algorithms. Other papers discussing aspects of adaptivity include [3, 4, 15, 24, 34, 35, 37, 47, 49].

The notion of a cache-oblivious algorithm was proposed by Frigo et al. [23, 40]. Because cache-oblivious algorithms can be optimal without resorting to memory parameters, they can be uniformly optimal on unknown, multilevel memory hierarchies. See [21, 30] for surveys of cache-oblivous algorithms. See [8, 9, 14, 16, 17, 19, 22, 23, 31, 44, 45] for discussions of implementations and performance analysis of cache-oblivious algorithms. See [6, 7, 13] for a discussion of the limits of cache-obliviousness.

Cache-oblivious programming helps multicore programming. Blelloch et al. [10] define a class of recursive algorithms HR that work well on multicores [10]; this class is more general than the one defined in Section 4. Cole and Ramachandran [20] define BP and HBP, two classes of recursive cache-oblivious algorithms that also behave well on multicores. Blelloch, Gibbons, and Simhadri [11] prove results for algorithms with low recursive depth.

Peserico [39] considers an alternative model for page replacement when the cache size fluctuates. Peserico's page-replacement model does not apply to the CA model, because the increases and decreases in the cache size appear at specific locations in the page-request sequence, rather than at specific points in times.

Katti and Ramachandran consider page replacement polices for multicore shared-cache environments [29]. Several authors have considered other aspects of paging where the size of internal memory or the pages themselves vary [2,25,27,32,33,46] e.g., because several processes share the same cache. For example, [32] considers a model where the application itself adjusts the cache-size, and [27,46] consider a model where the page sizes vary.

8 Conclusion

We show how to design cache-adaptive algorithms by starting with cache-oblivious algorithms. Many optimal cache-oblivious algorithms also have good experimental performance [14, 23]. Hence, our results may open up new ways to design cache-adaptive algorithms that are provably good, empirically good, and practical to implement.

Many open problems remain, suggesting an area ripe for further investigation. Can our general theorem be extended to algorithms with other recursive structures? Could we prove the optimality of cache-oblivious FFT [23], LCS, and edit distance [18] in the CA model? There has also been work in using subclasses of cache-oblivious algorithms to achieve universal performance bounds for shared-memory multiprocessors [10, 11, 20]. It is worth exploring additional links between cache-obliviousness, shared-memory parallel computing, and cache-adaptivity.

Acknowledgments

We gratefully acknowledge Goetz Graefe, Harumi Kuno, Bradley Kuszmaul, and Sivaramakrishnan Narayanan for discussions on memory adaptivity in databases.

References

[1] A. AGGARWAL AND S. VITTER, JEFFREY, *The input/output complexity of sorting and related problems*, Communications of the ACM, 31 (1988), pp. 1116–1127.

[2] R. D. BARVE, E. F. GROVE, AND J. S. VITTER, *Application-controlled paging for a shared cache*, SIAM J. Comput., 29 (2000), pp. 1290–1303.

[3] R. D. BARVE AND J. S. VITTER, *External memory algorithms with dynamically changing memory allocations*, tech. rep., Duke University, 1998.

[4] R. D. BARVE AND J. S. VITTER, *A theoretical framework for memory-adaptive algorithms*, in Proc. 40th Annual Symposium on the Foundations of Computer Science (FOCS), 1999, pp. 273–284.

[5] L. A. BELADY, *A study of replacement algorithms for a virtual-storage computer*, IBM Journal of Research and Development, 5 (1966), pp. 78–101.

[6] M. A. BENDER, G. S. BRODAL, R. FAGERBERG, D. GE, S. HE, H. HU, J. IACONO, AND A. LOPEZ-ORTIZ, *The cost of cache-oblivious searching*, in Proc. 44th Annual Symposium on Foundations of Computer Science (FOCS), 2003, pp. 271–280.

[7] M. A. BENDER, G. S. BRODAL, R. FAGERBERG, D. GE, S. HE, H. HU, J. IACONO, AND A. LÓPEZ-ORTIZ, *The cost of cache-oblivious searching*, Algorithmica, 61 (2011), pp. 463–505.

[8] M. A. BENDER, M. FARACH-COLTON, J. T. FINEMAN, Y. R. FOGEL, B. C. KUSZMAUL, AND J. NELSON, *Cache-oblivious streaming B-trees*, in Proc. 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), 2007, pp. 81–92.

[9] M. A. BENDER, M. FARACH-COLTON, AND B. C. KUSZMAUL, *Cache-oblivious string B-trees*, in Proc. 25th Annual ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS), 2006, pp. 233–242.

[10] G. E. BLELLOCH, R. A. CHOWDHURY, P. B. GIBBONS, V. RAMACHANDRAN, S. CHEN, AND M. KOZUCH, *Provably good multicore cache performance for divide-and-conquer algorithms*, in Proc. 19th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Society for Industrial and Applied Mathematics, 2008, pp. 501–510.

[11] G. E. BLELLOCH, P. B. GIBBONS, AND H. V. SIMHADRI, *Low depth cache-oblivious algorithms*, in Proc. 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), 2010, pp. 189–199.

[12] G. S. BRODAL AND R. FAGERBERG, *Cache oblivious distribution sweeping*, in Proc. of the 29th International Colloquium on Automata, Languages and Programming (ICALP), Springer-Verlag, 2002, pp. 426–438.

[13] G. S. BRODAL AND R. FAGERBERG, *On the limits of cache-obliviousness*, in Proc. 35th Annual ACM Symposium on Theory of Computing (STOC), 2003, pp. 307–315.

[14] G. S. BRODAL, R. FAGERBERG, AND K. VINTHER, *Engineering a cache-oblivious sorting algorithm*, ACM Journal of Experimental Algorithmics, 12 (2007).

[15] K. P. BROWN, M. J. CAREY, AND M. LIVNY, *Managing memory to meet multiclass workload response time goals*, in Proc. 19th International Conference on Very Large Data Bases (VLDB), Institute of Electrical & Electronics Engineers (IEEE), 1993, pp. 328–328.

[16] R. CHOWDHURY, M. RASHEED, D. KEIDEL, M. MOUSSALEM, A. OLSON, M. SANNER, AND C. BAJAJ, *Protein-protein docking with f2dock 2.0 and gb-rerank*, PLoS ONE, 8 (2013).

[17] R. A. CHOWDHURY, H.-S. LE, AND V. RAMACHANDRAN, *Cache-oblivious dynamic programming for bioinformatics*, IEEE/ACM Trans. Comput. Biology Bioinform., 7 (2010), pp. 495–510.

[18] R. A. CHOWDHURY AND V. RAMACHANDRAN, *Cache-oblivious dynamic programming*, in Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), ACM, 2006, pp. 591–600.

[19] ———, *The cache-oblivious gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation*, Theory of Computing Systems, 47 (2010), pp. 878–919.

[20] R. COLE AND V. RAMACHANDRAN, *Resource oblivious sorting on multicores*, Automata, Languages and Programming, (2010), pp. 226–237.

[21] E. D. DEMAINE, *Cache-oblivious algorithms and data struc-*

tures. Lecture Notes from the EEF Summer School on Massive Data Sets, 2002.

[22] M. FRIGO AND S. G. JOHNSON, *The design and implementation of FFTW3*, Proceedings of the IEEE, 93 (2005), pp. 216–231.

[23] M. FRIGO, C. E. LEISERSON, H. PROKOP, AND S. RAMACHANDRAN, *Cache-oblivious algorithms*, in Proc. 40th Annual Symposium on the Foundations of Computer Science (FOCS), 1999, pp. 285–298.

[24] G. GRAEFE, *A new memory-adaptive external merge sort*. Private communication, July 2013.

[25] A. HASSIDIM, *Cache replacement policies for multicore processors*, in Proc. 1st Annual Symposium on Innovations in Computer Science (ICS), 2010, pp. 501–509.

[26] J.-W. HONG AND H. T. KUNG, *I/O complexity: The red-blue pebble game*, in Proc. 13th Annual ACM Symposium on the Theory of Computation (STOC), 1981, pp. 326–333.

[27] S. IRANI, *Page replacement with multi-size pages and applications to web caching*, in Proc. 29th Annual ACM Symposium on the Theory of Computing (STOC), 1997, pp. 701–710.

[28] D. IRONY, S. TOLEDO, AND A. TISKIN, *Communication lower bounds for distributed-memory matrix multiplication*, Journal of Parallel and Distributed Computing, 64 (2004), pp. 1017–1026.

[29] A. K. KATTI AND V. RAMACHANDRAN, *Competitive cache replacement strategies for shared cache environments*, in Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IPDPS '12, IEEE Computer Society, 2012, pp. 215–226.

[30] P. KUMAR, *Cache oblivious algorithms*, in Algorithms for Memory Hierarchies, LNCS 2625, U. Meyer, P. Sanders, and J. Sibeyn, eds., Springer Verlag, 2003, pp. 193–212.

[31] R. E. LADNER, R. FORTNA, AND B.-H. NGUYEN, *A comparison of cache aware and cache oblivious static search trees using program instrumentation*, Experimental Algorithmics, (2002), pp. 78–92.

[32] A. LÓPEZ-ORTIZ AND A. SALINGER, *Minimizing cache usage in paging*, in Proc. 10th Workshop on Approximation and Online Algorithms (WAOA), 2012.

[33] A. LÓPEZ-ORTIZ AND A. SALINGER, *Paging for multi-core shared caches*, in Proc. Innovations in Theoretical Computer Science (ITCS), 2012, pp. 113–127.

[34] R. T. MILLS, *Dynamic adaptation to CPU and memory load in scientific applications*, PhD thesis, The College of William and Mary, 2004.

[35] R. T. MILLS, A. STATHOPOULOS, AND D. S. NIKOLOPOULOS, *Adapting to memory pressure from within scientific applications on multiprogrammed cows*, in Proc. 8th International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2004, p. 71.

[36] H. PANG, M. J. CAREY, AND M. LIVNY, *Memory-adaptive external sorting*, in Proc. 19th International Conference on Very Large Data Bases (VLDB), Morgan Kaufmann, 1993, pp. 618–629.

[37] H. PANG, M. J. CAREY, AND M. LIVNY, *Partially preemptible hash joins*, in Proc. 5th ACM SIGMOD International Conference on Management of Data (COMAD), 1993, p. 59.

[38] J.-S. PARK, M. PENNER, AND V. K. PRASANNA, *Optimizing graph algorithms for improved cache performance*, Parallel and Distributed Systems, IEEE Transactions on, 15 (2004), pp. 769–782.

[39] E. PESERICO, *Paging with dynamic memory capacity*, CoRR, abs/1304.6007 (2013).

[40] H. PROKOP, *Cache oblivious algorithms*, Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

[41] J. E. SAVAGE, *Extending the Hong-Kung model to memory hierachies*, in Proc. 1st Annual International Conference on Computing and Combinatorics (COCOON), vol. 959 of Lecture Notes in Computer Science, 1995, pp. 270–281.

[42] J. E. SAVAGE AND J. S. VITTER, *Parallelism in space-time tradeoffs*, Advances in Computing Research, 4 (1987), pp. 117–146.

[43] D. D. SLEATOR AND R. E. TARJAN, *Amortized efficiency of list update and paging rules*, Communications of the ACM, 28 (1985), pp. 202–208.

[44] S.-E. YOON, P. LINDSTROM, V. PASCUCCI, AND D. MANOCHA, *Cache-oblivious mesh layouts*, 24 (2005), pp. 886–893.

[45] K. YOTOV, T. ROEDER, K. PINGALI, J. GUNNELS, AND F. GUSTAVSON, *An experimental comparison of cache-oblivious and cache-conscious programs*, in Proc. 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), 2007, pp. 93–104.

[46] N. E. YOUNG, *On-line file caching*, Algorithmica, 33 (2002), pp. 371–383.

[47] H. ZELLER AND J. GRAY, *An adaptive hash join algorithm for multiuser environments*, in Proc. 16th International Conference on Very Large Data Bases (VLDB), 1990, pp. 186–197.

[48] W. ZHANG AND P. LARSON, *A memory-adaptive sort (masort) for database systems*, in Proc. of the 6th International Conference of the Centre for Advanced Studies on Collaborative research (CASCON), IBM Press, 1996, pp. 41–.

[49] ———, *Dynamic memory adjustment for external mergesort*, in Proc. of the 23rd International Conference on Very Large Data Bases (VLDB), Morgan Kaufmann Publishers Inc., 1997, pp. 376–385.

A Proof of Lemma 4.1

Statement of Lemma 4.1. *Assume a tree T of height H and branching factor greater than or equal to 2. Given any node x, consider the set of next z leaves, S_z, that appear after x in the DFS order of the tree. Let I_z be the set of internal nodes in the DFS order between x and the last node of S_z. Then $|I_z| \leq 3H + z$.*

Proof. Consider the set of maximal complete subtrees, $\mathcal{T} = \{T_1, \ldots, T_k\}$, ordered from left to right, induced by the leaves in S_z. Let u be the lowest common ancestor of the nodes in S_z. Let r_i be the root of T_i.

As illustrated in Figure 4, the DFS order of the tree will traverse

- the path P_x from x to r_1,
- the path P_1 between r_1 and u,
- the subtrees in $\mathcal{T}$,
- the path P_k from u to r_k.

The paths P_1 and P_x will have nodes in common, so we analyze them together. If x is not a descendent of u, then P_x contains P_1 and has length at most $2H$. If x is a descendent of u, then P_x intersects P_1 after at most H steps, and P_1 contains at most H additional nodes. In either case, the total number of distinct nodes in $P_1 \cup P_x$ is at most $2H$.

The path P_k contains at most H nodes.

Because the branching factor is at least 2, the subtrees in $\mathcal{T}$ contain a total of at most z internal nodes.

Thus the total number of nodes visited by the DFS is at most $3H + z$. ∎

B Lazy Funnel Sort is Optimally Cache-Adaptive

We start with a short exposition of the lazy funnel sort (LFS) algorithm [12]. Then, we prove that LFS is an optimal cache-adaptive algorithm.

At the core of the LFS algorithm lies the concept of a ***k-merger***, a perfectly balanced binary tree with k leaves and a binary merger at each internal node. Each leaf has a sorted input stream, and the root has an output stream with capacity k^d, where $d \geq 2$ is a tuning parameter. The size of buffers between internal nodes are defined recursively: Consider a horizontal cut in the tree at half its full height: $D_0 = \lceil \log(k)/2 \rceil$. The buffers between nodes of depth D_0 and $D_0 + 1$ have size $\lceil k^{d/2} \rceil$. The subtree above depth D_0 is the ***top tree*** and all the subtrees rooted below are ***bottom trees***. The sizes of the buffers in the top and bottom trees are defined recursively.

Upon each invocation, a k-merger merges k^d elements into its output buffer. We call a complete invocation of a k-merger, producing k^d elements in the output buffer, a round of execution of that k-merger. A k-merger together with its internal buffer is linearized in a recursive Van Emde Boas layout. First, the top subtree is laid out in a contiguous array and then all the bottom subtrees are laid out in contiguous arrays. The work flow of a k-merger is based on recursive calls to the underlying binary merger tree. A call is made to the root of a k-merger to fill its output stream by merging its two input buffers. When one of the buffers runs out of elements, a recursive call is made to the child node to fill it. For a complete description and analysis we refer the interested reader to [12].

LFS cannot use a single N-merger to sort the input array, since an N-merger would have superlinear size. Instead, LFS calls itself recursively to produce $N^{1/d}$ sorted streams of size $N^{1-1/d}$ and then merges these using an $N^{1/d}$-merger.

To analyze the algorithm we need the following lemma from [12].

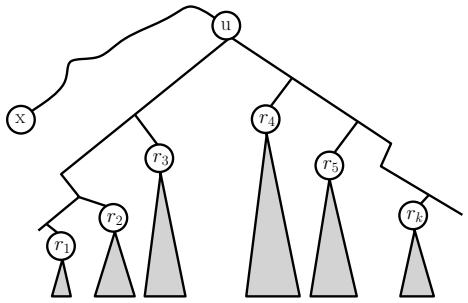

Figure 4: Pattern of nodes of S_z in the DFS order of T.

LEMMA B.1. (FROM [12]) *Let $d \geq 2$. The size of a k-merger (excluding its output buffer) is bounded by $pk^{(d+1)/2}$ for a constant $p \geq 1$. Assuming that $M(t)/2p \geq B^{(d+1)/(d-1)}$ for all t, if a k-merger together with one block from each of its input buffers fit in memory, it performs $O(k^d/B + k)$ I/Os to output k^d elements to its output buffer.*

First, we prove optimality of LFS on square profiles. The inductive charging scheme of Theorem 4.2 can be performed to extend the optimality to all profiles. The later part is omitted due to its simplicity and similarity to Theorem 4.2.

Statement of Theorem 4.4. *LFS with k-mergers with output buffers of size k^d is optimally cache-adaptive if $d \geq 2$ and $M(t) \geq 2B^{2d/(d-1)}$ for all t.*

Proof. First, we assume that $m(t)$ is a square profile. We will show there exists a constant q such that LFS with $4q$-speed augmentation is optimal. We prove the optimality of LFS by comparing the fractional progress of LFS (upper bound) with the fractional progress specified by the lower bound in any given square of the profile $m(t)$, which for simplicity we refer to as actions of an algorithm called OPT.

Let $\mathcal{J}$ be any square of the profile $m(t)$ and let $m_1 = M_1/B$ be the length of $\mathcal{J}$. By definition, the available memory in $\mathcal{J}$ is M_1 and LFS with $4q$ speed augmentation has $4qm_1$ I/Os during $\mathcal{J}$.

Lazy Funnel Sort's Analysis. In the LFS algorithm, each input element must pass through $\lceil \lg N \rceil$ binary mergers to reach to the buffer at the root of the tree. As such, the total task of LFS is for all N elements to climb $\lceil \lg N \rceil$ binary mergers. Let $\phi(\mathcal{J})$ denote the total number of individual ***climbs*** that the LFS algorithm achieves during an interval $\mathcal{J}$. Thus $\mathrm{FP}_{\mathrm{LFS}}(\mathcal{J}) = \phi(\mathcal{J})/N\lceil \lg N \rceil$.

We are going to argue that certain x-mergers, which we refer to as ***active trees*** , are operational in $\mathcal{J}$. The recursive definition of buffer sizes also defines recursive subtrees. We roll out the recursive definition of these subtrees until we reach an x-merger such that x is the biggest value for which

$$x^d \leq M_1/2. \tag{B.1}$$

We refer to such a subtree as an ***active tree***. Note that since x is the biggest such value, we have that

$$x^{2d} > M_1/2. \tag{B.2}$$

An active tree is a little different than the notion of ***base tree*** in [12], because it is not the biggest x-merger that fits in memory during $\mathcal{J}$, but rather the biggest x-merger that can perform at least one round during $\mathcal{J}$ while fitting in memory. This difference is essential in our analysis, since we measure finished rounds of x-mergers in $\mathcal{J}$.

Lemma B.2 follows the strategy of Brodal, et al. [12], to prove the following properties of active trees.

LEMMA B.2. *An **active tree** with one block from each of its input buffers fits in a memory of size M_1. It takes an active tree $O(x^d/B)$ I/Os to perform one round.*

Proof. We show that the x-merger itself and the set of input buffer blocks both take at most $M_1/2$ memory.

The size of an active tree (an x-merger) is bounded by $px^{(d+1)/2}$ by Lemma B.1. We have:

$$\begin{aligned} px^{\frac{d+1}{2}} &\le p(M_1/2)^{\frac{d+1}{2d}} && \triangleright \text{ since } x \le (M_1/2)^{\frac{1}{d}} \text{ by Eq. B.1} \\ &\ll M_1/2 && \triangleright \text{ since } d \ge 2. \end{aligned}$$

Since an x-merger has x leaves, one block from each input buffer will occupy a total of x blocks of memory. Therefore

$$\begin{aligned} Bx &\le B(M_1/2)^{\frac{1}{d}} && \triangleright \text{ since } x \le (M_1/2)^{\frac{1}{d}} \\ &\le (M_1/2)^{\frac{d-1}{2d}}(M_1/2)^{\frac{1}{d}} && \triangleright \text{ since } M_1 \ge 2B^{\frac{2d}{d-1}} \\ &\le (M_1/2)^{\frac{d+1}{2d}} \\ &\ll M_1/2 && \triangleright \text{ since } d \ge 2. \end{aligned}$$

which gives us the desired space bound.

To finish one round, an active tree needs to load the x-merger structure together with one block from each of its input buffers into memory which takes $O(px^{(d+1)/2}/B + x)$ I/Os. By definition, the x-merger outputs x^d elements to its output buffer and reading these elements takes $O(x^d/B)$ I/Os.

Hence, an active tree takes $O(x^d/B + px^{(d+1)/2}/B + x)$ I/Os to finish one round. We argue that

$$O(x^d/B + px^{(d+1)/2}/B + x) = O(x^d/B).$$

Since $d \ge 2$, we have $px^{(d+1)/2}/B = O(x^d/B)$. To get that $x = O(x^d/B)$ too, we prove that $x^d \ge Bx$.

$$\begin{aligned} x^d &\ge x\left((M_1/2)^{\frac{1}{2d}}\right)^{d-1} && \triangleright \text{ since } x \ge (M_1/2)^{\frac{1}{2d}} \\ &\ge x\,(M_1/2)^{\frac{d-1}{2d}} \\ &\ge xB^{\frac{2d}{d-1}\frac{d-1}{2d}} && \triangleright \text{ since } M_1 \ge 2B^{\frac{2d}{d-1}} \\ &\ge xB. \end{aligned}$$

■

Now we measure how many rounds active trees perform during $\mathcal{J}$.

LEMMA B.3. *The number of **active trees** that start and finish during $\mathcal{J}$ is at least $\frac{2Bqm_1}{c_1x^d}$, for some constant c_1.*

Proof. By Lemma B.2, we have that one round of an active tree takes $O(x^d/B)$ I/Os. We also have that:

$$\begin{aligned} O(x^d/B) &\le c_1x^d/B && \triangleright \text{ For some constant } c_1 \\ &\le c_1m_1/2 && \triangleright \text{ since } x^d \le M_1/2 \text{ by Eq. B.1.} \end{aligned}$$

Since $\mathcal{J}$ contains $4qm_1$ I/Os, LFS must complete at least $\frac{4Bqm_1}{c_1x^d} - 2$ rounds during $\mathcal{J}$ (the first and last rounds may not be contained entirely within $\mathcal{J}$ due to alignment). If we choose $q > c_1/2$, then, since each round takes less than $c_1m_1/2$ I/Os, the number of rounds completed during $\mathcal{J}$ will be at least 2. In this case, $\frac{4Bqm_1}{c_1x^d} - 2 \ge \frac{2Bqm_1}{c_1x^d}$. ■

LEMMA B.4. *The fractional progress of LFS during $\mathcal{J}$ has the following lower bound:*

$$\mathrm{FP}_{\mathrm{LFS}}(\mathcal{J}) \ge \frac{q}{2c_1d}\frac{M_1\left(\lg m_1 + \lg B - 1\right)}{N \lg N}. \tag{B.3}$$

Proof. During each round, x^d elements climb up $\lg x$ binary mergers. Since $x > (M_1/2)^{1/2d}$, we have that $\lg x \ge \frac{1}{2d}\lg\frac{M_1}{2}$. Therefore, the total number of climbs in $\mathcal{J}$ by LFS is at least:

$$\begin{aligned} \phi(\mathcal{J}) &\ge \frac{2Bqm_1}{c_1x^d}x^d\left(\frac{1}{2d}\lg\frac{M_1}{2}\right) \\ &\ge \frac{q}{c_1d}M_1\left(\lg m_1 + \lg B - 1\right). \end{aligned}$$

This means that

$$\begin{aligned} \mathrm{FP}_{\mathrm{LFS}}(\mathcal{J}) &\ge \frac{q}{c_1d}\frac{M_1\left(\lg m_1 + \lg B - 1\right)}{N\lceil \lg N\rceil} \\ &\ge \frac{q}{2c_1d}\frac{M_1\left(\lg m_1 + \lg B - 1\right)}{N \lg N}. \end{aligned}$$

■

Lower Bound Analysis. We upper bound the maximum possible fractional progress of OPT in $\mathcal{J}$.

LEMMA B.5.

$$\mathrm{FP}_{\mathrm{OPT}}(\mathcal{J}) \le \frac{2M_1\left(\lg m_1 + \lg B + \lg 4e\right)}{N \lg N}. \tag{B.4}$$

Proof. We review the piece-wise lower bound of Aggarwal and Vitter [1]. Each time an algorithm reads in a block, it can compare every element in that block with every other element in memory. These comparisons are not all independent though, so the total number of distinct outcomes

is bounded by $\binom{M}{B}B!$. Any sorting algorithm must perform enough page loads to be able to generate all $N!$ outcomes of the sort. By taking logs, we obtain a per-I/O progress bound

$$\lg\left(\binom{M}{B}(B!)\right) \leq B\left(\lg m + \lg B + \lg e\right)$$

and a work lower bound of $W(N) = \lg N! \geq \frac{N \lg N}{2}$. Therefore, in a time interval $\mathcal{J}$ with M_1 memory and m_1 I/Os,

$$\text{FP}_{\text{OPT}}(\mathcal{J}) \leq \frac{2M_1\left(\lg m_1 + \lg B + \lg 4e\right)}{N \lg N}.$$

■

By comparing Eq. B.3 and Eq. B.4, we get that if $q \geq \max(4c_1 d, c_1/2)$, then $\text{FP}_{\text{LFS}}(\mathcal{J}) \geq \text{FP}_{\text{OPT}}(\mathcal{J})$.

The piecewise lower bound allows us to sum the fractional progress terms over different squares of m. Hence, we proved that LFS is an optimal cache-adaptive algorithm over all square profiles.

Applying the inductive charging argument of Theorem 4.2, we can extend the optimality to all profiles. ■

C An Optimal Cache-Adaptive Permutation Algorithm

Permutation is very similar to sorting—in fact, permutation can be performed optimally in the DAM model by sorting or by moving each element to its final location explicitly, whichever is faster [1]. However, there can be no optimal cache-oblivious permutation algorithm [13].

We show that by interleaving the cache-adaptive sorting algorithm from Subsection 4.4 and the naïve algorithm that puts each element in the correct place individually, we obtain a cache-adaptive permutation algorithm. Let $\mathcal{IA}$ be the algorithm that runs cache-adaptive sorting and the naïve placement algorithm simultaneously, using half of the cache for each, and doing one I/O for sorting, then one I/O for naïve placement, alternating until one of the methods is successful.

Statement of Theorem 6.1. *Let $d \geq 2$ be the same parameter as in Theorem 4.4. If for all t, $M(t) \geq 2B^{\frac{2d}{d-1}}$, then the interleaving algorithm $\mathcal{IA}$, is an optimal cache-adaptive algorithm.*

Proof. We apply the same technique as in the proof of Theorem 4.4. Therefore, it suffices to show optimality of the interleaving algorithm $\mathcal{IA}$ on *square profiles*.

Let $\mathcal{J}$ be any square of the profile $m(t)$ and let $m_1 = M_1/B$ be the length of $\mathcal{J}$. By definition, the available memory in $\mathcal{J}$ is M_1. We show that there exists a constant q such that $\mathcal{IA}$ with $4q$-speed augmentation is optimal. By definition, there are $4qm_1$ I/Os available to $\mathcal{IA}$ in $\mathcal{J}$.

Aggarwal and Vitter [1] showed that, with a cache of size M, a single I/O could increase the number of permutations an algorithm could generate by a factor of at most

$$N(1 + \lg N)B!\binom{M}{B}$$

This is a multiplicative piecewise progress bound. The total number of permutions is $N!$, so taking logs and applying Stirling's approximation yields a bound on fractional progress:

$$\text{FP}_{\text{OPT}}(\mathcal{J}) \leq \frac{2M_1(\lg m_1 + \lg B + \lg e)}{N \lg N} + \frac{2m_1(\lg N + \lg\lg N)}{N \lg N}.$$

Using the analysis of LFS from Appendix B, the time spent doing Lazy Funnel Sort on $2qm_1$ I/Os (half the I/Os in $\mathcal{J}$) on $M_1/2$ memory gives fractional progress of at least

$$\frac{q}{8c_1 d}\frac{M_1\left(\lg m_1 + \lg B - 3\right)}{N \lg N}.$$

During the I/Os spent on the naïve algorithm, each I/O writes exactly one element to disk; N items must be written in total. The fractional progress of one block transfer is $1/N$, so the total fractional progress in $\mathcal{J}$ is $2qm_1/N$. Then using half the I/Os in the interval for sorting and half for naïve placement, we have:

$$\begin{aligned}\text{FP}_{\mathcal{IA}}(\mathcal{J}) &\geq \frac{q}{8c_1 d}\frac{M_1\left(\lg m_1 + \lg B - 3\right)}{N \lg N} + \frac{2qm_1}{N} \\ &\geq \frac{q}{8c_1 d}\frac{M_1\left(\lg m_1 + \lg B - 3\right)}{N \lg N} + \frac{2qm_1 \lg N}{N \lg N}.\end{aligned}$$

If $q \geq \max\{16c_1 d, 2\}$, then $2q \lg N \geq 2 \lg N + \lg\lg N$ for $N \geq 2$, and hence $\text{FP}_{\mathcal{IA}}(\mathcal{J}) \geq \text{FP}_{\text{OPT}}(\mathcal{J})$.

Repeating the strategy from Theorem 4.4 can extend this result to show that $\mathcal{IA}$ is optimally cache-adaptive on all memory profiles. ■