Towards a Theory of Cache-Efficient Algorithms

SANDEEP SEN

Indian Institute of Technology Delhi, New Delhi, India

SIDDHARTHA CHATTERJEE

IBM Research, Yorktown Heights, New York

AND

NEERAJ DUMIR

Indian Institute of Technology Delhi, New Delhi, India

Abstract. We present a model that enables us to analyze the running time of an algorithm on a computer with a memory hierarchy with limited associativity, in terms of various cache parameters. Our cache model, an extension of Aggarwal and Vitter's I/O model, enables us to establish useful relationships between the cache complexity and the I/O complexity of computations. As a corollary, we obtain cache-efficient algorithms in the single-level cache model for fundamental problems like sorting, FFT, and an important subclass of permutations. We also analyze the average-case cache behavior of mergesort, show that ignoring associativity concerns could lead to inferior performance, and present supporting experimental evidence.

We further extend our model to multiple levels of cache with limited associativity and present optimal algorithms for matrix transpose and sorting. Our techniques may be used for systematic

---

Some of the results in this article appeared in preliminary form in SEN, S., AND CHATTERJEE, S. 2000. Towards a theory of cache-efficient algorithms. In *Proceedings of the Symposium on Discrete Algorithms*. ACM, New York.

This work is supported in part by DARPA Grant DABT63-98-1-0001, NSF Grants CDA-97-2637 and CDA-95-12356, The University of North Carolina at Chapel Hill, Duke University, and an equipment donation through Intel Corporation's Technology for Education 2000 Program.

The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

Part of this work was done when S. Sen was a visiting faculty member in the Department of Computer Science, The University of North Carolina, Chapel Hill, NC 27599-3175.

This work was done when S. Chatterjee was a faculty member in the Department of Computer Science, The University of North Carolina, Chapel Hill, NC 27599-3175.

Authors' addresses: S. Sen and N. Dumir, Department of Computer Science and Engineering, IIT Delhi, New Delhi 110016, India, e-mail: ssen@cse.iitd.ernet.in; S. Chatterjee, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, e-mail: sc@us.ibm.com.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2002 ACM 0004-5411/02/1100-0828 $5.00

exploitation of the memory hierarchy starting from the algorithm design stage, and for dealing with the hitherto unresolved problem of limited associativity.

Categories and Subject Descriptors: F1.1 [**Computation by Abstract Devices**]: Models of Computation; F.2 [**Analysis of Algorithms and Problem Complexity**]

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Hierarchical memory, I/O complexity, lower bound

1. *Introduction*

Models of computation are essential for abstracting the complexity of real machines and enabling the design and analysis of algorithms. The widely used RAM model owes its longevity and usefulness to its simplicity and robustness. Although it is far removed from the complexities of any physical computing device, it successfully predicts the relative performance of algorithms based on an abstract notion of operation counts.

The RAM model assumes a flat memory address space with unit-cost access to any memory location. With the increasing use of caches in modern machines, this assumption grows less justifiable. On modern computers, the running time of a program is often as much a function of operation count as of its cache reference pattern. A result of this growing divergence between model and reality is that operation count alone is not always a true predictor of the running time of a program, and manifests itself in anomalies such as a matrix multiplication algorithm demonstrating $O(n^5)$ running time instead of the expected $O(n^3)$ behavior [Alpern et al. 1994]. Such shortcomings of the RAM model motivate us to seek an alternative model that more realistically models the presence of a memory hierarchy. In this article, we address the issue of better and systematic utilization of caches starting from the algorithm design stage.

A challenge in coming up with a good model is achieving a balance between abstraction and fidelity, so as not to make the model unwieldy for theoretical analysis or simplistic to the point of lack of predictiveness. The memory hierarchy models used by computer architects to design caches have numerous parameters and suffer from the first shortcoming [Agarwal et al. 1989; Przybylski 1990]. The early theoretical work in this area focused on a two-level memory model [Hong and Kung 1981]—a very large capacity memory with slow access time (external memory) and a limited size faster memory (internal memory)—in which all computation is performed on elements in the internal memory and where there is no restriction on placement of elements in the internal memory (a fully associative mapping and a user-defined replacement policy).

The focus of this article is on the interaction between main memory and *cache*, which is the first level of memory hierarchy that is searched for data once the address is provided by the CPU. A single level of cache memory is characterized by three structural parameters—*A*ssociativity, *B*lock size, and *C*apacity[1]—and one functional parameter: the cache *replacement policy*. Capacity and block size are in units of the minimum memory access size (usually one byte). A cache can hold a maximum of C bytes. However, due to physical constraints, the cache is divided

[1] This characterization is referred to as the ABC model of caches in the computer architecture community.

into *cache frames* of size B that contain B contiguous bytes of memory—called a *memory block*. The associativity A specifies the number of different frames in which a memory block can reside. If a block can reside in any frame (i.e., $A = C/B$), the cache is said to be *fully associative*; if $A = 1$, the cache is said to be *direct-mapped*; otherwise, the cache is A*-way set associative*.

For an access to a given memory address m, the hardware inspects the cache to determine if the data at memory address m is resident in the cache. This is accomplished by using an indexing function to locate the appropriate set of cache frames that may contain the memory block enclosing address m. If the memory block is not resident, a *cache miss* is said to occur. From an architectural standpoint, misses in a target cache can be partitioned into one of three classes [Hill and Smith 1989].

—A *compulsory miss* (also called a *cold miss*) is one that is caused by referencing a previously unreferenced memory block.

—A reference that is not a compulsory miss but misses both in the target cache and in a fully associative cache of equal capacity and with LRU replacement is classified as a *capacity miss*. Capacity misses are caused by referencing more memory blocks than can fit in the cache.

—A reference that is not a compulsory miss and hits in a fully associative cache of equal capacity and with LRU replacement but misses in the target cache is classified as a *conflict miss*. Such a miss occurs because of the restriction in the address mapping and not because of lack of space in the cache.

Conflict misses pose an additional challenge in designing efficient algorithms for cache. This class of misses is not present in the I/O model developed for the memory-disk interface [Aggarwal and Vitter 1988], where the mapping between internal and external memory is fully associative and the replacement policy is not fixed and predetermined.

Existing memory hierarchy models [Aggarwal and Vitter 1988; Aggarwal et al. 1987a, 1987b; Alpern et al. 1994] do not model certain salient features of caches, notably the lack of full associativity in address mapping and the lack of explicit control over data movement and replacement. Unfortunately, these small differences are malign in the effect.[2] They introduce *conflict misses* that make analysis of algorithms much more difficult [Fricker et al. 1995]. Carter and Gatlin [1998] conclude a recent paper saying

> What is needed next is a study of "messy details" not modeled by UMH (particularly cache associativity) that are important to the performance of the remaining steps of the FFT algorithm.

In this article, we develop a two-level memory hierarchy model to capture the interaction between cache and main memory. Our model is a simple extension of the two-level I/O model that Aggarwal and Vitter [1988] proposed for analyzing external memory algorithms. However, it captures several additional constraints of caches, namely, lower miss penalties, lack of full associativity, and lack of explicit program control over data movement and cache replacement. The work in this

[2] See the discussion in Carter and Gatlin [1998] on a simple matrix transpose program.

article shows that the constraint imposed by limited associativity can be tackled quite elegantly, allowing us to extend the results of the I/O model to the cache model very efficiently.

Most modern architectures have a memory hierarchy consisting of multiple cache levels. In the second half of this article, we extend the two-level cache model to a multilevel cache model.

The remainder of this article is organized as follows. Section 2 surveys related work. Section 3 defines our cache model and establishes an efficient emulation scheme between the I/O model and our cache model. As direct corollaries of the emulation scheme, we obtain cache-optimal algorithms for several fundamental problems such as sorting, FFT, and an important class of permutations. Section 4 illustrates the importance of the emulation scheme by demonstrating that a direct (i.e., bypassing the emulation) implementation of an I/O-optimal sorting algorithm (multiway mergesort) is both provably and empirically inferior, even in the average case, in the cache model. Section 5 describes a natural extension of our model to multiple levels of caches. We present an algorithm for transposing a matrix in the multilevel cache model that attains optimal performance in the presence of any number of levels of cache memory. Our algorithm is not cache-oblivious, that is, we do make explicit use of the sizes of the cache at various levels. Next, we show that with some simple modifications, the funnel-sort algorithm of Frigo et al. [1999] attains optimal performance in a single level (direct-mapped) cache in an oblivious sense, that is, no knowledge of memory parameters is required. Finally, Section 6 presents conclusions, possible refinements to the model, and directions for future work.

2. *Related Work*

The I/O model (discussed in greater detail in Section 3) assumes that most of the data resides on disk and has to be transferred to main memory to do any processing. Because of the tremendous difference in speeds, it ignores the cost of internal processing and counts only the number of I/O operations. Floyd [1972] originally defined a formal model and proved tight bounds on the number of I/O operations required to transpose a matrix using two pages of internal memory. Hong and Kung [1981] extended this model and studied the I/O complexity of FFT when the internal memory size is bounded by M. Aggarwal and Vitter [1988] further refined the model by incorporating an additional parameter B, the number of (contiguous) elements transferred in a single I/O operation. They gave upper and lower bounds on the number of I/Os for several fundamental problems including sorting, selection, matrix transposition, and FFT. Following their work, researchers have designed I/O-optimal algorithms for fundamental problems in graph theory [Chiang et al. 1995] and computational geometry [Goodrich et al. 1993].

Researchers have also modeled multiple levels of memory hierarchy. Aggarwal et al. [1987a] defined the *Hierarchical Memory Model* (HMM) that assigns a function $f(x)$ to accessing location x in the memory, where f is a monotonically increasing function. This can be regarded as a continuous analog of the multilevel hierarchy. Aggarwal et al. [1987b] added the capability of block transfer to the HMM, which enabled them to obtain faster algorithms. Alpern et al. [1994] described the *Uniform Memory Hierarchy* (UMH) model, where the access costs

differ in discrete steps. Very recently, Frigo et al. [1999] presented an alternate strategy of algorithm design on these models, which has the added advantage that explicit values of parameters related to different levels of the memory hierarchy are not required. Bilardi and Peserico [2001] investigate further the complexity of designing algorithms without the knowledge of architectural parameters. However, these models do not address the problem of limited associativity in cache. Other attempts were directed towards extracting better performance by parallel memory hierarchies [Aggarwal and Vitter 1988; Vitter and Nodine 1993; Vitter and Shriver 1994; Cormen et al. 1999], where several blocks could be transferred simultaneously.

Ladner et al. [1999] describe a stochastic model for performance analysis in cache. Our work is different in nature, as we follow a more traditional worst-case analysis. Our analysis of sorting in Section 4 provides a better theoretical basis for some of the experimental work of LaMarca and Ladner [1997].

To the best of our knowledge, the only other paper that addresses the problem of limited associativity in cache is recent work of Mehlhorn and Sanders [2000]. They show that for a class of algorithms based on merging multiple sequences, the I/O algorithms can be made nearly optimal by use of a simple randomized shift technique. Our Theorems 3.1 and 3.3 not only provide a deterministic solution for the same class of algorithms, but also work for more general situations. The results in [Sanders 1999] are nevertheless interesting from the perspective of implementation.

3. *The Cache Model*

The (two-level) I/O model of Aggarwal and Vitter [1988] captures the interaction between a slow (secondary) memory of infinite capacity and a fast (primary) memory of limited capacity. It is characterized by two parameters: M, the capacity of the fast memory; and B, the size of data transfers between slow and fast memories. Such data movement operations are called *I/O operations* or *block transfers*. As is traditional in classical algorithm analysis, the input problem size is denoted by N. The use of the model is meaningful when $N \gg M$.

The I/O model contains the following further assumptions.

(1) A datum can be used in a computation if and only if it is present in fast memory. All initial data and final results reside in slow memory. I/O operations transfer data between slow and fast memory (in either direction).

(2) Since the latency for accessing slow memory is very high, the average cost of transfer per element can be reduced by transferring a block of B elements at little additional cost. This may not be as useful as it may seem at first sight, since these B elements are not arbitrary, but are contiguous in memory. The onus is on the programmer to use all the elements, as traditional RAM algorithms are not necessarily designed for such restricted memory access patterns. We denote the map from a memory address to its block address by $\mathbb{B}$.[3] The internal memory can hold at least three blocks, that is, $M \geqslant 3 \cdot B$.

[3] The notion of *block address* corresponds to the notion of *track* in the I/O model [Aggarwal and Vitter 1988, Definition 3.2]. The different nomenclature reflects the terminology in common use in the underlying hardware technologies, namely, cache memory and disk.

(3) The computation cost is ignored in comparison to the cost of an I/O operation. This is justified by the high access latency of slow memory. However, classical algorithm analysis can be used to provide a measure of computational complexity.
(4) A block of data from slow memory can be placed in any block of fast memory (i.e., the user controls replacement policy).
(5) I/O operations are explicit in the algorithm.

The goal of algorithm design in this model is to minimize T, the number of I/O operations.

We adopt much of the framework of the I/O model in developing a cache model to capture the interactions between cache and main memory. In our case, the cache assumes the role of the fast memory, while main memory assumes the role of the slow memory. Assumptions (1) and (2) of the I/O model continue to hold in our cache model. However, assumptions (3)–(5) are no longer valid and need to be replaced as follows.

—*Lower Cache Latency*. The difference between the access times of slow and fast memory is considerably smaller than in the I/O model, namely a factor of 5–100 rather than factor of 10000. *We use an additional parameter L to denote the normalized cache latency.* This cost function assigns a cost of 1 for accessing an element in cache and a cost of L for accessing an element in main memory. In this way, we also account for computation cost in the cache model. We can consider the I/O model as the limiting case of the cache model as $L \to \infty$.

—*Limited Cache Associativity*. Main memory blocks are mapped into cache sets using a *fixed* and predetermined mapping function that is implemented in hardware. Typically, this is a modulo mapping based on the low-order address bits. (The results of this section hold for a larger class of address mapping functions that distribute the memory blocks evenly to the cache frames, although we do not attempt to characterize these functions here.) We denote this mapping from main memory blocks to cache sets by $\mathbb{S}$. We occasionally slightly abuse this notation and apply $\mathbb{S}$ directly to a memory address x rather than to $\mathbb{B}(x)$. *We use an additional parameter A in the model to represent this limited cache associativity*, as discussed in Section 1.

—*Cache Replacement Policy*. The replacement policy of cache sets is fixed and predetermined. *We assume an LRU replacement policy when necessary.*

—*Lack of Explicit Program Control over Cache Operation*. The cache is not directly visible to the programmer.[4] When a program accesses a memory location x, an *image* (copy) of the main memory block $b = \mathbb{B}(x)$ that contains location x is brought into the cache set $\mathbb{S}(b)$ if it is not already present there. The block b continues to reside in cache until it is evicted by some other block b' that is mapped to the same cache set (i.e., $\mathbb{S}(b) = \mathbb{S}(b')$). In other words, *a cache*

[4] Some modern processors, such as the IBM PowerPC, include cache-control instructions in their instruction set, allowing a program to prefetch blocks into cache, flush blocks from cache, and specify the replacement victim for the next access to a cache set. We leave such operations out of the scope of our cache model for two reasons: first, they are often privileged-mode instructions that user-level code cannot use; and second, they are often *hints* to the memory system that may be abandoned under certain conditions, such as a TLB miss.

set contains the most recently referenced A distinct memory blocks that map to it.

We use the notation $\mathfrak{C}(M, B, L, A)$ to denote our four-parameter cache model. The goal of algorithm design in this model is to minimize *running time*, defined as the number of cache accesses plus L times the number of main memory accesses. We use n and m to denote N/B and M/B, respectively.

The usual performance metric in the I/O model is T, the number of accesses to slow memory, while the performance metric in the cache model is a $I + L \cdot T$, where I is the number of accesses to fast memory. Since our intention is to relate the two models, we use one notational device to unify the two performance metrics. We redefine the performance metric in the I/O model to also be $I + L \cdot T$. Note that this is equivalent to the Aggarwal–Vitter I/O model [Aggarwal and Vitter 1988] under the condition $L \to \infty$. It is clear that an optimal algorithm in the original metric of the I/O model remains optimal under the modified metric. In summary, we shall use the notation $\mathfrak{I}(M, B, L)$ to denote the I/O model with parameters M, B, and L.

The assumptions of our cache model parallel those of the I/O model, except as noted above.[5] The differences between the two models listed above would appear to frustrate any efforts to naively map an I/O algorithm to the cache model, given that we neither have the control nor the flexibility of the I/O model. Indeed, executing an algorithm $\mathbb{A}$ designed for $\mathfrak{I}(M, B, L)$ unmodified in $\mathfrak{C}(M, B, L, A)$ does not guarantee preservation of the original I/O complexity, even when $A = M/B$ (a fully associative cache), because of the fixed LRU replacement policy of the cache model. Going the other way, however, is straightforward:

Remark 1. Any algorithm in $\mathfrak{C}(M, B, L, A)$ can be run unmodified in $\mathfrak{I}(M, B, L)$ without loss of efficiency.

In going from the I/O model to the cache model, we emulate the behavior of the I/O algorithm by maintaining a memory buffer that is the size of the cache. All computation is done out of this buffer, and I/O operations move data in and out of this buffer. However, since the copying implicit in an I/O operation goes through the cache, we need to ensure that an emulated I/O operation does not result in cache thrashing. To guarantee this property, we may need to copy data through an intermediate block. We now establish a bound on the cost of a block copy in the cache model.

LEMMA 3.1. *One memory block can be copied to another memory block in no more than* $3L + 2B$ *steps in* $\mathfrak{C}(M, B, L, A)$.

PROOF. Let a and b denote the two memory blocks. If $\mathbb{S}(a) \neq \mathbb{S}(b)$, then a copy of a to b costs no more than $2L + B$ steps: L steps to bring block a into cache, L steps to bring block b into cache, and B steps to copy data between the two cache-resident blocks. If $\mathbb{S}(a) = \mathbb{S}(b)$ and $A > 1$, then again the copy costs no more than $2L + B$ steps. If $\mathbb{S}(a) = \mathbb{S}(b)$ and $A = 1$, this naive method of copying will lead to *thrashing* and will result in a copy cost of $2BL$ steps. However, we can

[5] Frigo et al. [1999] independently arrive at a very similar parameterization of their model, except that their default model assumes full associativity.

avoid this situation by using a third memory block c such that $\mathbb{S}(a) \neq \mathbb{S}(c)$. A copy from a to b is accomplished by a copy from a to c followed by a copy from c to b, with cost at most $2L + B$ for the first copy and $L + B$ for the second. Thus, in all cases, $3L + 2B$ steps suffice to copy memory block a into memory block b. □

Remark 2. We henceforth use the term *bounded copy* to refer the block copying technique described in the proof of Lemma 3.1.

Remark 3. As a matter of practical interest, a possible alternative to using intermediate memory-resident buffers to avoid thrashing is to use machine registers, since register access is much faster. In particular, if we have B registers, then we can bring down the cost of bounded-copying to $2L + 2B$ in the problematic case of Lemma 3.1.

The idea of bounded copy presented above leads to a simple and generic emulation scheme that establishes a connection between the I/O model and the cache model. We first present the emulation scheme for direct-mapped caches ($A = 1$) in Section 3.1, and then extend it to the general set-associative case in Section 3.2.

3.1 EMULATING I/O ALGORITHMS: THE DIRECT-MAPPED CASE

THEOREM 3.1 (EMULATION THEOREM). *An algorithm $\mathbb{A}$ in $\mathfrak{I}(M, B, L)$ using T block transfers and I processing steps can be converted to an equivalent algorithm $\mathbb{A}^c$ in $\mathfrak{C}(M, B, L, 1)$ that runs in $O(I + (L + B) \cdot T)$ steps. The memory requirement of $\mathbb{A}^c$ is an additional $m + 2$ blocks beyond that of $\mathbb{A}$.*

PROOF. As indicated above, the cache algorithm $\mathbb{A}^c$ will emulate the behavior of the I/O algorithm $\mathbb{A}$ using an additional main memory buffer *Buf* of size M that serves as a "proxy" for main memory in the I/O model. More precisely, *Buf* must consist of m blocks that map to distinct cache sets. This property can always be satisfied under the assumptions of the model. In the common case where $\mathbb{S}$ is a modulo mapping, it suffices to have *Buf* consist of M contiguous memory locations starting at a memory address that is a multiple of M. Without loss of generality, we assume this scenario in the proof, as it simplifies notation considerably. The proof consists of two parts: the definition of the emulation scheme, and the accounting of costs in the cache model to establish the desired complexity bounds.

In the cache model, let $Mem[i]$ (with $0 \leqslant i < n$) denote the B-element block consisting of memory address x such that $\mathbb{B}(x) = i$, and let $Buf[j]$ (with $0 \leqslant j < m$) denote the B-element block consisting of memory addresses y such that $\mathbb{S}(y) = j$.[6] Partition the I/O algorithm $\mathbb{A}$ into T *rounds*, where round i is defined to consist of the ith block transfer and any computation performed between block transfers i and $i + 1$ (define program termination to be the $(T + 1)$st block transfer). Then the cache algorithm $\mathbb{A}^c$ will consist of T *stages*, defined as follows:

—If round i of $\mathbb{A}$ transfers disk block b_i to/from main memory block a_i, then stage i of $\mathbb{A}^c$ will *bounded-copy* the B elements of $Mem[b_i]$ to/from $Buf[a_i]$.

—If the computation in round i of $\mathbb{A}$ accesses a main memory block c_i, then stage i of $\mathbb{A}^c$ will access $Buf[c_i]$ and perform the same computation.

[6] Although *Buf* is a memory-resident data structure, that is, $\forall j : \exists k : Buf[j] = Mem[k]$, we use the different indexing schemes to emphasize the special role that *Buf* plays in the emulation scheme.

Assuming that $\mathbb{A}^c$ is actually a valid algorithm in the cache model, it is clear that its final outcome is the same as that of $\mathbb{A}$. In order for $\mathbb{A}^c$ to be a valid algorithm in the cache model, it is sufficient to maintain the invariant that *Buf* is cache-resident when the computations are performed. Only the *bounded-copy* operations can alter the cache residency of *Buf*. A single bounded copy can evict at most two blocks of *Buf* from the cache (the block mapping to the same set as the main memory block being copied, and the block mapping to the same set as the intermediate block used in the bounded copy), allowing the restoration of the desired invariant at cost $2L$.

Lemma 3.1 bounds the cost of the bounded copy at stage i of $\mathbb{A}^c$ to $3L+2B$ steps. The internal processing cost I_i of stage i of $\mathbb{A}^c$ is identical to that of round i of $\mathbb{A}$. Thus, the total cost of $\mathbb{A}^c$ is at most $\sum_{i=1}^{T}(I_i+3L+2B+2L) = I+5L\cdot T+2B\cdot T$.

Having two intermediate buffers mapping to distinct cache sets suffices for all cases of bounded copy. The additional memory requirement of $\mathbb{A}^c$ is therefore *Buf* and these two blocks, establishing the space bound. □

The basic idea of copying data into contiguous memory locations to reduce interference misses has been exploited before in some specific contexts like matrix multiplication [Lam et al. 1991] and bit-reversal permutation [Carter and Gatlin 1998]. theorem 3.1 unifies these previous results within a common framework.

The term $O(B \cdot T)$ is subsumed by $O(I)$ if computation is done on at least a constant fraction of the elements in the block transferred by the I/O algorithm. This is usually the case for efficient I/O algorithms. We call such I/O algorithms *block-efficient*.

COROLLARY 3.2. *A block-efficient I/O algorithm for $\mathfrak{I}(M, B, L)$ that uses T block transfers and I processing steps can be emulated in $\mathfrak{C}(M, B, L, 1)$ in $O(I + L \cdot T)$ steps.*

Remark 4. The algorithms for sorting, FFT, matrix transposition, and matrix multiplication described in Aggarwal and Vitter [1988] are block-efficient.

3.2. EXTENSION TO SET-ASSOCIATIVE CACHE. The emulation technique of the previous section would extend to the set-associative scenario easily if we had explicit control over replacement policy. This not being the case, we shall tackle it indirectly by making use of an useful property of LRU that Frigo et al. [1988] exploited in the context of designing cache-oblivious algorithms for a fully associative cache.

LEMMA 3.2 ([SLEATOR AND TARJAN 1985]). *For any sequence s, F_{LRU}, the number of misses incurred by LRU with cache size n_{LRU} is no more than $(\frac{n_{LRU}}{n_{LRU}-n_{OPT}+1}\cdot F_{OPT})$, where F_{OPT} is the minimum number of misses by an optimal replacement strategy with cache size n_{OPT}.*

We use this lemma in the following way. We run the emulation technique for only half the cache size, that is, we choose the buffer to be of total size $m/2$, such that for the A cache frames in a set, we have only $A/2$ buffer blocks. We can think of the buffer to be a set of $A/2$ arrays each having size equal to the number of cache sets.

We follow the same strategy as before—namely, we copy the blocks into the buffer corresponding to the block accesses of the I/O algorithm and perform computations on elements within the buffer. However, we cannot guarantee that the contents of a given cache set are in 1-1 correspondence with the corresponding

buffer blocks because of the (LRU) replacement policy in the cache. That is, some frame may be evicted from the cache that we do not intend to replace in the buffer. Since it is difficult to keep track of the contents of any given cache set explicitly, we analyze the above scenario in the following manner. Let the sequence of block accesses to the $A/2$ buffer blocks (to a given cache set) be $\sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_t\}$. Between these accesses there are computation steps involving blocks present in the buffer (but not necessarily in the cache set). In other words, there is a memory reference sequence σ' such that $\sigma \subset \sigma'$ is the set of *misses* from the $A/2$ buffer blocks with *explicit* replacement. We want to bound the number of misses from the corresponding A cache frames for the same sequence σ' under LRU replacement.

From Lemma 3.2, we know that the number of misses in each each cache set is no more than twice the optimal, which is in turn bounded by the number of misses incurred by the I/O algorithm, namely $|\sigma|$. Since any memory reference while copying to the buffer may cause an unwanted eviction from some cache set, we restore it by an extra read operation (as in the case of the proof of Theorem 3.1).

THEOREM 3.3 (GENERALIZED EMULATION THEOREM). *Any given algorithm $\mathbb{A}$ in $\mathfrak{I}(M/2, B, L)$ using T block transfers and I processing steps can be converted to an equivalent algorithm $\mathbb{A}^c$ in $\mathfrak{C}(M, B, L, A)$ that runs in $O(I + (L + B) \cdot T)$ steps. The memory requirement of $\mathbb{A}^c$ is an additional $m/2 + 2$ blocks beyond that of $\mathbb{A}$.*

3.3. THE CACHE COMPLEXITY OF SORTING AND OTHER PROBLEMS. We use the following lower bound for sorting and FFT in the I/O model.

LEMMA 3.3 ([AGGARWAL AND VITTER 1988]). *The average-case and worst-case number of I/Os required for sorting N records and for computing the N-input FFT digraph is*

$$\Theta\left(\frac{N}{B}\,\frac{\log(1 + N/B)}{\log(1 + M/B)}\right).^7$$

THEOREM 3.4. *The lower bound for sorting in $\mathfrak{C}(M, B, L, 1)$ is*

$$\Omega\left(N \log N + L \cdot \frac{N}{B} \cdot \frac{\log(1 + N/B)}{\log(1 + M/B)}\right).$$

PROOF. The Aggarwal–Vitter lower bound is information-theoretic and is therefore independent of the replacement policy in the I/O model. The lower bound on the number of block transfers in $\mathfrak{I}(M, B, L)$ therefore carries over to $\mathfrak{C}(M, B, L, 1)$. The lower bound in the cache model is the greater of the $\Omega(N \log N)$ lower bound on number of comparisons and L times the bound in Lemma 3.3, leading to the indicated complexity using the identity $\max\{a, b\} \geqslant (a + b)/2$. □

THEOREM 3.5. *N numbers can be sorted in $O(N \log N + L \cdot \frac{N}{B} \cdot \frac{\log(1+N/B)}{\log(1+M/B)})$ steps in $\mathfrak{C}(M, B, L, 1)$, and this is optimal.*

[7] We are setting $P = 1$ in the original statement of the theorem.

PROOF. The M/B-way mergesort algorithm described in Aggarwal and Vitter [1988] has an I/O complexity of $O(\frac{N}{B}\frac{\log(1+N/B)}{\log(1+M/B)})$.[8] The processing time involves maintaining a heap of size M/B and is $O(\log M/B)$ per output element. For N elements, the number of phases is $\frac{\log N}{\log M/B}$, so the total processing time is $O(N \log N)$. From Corollary 3.2, and Remark 4, the cost of this algorithm in the cache model is $O(N \log N + L \cdot \frac{N}{B} \cdot \frac{\log(1+N/B)}{\log(1+M/B)})$. Optimality follows from Theorem 3.4. □

We can prove a similar result for FFT computations.

THEOREM 3.6. *The FFT of N numbers can be computed in* $O(N \log N + L \cdot \frac{N \log(1+N/B)}{B \log(1+M/B)})$ *steps in* $\mathfrak{C}(M, B, L, 1)$.

Remark 5. The FFTW algorithm [Frigo and Johnson 1998] is optimal only for $B = 1$. Barve (R. Barve, private communication) has independently obtained a similar result.

The class of Bit Matrix Multiply Complement (BMMC) permutations include many important permutations like matrix transposition and bit reversal. A BMMC permutation is defined as $y = Ax$ X-OR c where y and x are binary represention of the source and destination addresses, c is a binary vector, and the computations are performed over $GF(2)$. Combining the work of Cormen et al. [1999] with our emulation scheme, we obtain the following result.

THEOREM 3.7. *The class of BMMC permutations for N elements can be achieved in* $\Theta(N + L \cdot \frac{N}{B}\frac{\log r}{\log(M/B)})$ *steps in* $\mathfrak{C}(M, B, L, 1)$. *Here r is the rank of submatrix* $A_{\log B \cdot\cdot \log N-1, 0 \cdot\cdot \log B}$, *that is,* $r \leqslant \log B$.

Remark 6. Many known geometric algorithms [Chiang et al. 1995] and graph algorithms [Goodrich et al. 1993] in the I/O model, such as convex hull and graph connectivity, can be transformed optimally into the cache model.

4. *The Utility of the Emulation Theorem*

Although the proof of Theorem 3.1 supplies a simple emulation scheme that is both universal and bounds-preserving, one can question its utility. In other words, one can ask the question: "How large a performance degradation might one expect if one were to run an I/O-optimal algorithm unmodified in the cache model?" In this section, we analyze the performance of the I/O-optimal k-way mergesort in the cache model and show that the result of bypassing of the emulation scheme is a cache algorithm that is asymptotically worse than the algorithm resulting from Theorem 3.1. Since it is easy to construct a worst-case input permutation where every access to an input element will result in a miss in the cache model (a cyclic distribution suffices), we use average-case analysis and demonstrate the result even for this measure of algorithmic complexity. Section 4.1 derives this result, while Section 4.2 provides experimental evidence on the validity of these results for a real machine.

[8] The M/B-way distribution sort (multiway quicksort) also has the same upper bound.

3.1. AVERAGE-CASE PERFORMANCE OF MULTIWAY MERGESORT IN THE CACHE MODEL. Of the three classes of misses described in Section 1, we note that compulsory misses are unavoidable and that capacity misses are minimized while designing algorithms for the I/O model. We are therefore interested in bounding the number of conflict misses for a straightforward implementation of the I/O-optimal k-way mergesort algorithm.

We assume that s cache sets are available for the leading blocks of the k runs $S_1, \ldots, S_k$. In other words, we ignore the misses caused by heap operations (or equivalently ensure that the heap area in the cache does not overlap with the runs).

We create a random instance of the input as follows: Consider the sequence $\{1, \ldots, N\}$, and distribute the elements of this sequence to runs by traversing the sequence in increasing order and assigning element i to run S_j with probability $1/k$. From the nature of our construction, each run S_i is sorted. We denote jth element of S_i as $S_{i,j}$. The expected number of elements in any run S_i is N/k.

During the k-way merge, the leading blocks are critical in the sense that the heap is built on the *leading element* of every sequence S_i. The leading element of a sequence is the smallest element that has not been added to the merged (output) sequence. The *leading block* is the cache line containing the leading element. Let b_i denote the leading block of run S_i. *Conflict* can occur when the leading blocks of different sequences are mapped to the same cache set. In particular, a *conflict miss* occurs for element $S_{i,j+1}$ when there is at least one element $x \in b_k$, for some $k \neq i$, such that $S_{i,j} < x < S_{i,j+1}$ and $\mathbb{S}(b_i) = \mathbb{S}(b_k)$. (We do not count conflict misses for the first element in the leading block, that is, $S_{i,j}$ and $S_{i,j+1}$ must belong to the same block, but we will not be very strict about this in our calculations.)

Let p_i denote the probability of conflict for element $i \in [1, N]$. Using indicator random variables X_i to count the conflict miss for element i, the total number of conflict misses $X = \sum_i X_i$. It follows that the expected number of conflict misses $E[X] = \sum_i E[X_i] = \sum_i p_i$. In the remaining section, we try to estimate a lower bound on p_i for i large enough to validate the following assumption.

A1. The cache sets of the leading blocks, $\mathbb{S}(b_i)$, are randomly distributed in cache sets $1, \ldots, s$ independent of the other sorted runs. Moreover, the exact position of the leading element within the leading block is also uniformly distributed in positions $\{1, \ldots, sB\}$.

Remark 7. A recent variation of the mergesort algorithm (see Barve et al. [1997]) actually satisfies assumption A1 by its very nature. So, the present analysis is directly applicable to its average-case performance in cache. A similar observation was made independently by Sanders [1999] who obtained upper-bounds for mergesort for a set associative cache.

From our previous discussion and the definition of a conflict miss, we would like to compute the probability of the following event.

E1. For some i, j, for all elements x, such that $S_{i,j} < x < S_{i,j+1}$, $\mathbb{S}(x) \neq \mathbb{S}(S_{i,j})$.

In other words, none of the leading blocks of the sorted sequences S_j, $j \neq i$, conflicts with b_i. The probability of the complement of this event (i.e., $\Pr[\overline{E1}]$) is the probability that we want to estimate. We compute an upper bound on $\Pr[E1]$, under Assumption A1, thus deriving a lower bound on $\Pr[\overline{E1}]$.

LEMMA 4.1. *For $k/s > \epsilon$, $\Pr[E1] < 1-\delta$, where ϵ and δ are positive constants (dependent only on s and k but not on n or B).*

PROOF. See Appendix A. □

Thus, we can state the main result of this section as follows:

THEOREM 4.1. *The expected number of conflict misses in a random input for doing a k-way merge in an s-set direct-mapped cache, where k is $\Omega(s)$, is $\Omega(N)$, where N is the total number of elements in all the k sequences. Therefore, the (ordinary I/O-optimal) M/B-way mergesort in an M/B-set cache will exhibit $O(N \frac{\log N/B}{\log M/B})$ cache misses, which is asymptotically larger than the optimal value of $O(\frac{N}{B} \frac{\log N/B}{\log M/B})$.*

PROOF. The probability of conflict misses is $\Omega(1)$ when k is $\Omega(s)$. Therefore, the expected total number of conflict misses is $\Omega(N)$ for N elements. The I/O-optimal mergesort uses M/B-way merging at each of the $\frac{\log N/B}{\log M/B}$ levels, hence the second part of the theorem follows. □

Remark 8. Intuitively, by choosing $k \ll s$, we can minimize the probability of conflict misses at the cost of an increased number of merge phases (and hence reduce running time). This underlines the critical role of conflict misses *vis-a-vis* capacity misses that forces us to use only a small fraction of the available cache. Recently, Sanders [1999] has shown that by choosing k to be $O(\frac{M}{B^{1+1/a}})$ in an a-way set associative cache with a modified version of mergesort of Barve et al. [1997], the expected number of conflict misses per phase can be bounded by $O(N/B)$.

In comparison, the use of the emulation theorem guarantees minimal worst-case conflict misses while making good use of cache.

3.2. EXPERIMENTAL RESULTS. The experiment described in this section pertains to the average-case behavior of the Aggarwal–Vitter k-way mergesort for a large problem of fixed size as k is varied, to present experimental evidence supporting Theorem 4.1. We present both the trend in conflict misses (as calculated by a cache simulator) and the trend in execution time (as measured on a real machine).

The experiment was performed on a single processor of an unloaded dual-processor Sun workstation, with 300 MHz UltraSPARC-II CPUs and a direct-mapped L1 data cache with 16 KB capacity and a block size of 32 bytes. The code for the k-way mergesort was written in C, and was compiled with the SUN-Wspro optimizing compiler with the `-fast` optimization flag. The problem size is 4.75×10^7 integer elements, and the input array is populated with a random permutation. The mergesort initially merges sorted runs of four elements, which are created using bubblesort. The merge degree is varied from 2 to 3446. The `cprof` cache simulator [Lebeck and Wood 1994] was used for cache simulation.

For the above values of problem parameters, the number of merge phases decreases as the merge degree k crosses certain thresholds, as shown in Table I. The threshold values should be kept in mind in interpreting the remaining data.

Figure 1 shows the number of conflict misses in the mergesort as a function of merge degree around the threshold points of Table I. It is seen that the number of conflict misses increases dramatically as the merge degree is increased. Figure 2 shows the actual execution times of the mergesort as a function of merge degree.

TABLE I. NUMBER OF MERGE PHASES AS A FUNCTION OF MERGE DEGREE k, FOR 4.75×10^7 ELEMENTS AND FOUR-ELEMENT INITIAL SORTED RUNS

Number of merge phases	Values of k
6	16–25
5	26–58
4	59–228
3	229–3446
2	3447–11874999

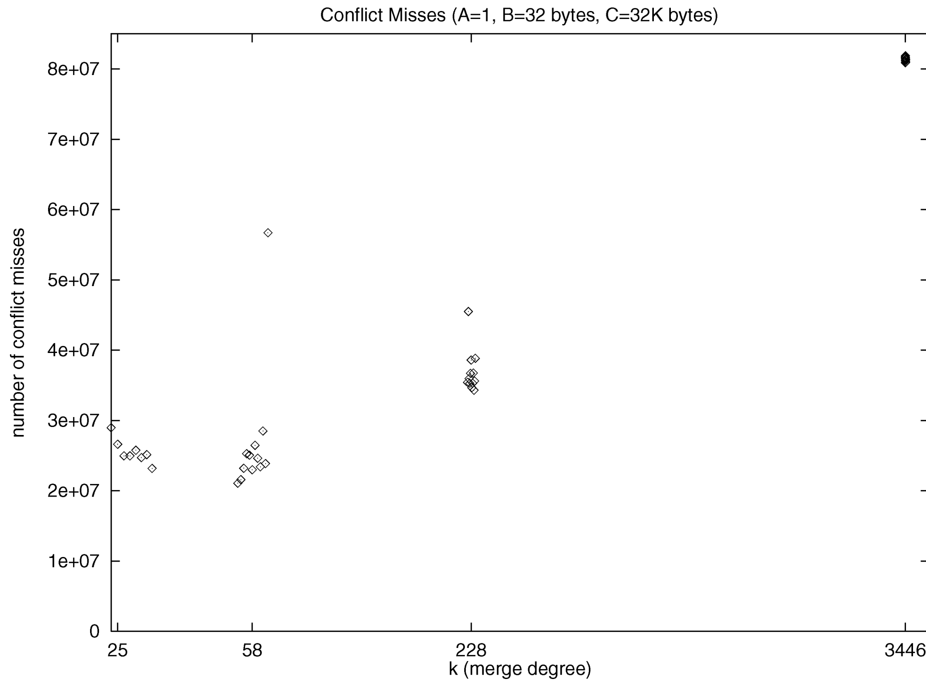

FIG. 1. Conflict misses as function of merge degree, for 4.75×10^7 elements and four-element initial sorted runs. Misses are shown at those values of merge degree where the number of merge phases change. Note how conflict misses increase even as the number of merge phases decrease.

It demonstrates that execution time does increase significantly as merge degree is increased, and that the best execution time occurs at a relatively small value of k.

5. *The Multi-Level Model*

Most modern architectures have a memory hierarchy consisting of multiple cache levels. Consider two cache levels $\mathcal{L}_1$ and $\mathcal{L}_2$ preceding main memory, with $\mathcal{L}_1$ being faster and smaller. The operation of the memory hierarchy in this case is as follows: The memory location being referenced is first looked up in $\mathcal{L}_1$. If it is not present in $\mathcal{L}_1$, then it is searched for in $\mathcal{L}_2$ (these can be overlapped with appropriate hardware support). If the item is not present in $\mathcal{L}_1$ but it is in $\mathcal{L}_2$, then it is brought into $\mathcal{L}_1$. In case that it is not in $\mathcal{L}_2$, then a cache line is brought into $\mathcal{L}_2$ and into $\mathcal{L}_1$. The size of cache line brought into $\mathcal{L}_2$ (denoted by B_2) is usually larger than the one

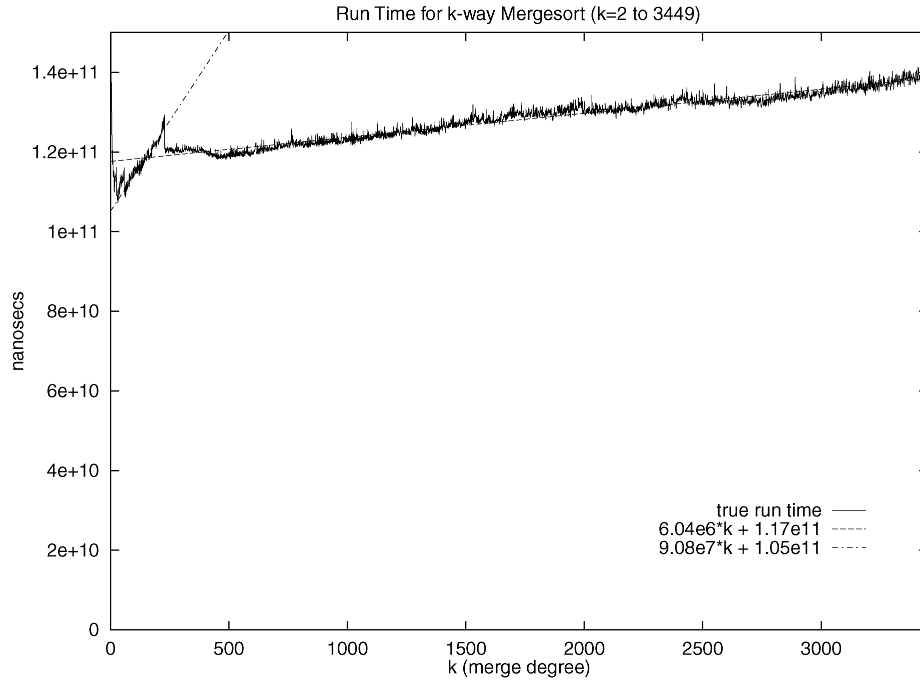

FIG. 2. Execution time as function of merge degree, for 4.75×10^7 elements and four-element initial sorted runs.

brought into $\mathcal{L}_1$ (denoted by B_1). The expectation is that the more frequently used items will remain in the faster cache.

The multilevel cache model is an extension to multiple cache levels of the previously introduced cache model. Let $\mathcal{L}_i$ denote the ith level of cache memory. The parameters involved here are the problem size N, the size of $\mathcal{L}_i$ which is denoted by M_i, the frame size (unit of data transfer) of $\mathcal{L}_i$ denoted by B_i, and the latency factor l_i. If a data item is present in the $\mathcal{L}_i$, then it is present in $\mathcal{L}_j$ for all $j \geqslant i$ (sometimes referred to as the *inclusion property*). If it is not present in $\mathcal{L}_i$, then the cost for a miss is l_i plus the cost of fetching it from $\mathcal{L}_{i+1}$ (if it is present in $\mathcal{L}_{i+1}$, then this cost is zero). For convenience, the latency factor l_i is the ratio of time taken on a miss from the ith level to the amount of time taken for a unit operation. Unless mentioned otherwise, we assume that all levels are direct-mapped.

The trivial lower bound for matrix transposition of an $N \times N$ matrix in the multilevel cache hierarchy is clearly the time to scan N^2 elements, namely,

$$\Omega\left(\sum_i \frac{N^2}{B_i} l_i\right),$$

where

B_i is the number of elements in one cache line in $\mathcal{L}_i$ cache
L_i is the number of cache lines in $\mathcal{L}_i$ cache, which is M_i/B_i
l_i is the latency factor for $\mathcal{L}_i$ cache.

This is the time to scan N^2 data items. Figure 3 shows the memory mapping for a two-level cache architecture. The shaded part of main memory is of size B_1

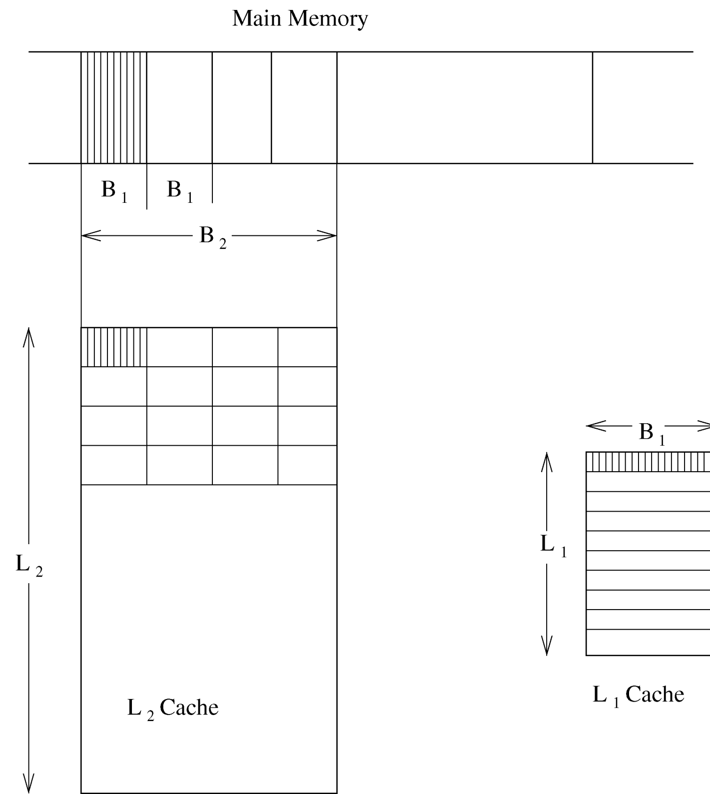

FIG. 3. Memory mapping in a two-level cache hierarchy.

and therefore it occupies only a part of a line of the $\mathcal{L}_2$ cache which is of size B_2. There is a natural generalization of the memory mapping to multiple levels of cache.

We make the following assumptions in this section that are consistent with the existing architectures. We use L_i to denote the number of cache frames in $\mathcal{L}_i (= M_i/B_i)$.

(A1) For all i, B_i, L_i are powers of 2

(A2) $2B_i \leqslant B_{i+1}$ and the number of cache lines $L_i \leq L_{i+1}$.

(A3) $B_k \leq L_1$ and $4B_k \leq B_1 L_1$ (i.e., $B_1 \geqslant 4$) where $\mathcal{L}_k$ is the largest and slowest cache. This implies that

$$L_i \cdot B_i \geqslant B_k \cdot B_i. \tag{1}$$

This will be useful for the analysis of the algorithms and are sometimes termed as *tall cache* in reference to the aspect ratio.

5.1. MATRIX TRANSPOSE. In this section, we provide an approach for transposing a matrix in a multilevel cache model. Our algorithm uses a more general form of the emulation theorem to get the submatrices to fit into cache in a regular fashion. The work in this section shows that it is possible to handle the constraints imposed by limited associativity even in a multilevel cache model.

We subdivide the matrix into $B_k \times B_k$ submatrices. Thus, we get $\lceil n/B_k \rceil \times \lceil n/B_k \rceil$ submatrices from an $n \times n$ submatrix.

$$A = \begin{pmatrix} a_1 & a_2 & \dots & \dots & a_n \\ a_{n+1} & a_{n+2} & \dots & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n^2-n+1} & \dots & \dots & \dots & a_{n^2} \end{pmatrix} = \begin{pmatrix} A_1 & A_2 & \dots & A_{n/B} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n^2-nB/B} & \dots & \dots & A_{n^2/B^2} \end{pmatrix}.$$

Note that the submatrices in the last row and column need not be square as one side may have $\leq B_k$ rows or columns.

Let $m = \lceil n/B_k \rceil$; then

$$A^T = \begin{pmatrix} A_1^T & A_{m+1}^T & \dots & \dots & A_{m^2-m+1}^T \\ A_2^T & A_{m+2}^T & \dots & \dots & A_{2m}^T \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ A_m^T & \dots & \dots & \dots & A_{m^2}^T \end{pmatrix}.$$

For simplicity, we describe the algorithm as transposing a square matrix A in another matrix B, that is, $B = A^T$. The main procedure is **Rec_Trans**(A, B, s), where A is transposed into B by dividing A (B) into s^2 submatrices and then recursively transposing the submatrices. Let $A_{i,j}$ $(B_{i,j})$ denote the submatrices for $1 \leqslant i, j \leqslant s$. Then $B = A^T$ can be computed as **Rec_Trans**$(A_{i,j}, B_{j,i}, s')$ for all i, j and some appropriate s', which depends on B_k and B_{k-1}. In general, if $t_k, t_{k-1} \cdots t_1$ denote the values of s' at the $1, 2 \cdots$ level of recursion, then $t_i = B_{i+1}/B_i$. If the submatrices are $B_1 \times B_1$ (base case), then perform the transpose exchange of the symmetric submatrices directly. We perform matrix transpose as follows, which is similar to the familiar recursive transpose algorithm:

(1) Subdivide the matrix as shown into $B_k \times B_k$ submatrices.
(2) Move the symmetric submatrices to contiguous memory locations.
(3) **Rec_Trans**$(A_{i,j}, B_{j,i}, B_k/B_{k-1})$.
(4) Write back the $B_k \times B_k$ submatrices to original locations.

In the following sections, we analyze the data movement of this algorithm to bound the number of cache misses at various levels.

5.2. MOVING A SUBMATRIX TO CONTIGUOUS LOCATIONS. To move a submatrix, we move it cache line by cache line. By choice of size of submatrices $(B_k \times B_k)$, each row will be an array of size B_k, but the rows themselves may be far apart.

LEMMA 5.1. *If two cache lines x, y of size B_k are aligned in $\mathcal{L}_k$ cache map to the same cache lines in $\mathcal{L}_i$ cache for some $1 \leq i \leq k$, then x and y map to the same lines in each $\mathcal{L}_j$ cache for all $1 \leq j \leq i$.*

PROOF. If x and y map to the same cache lines in $\mathcal{L}_i$ cache then their ith level memory block numbers (to be denoted by $b^i(x)$ and $b^i(y)$) differ by a multiple of L_i. Let $b^i(x) - b^i(y) = \alpha L_i$. Since $L_j | L_i$ (both are powers of two), $b^i(x) - b^i(y) = \beta L_j$ where $\beta = \alpha \cdot L_i / L_j$. Let x', y' be the *corresponding* subblocks of x and y at the jth level. Then their block numbers $b^j(x')$, $b^j(y')$ differ by $B_i / B_j \cdot \beta \cdot L_j$, that is, a multiple of L_j as $B_j | B_i$. Note that blocks are aligned across different levels of cache. Therefore, x and y also collide in $\mathcal{L}_j$. □

COROLLARY 5.1. *If two blocks of size B_k that are aligned in $\mathcal{L}_k$ Cache do not conflict in level i, they do not conflict in any level j for all $i \le j \le k$.*

THEOREM 5.2. *There is an algorithm that moves a set of blocks of size B_k (where there are k levels of cache with block size B_i for each $1 \le i \le k$) into a contiguous area in main memory in*

$$O\left(\sum \frac{N}{B_i} l_i\right),$$

where N is the total data moved and l_i is the cost of a cache miss for the ith level of cache.

PROOF. Let the set of blocks of size B_k be I (we are assuming that the blocks are aligned). Let the target block in the contiguous area for each block $i \in I$ be in the corresponding set J where each block $j \in J$ is also aligned with a cache line in $\mathcal{L}_k$ cache.

Let block a map to $R_{b,a}$, $b = \{1, 2, \ldots, k\}$ where $R_{b,a}$ denote the set of cache lines in the $\mathcal{L}_b$ cache. (Since a is of size B_k, it will occupy several blocks in lower levels of cache).

Let the ith block map to line $R_{k,i}$ of the $\mathcal{L}_k$ cache. Let the target block j map to line $R_{k,j}$. In the worst case, $R_{k,j}$ is equal to $R_{k,i}$. Thus, in this case, the line $R_{k,i}$ has to be moved to a temporary block say x (mapped to $R_{k,x}$) and then moved back to $R_{k,j}$. We choose x such that $R_{1,x}$ and $R_{1,i}$ do not conflict and also $R_{1,x}$ and $R_{1,j}$ do not conflict. Such a choice of x is always possible because our temporary storage area X of size $4B_k$ has at least four lines of $\mathcal{L}_k$ cache (i and j will take up two blocks of $\mathcal{L}_k$ cache thus leaving at least one block free to be used as temporary storage). Recall that we had assumed that $4B_k \le B_1 L_1$. That is, by dividing the $\mathcal{L}_1$ cache into $B_1 L_1 / B_k$ zones, there is always a zone free for x.

For convenience of analysis, we maintain the *invariant that X is always in $\mathcal{L}_k$ cache*. By application of the previous corollary on our choice of x (such that $R_{1,i} \ne R_{1,x} \ne R_{1,j}$) we also have $R_{a,i} \ne R_{a,x} \ne R_{a,j}$ for all $1 \le a \le k$. Thus, we can move i to x and x to j without any conflict misses. The number of cache misses involved is three for each level: one for getting ith block, one for writing the jth block, and one for maintaining the invariant since we have to touch the line displaced by i. Thus, we get a factor of 3.

Thus, the cost of this process is

$$3\left(\sum \frac{N}{B_i} l_i\right),$$

where N is the amount of data moved.

For blocks I that are not aligned in $\mathcal{L}_k$ cache, the constant would increase to 4 since we would need to bring up to two cache lines for each $i \in I$. The rest of the proof would remain the same. □

COROLLARY 5.3. *A $B_k \times B_k$ submatrix can be moved into contiguous locations in the memory in $O(\sum_{i=1}^{i=k} \frac{B_k{}^2}{B_i} l_i)$ time in a computer that has k levels of (direct-mapped) cache.*

This follows from the preceding discussion. We allocate memory C of size $B_k \times B_k$ for placing the submatrix and memory, X of size $4B_k$ for temporary storage and keep both these areas distinct.

Remark 9

(1) If we have set associativity ($\geq$2) in all levels of cache, then we do not need an intermediate buffer x as line i and j can both reside in cache simultaneously and movement from one to the other will not cause thrashing. Thus, the constant will come down to two. Since, at any point in time, we will only be dealing with two cache lines and will not need the lines i or j once we have read or written to them the replacement policy of the cache does not affect our algorithm.

(2) If the number of registers is greater than the size of the cache line (B_k) of the outermost cache level ($\mathcal{L}_k$) then we can move data without worrying about collision by copying from line i to registers and then from registers to line j. Thus, even in this, the constant will come down to two.

Once we have the submatrices in contiguous locations, we perform the transpose as follows:

For each of the submatrices, we divide the $B_r \times B_r$ submatrix (say S) in level $\mathcal{L}_r$ (for $2 \leq r \leq k$) further into $B_{r-1} \times B_{r-1}$ size submatrices as before. Each $B_{r-1} \times B_{r-1}$ size subsubmatrix fits into $\mathcal{L}_{r-1}$ cache completely (since $B_{r-1} \cdot B_{r-1} \leqslant B_{r-1} \cdot B_k \leqslant B_{r-1} \cdot L_{r-1}$ from Eq. (1)). Let $B_r/B_{r-1} = k_r$.

Thus, we have the sub matrices as

$$\begin{pmatrix} S_{1,1} & S_{1,2} & \dots & S_{1,k_r} \\ \vdots & \vdots & \vdots & \vdots \\ S_{k_r,1} & \dots & \dots & S_{k_r,k_r} \end{pmatrix}.$$

So we perform matrix transpose of each $S_{i,j}$ in place without incurring any misses as it resides completely inside the cache. Once we have transposed each $S_{i,j}$, we exchange $S_{i,j}$ with $S_{j,i}$. We show that $S_{i,j}$ and $S_{j,i}$ cannot conflict in $\mathcal{L}_{r-1}$ cache for $i \neq j$.

The rows of $S_{i,j}$ and $S_{j,i}$ correspond to $(iB_{r-1}+a_1)\,k_r + j$ and $(jB_{r-1}+a_2)\,k_r + i$ B_{r-1} sized blocks where $a_1, a_2 \in \{1, 2.\dots B_{r-1}\}$ and

$$\frac{B_r}{B_{r-1}} = k_r.$$

If these conflict in $\mathcal{L}_{r-1}$, then

$$(iB_{r-1} + a_1)\,k_r + j \equiv (jB_{r-1} + a_2)\,k_r + i (mod L_{r-1}).$$

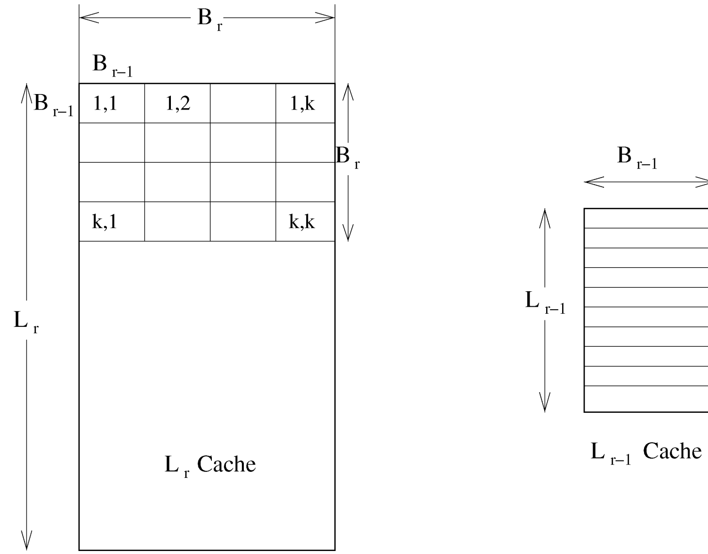

FIG. 4. Positions of symmetric submatrices in Cache.

Since $B_{r-1} = 2^u$ and $B_r = 2^v$ and $L_{r-1} = 2^w$ (all powers of two),

$$k_r = 2^{v-u}.$$

Therefore, k_r divides $\mathcal{L}_{r-1}$ (because $k_r = B_r/B_{r-1} < B_r \leq L_{r-1}$). Hence,

$$j \equiv i(mod k_r).$$

.

Since $i, j \leq k_r$, the above implies

$$i = j.$$

Note that $S_{i,i}$'s do not have to be exchanged. Thus, we have shown that a $B_r \times B_r$ matrix can be divided into $B_{r-1} \times B_{r-1}$, which completely fits into $\mathcal{L}_{r-1}$ cache. Moreover, the symmetric submatrices do not interfere with each other. The same argument can be extended to any $B_j \times B_j$ submatrix for $j < r$. Applying this recursively, we end up dividing the $B_k \times B_k$ size matrix in $\mathcal{L}_k$ cache to $B_1 \times B_1$ sized submatrices in $\mathcal{L}_1$ cache that can then be transposed and exchanged easily. From the preceding discussion, the corresponding submatrices do not interfere in any level of the cache (see Figure 4).

Note. Even though we keep subdividing the matrix at every cache level recursively and claim that we then have the submatrices in cache and can take the transpose and exchange them, the actual movement, that is, transpose and exchange happens only at the $\mathcal{L}_1$ cache level where the submatrices are of size $B_1 \times B_1$.

The time taken by this operation is

$$\sum \frac{N^2}{B_i} l_i.$$

This is because each $S_{i,j}$ and $S_{j,i}$ pair (such that $i \neq j$) has to be brought into $\mathcal{L}_{r-1}$ cache only once for transposing and exchanging of $B_1 \times B_1$ submatrices.

Similarly, at any level of cache, a block from the matrix is brought in only once. The sequence of the recursive calls ensures that cache-line is used completely as we move from submatrix to submatrix.

Lastly, we move the transposed symmetric sub matrices of size $B_k \times B_k$ to their location in memory, that is, reverse the process of bringing in blocks of size B_k from random locations to a contiguous block. This procedure is exactly the same as in Theorem 5.2 in the previous section that has the constant 3.

Remark 10

(1) The above constant of 3 for writing back the matrix to an appropriate location depends on the assumption that we can keep the two symmetric submatrices of size $B_k \times B_k$ in contiguous locations at the same time. This would allow us to exchange the matrices during the write back stage. If we are restricted to a contiguous temporary space of size $B_k \times B_k$ only, then we will have to move the data twice, incurring the cost twice.

(2) Even though, in the above analysis, we have always assumed a square matrix of size $N \times N$ the algorithm works correctly without any change for transposing a matrix of size $M \times N$ if we are transposing a matrix A and storing it in B. This is because the same analysis of subdividing into submatrices of size $B_k \times B_k$ and transposing still holds. However, if we want to transpose a $M \times N$ matrix in place, then the algorithm fails because the location to write back to would not be obvious and the approach used here would fail.

THEOREM 5.4. *The algorithm for matrix transpose runs in*

$$O\left(\sum_{i=1}^{i=k} \frac{N^2}{B_i} l_i\right) + O(N^2)$$

steps in a computer that has k levels of direct-mapped cache memory.

If we have temporary storage space of size $2B_k \times B_k + 4B_k$ and assume block alignment of all submatrices, then the constant is 7. This includes 3 for initial movement to contiguous location, 1 for transposing the symmetric submatrices of size $B_k \times B_k$ and 3 for writing back the transposed submatrix to its original location. Note that the constant is independent of the number of levels of cache.

Even if we have set associativity (≥ 2) in any level of cache the analysis goes through as before (though the constants will come down for data copying to contiguous locations). For the transposing and exchange of symmetric submatrices, the set associativity will not come into play because we need a line only once in the cache and are using only two lines at a given time. So either LRU or even FIFO replacement policy would only evict a line that we have already finished using.

5.3. SORTING IN MULTIPLE LEVELS. We first consider a restriction of the model described above where data cannot be transferred simultaneously across nonconsecutive cache levels. We use C_i to denote $\sum_{j=1}^{j=i} M_j$.

THEOREM 5.5. *The lower bound for sorting in the restricted multilevel cache model is* $\Omega(N \log N + \sum_{i=1}^{k} \ell_i \cdot \frac{N}{B_i} \frac{\log N/B_i}{\log C_i/B_i})$.

PROOF. The proof of Aggarwal and Vitter [1988] can be modified to disregard block transfers that merely rearrange data in the external memory. Then it can be

applied separately to each cache level, noting that the data transfer in the higher levels do not contribute for any given level. □

These lower bounds are in the same spirit as those of Vitter and Nodine [1993] (for the S-UMH model) and Savage [1995], that is, the lower bounds do not capture the simultaneous interaction of the different levels.

If we remove this restriction, then the following can be proved along similar lines as Theorem 3.4.

LEMMA 5.2. *The lower bound for sorting in the multilevel cache model is*

$$\Omega\left(\max_{i=1}^{k}\left\{N\log N,\, \ell_i \cdot \frac{N\cdot\log N/B_i}{B_i\log C_i/B_i}\right\}\right).$$

This bound appears weak if k is large. To rectify this, we observe the following: Across each cache boundary, the minimum number of I/Os follow from Aggarwal and Vitter's arguments. The difficulty arises in the multilevel model as a block transfer in level i propagates in all levels $j < i$ although the block sizes are different. The minimum number of I/Os from (the highest) level k remains unaffected, namely, $\frac{N}{B_k}\frac{\log N/B_k}{\log C_k/B_k}$. For level $k-1$, we subtract this number from the lower bound of $\frac{N}{B_{k-1}}\frac{\log N/B_{k-1}}{\log C_{k-1}/B_{k-1}}$. Continuing in this fashion, we obtain the following lower bound.

THEOREM 5.6. *The lower bound for sorting in the multilevel cache model is*

$$\Omega\left(N\log N + \sum_{i=1}^{k}\ell_i\cdot\left(\frac{N\cdot\log N/B_i}{B_i\log C_i/B_i} - \left(\sum_{j=i+1}^{k}\frac{N\cdot\log N/B_j}{B_j\log C_j/B_j}\right)\right)\right).$$

If we further assume that $\frac{C_i}{C_{i-1}} \geqslant \frac{B_i}{B_{i-1}} \geqslant 3$, we obtain a relatively simple expression that resembles Theorem 5.5. Note that the consecutive terms in the expression in the second summation of the previous lemma decrease by a factor of 3.

COROLLARY 5.7. *The lower bound for sorting in the multilevel cache model with geometrically decreasing cache sizes and cache lines is* $\Omega(N\log N + \frac{1}{2}\sum_{i=1}^{k}\ell_i\cdot\frac{N\cdot\log N/B_i}{B_i\log C_i/B_i})$.

THEOREM 5.8. *In a multilevel cache, where the B_i blocks are composed of B_{i-1} blocks, we can sort in expected time* $O(N\log N + (\frac{\log N/B_1}{\log M_1/B_1})\cdot\sum_{i=1}^{k}\ell_i\cdot\frac{N}{B_i})$.

PROOF. We perform a M_1/B_1-way mergesort using the variation proposed by Barve et al. [1997] in the context of parallel disk I/Os. The main idea is to shift each sorted stream cyclically by a random amount R_i for the ith stream. If $R_i \in [0, M_k-1]$, then the leading element is in any of the cache sets with equal likelihood. Like Barve et al. [1997], we divide the merging into phases where a phase outputs m elements, where m is the merge degree. In the previous section, we counted the number of conflict misses for the input streams, since we could exploit symmetry based on the random input. It is difficult to extend the previous arguments to a worst case input. However, it can be shown easily that if $\frac{m}{s} < \frac{1}{m^3}$ (where s is the number of cache sets), the expected number of conflict misses is $O(1)$ in each phase. So the total expected number of cache misses is $O(N/B_i)$ in the level i cache for all $1 \leqslant i \leqslant k$.

The cost of writing a block of size B_1 from level k is spread across several levels. The cost of transferring B_k/B_1 blocks of size B_1 from level k is $\ell_k + \ell_{k-1}\frac{B_k}{B_{k-1}} + \ell_{k-2}\frac{B_k}{B_{k-1}}\frac{B_{k-1}}{B_{k-2}} + \cdots + \ell_1\frac{B_k}{B_1}$. Amortizing this cost over B_k/B_1 transfers gives us the required result. Recall that $O(N/B_1(\frac{\log N/B_1}{\log M_1/B_1}))$ B_1 block transfers suffice for $(M_1/B_1)^{1/3}$-way mergesort. □

Remark 11. This bound is reasonably close to Corollary 5.7 if we ignore constant factors. Extending this to the more general emulation scheme of Theorem 3.1 is not immediate as we require the block transfers across various cache boundaries to have a nice pattern, namely the *subblock* property. This is satisfied by the mergesort and quicksort and a number of other algorithms but cannot be assumed in general.

5.4. CACHE-OBLIVIOUS SORTING. In this section, we focus on two-level Cache model that has limited associativity. One of the *cache-Oblivious* algorithms presented by Frigo et al. [1999] is the Funnel Sort algorithm. They showed that the algorithm is optimal in the I/O Model (which is fully associative). However, it is not clear whether the optimality holds in the Cache Model. In this section, we show that with some simple modification the Funnel Sort is optimal even in the direct-mapped Cache Model.

The funnel sort algorithm can be described as follows:

—Split the input into $n^{1/3}$ contiguous arrays of size $n^{2/3}$ and sort these arrays recursively.

—Merge the $n^{1/3}$ sorted sequences using a $n^{1/3}$-merger, where a k-merger works as follows.

A k-merger operates by recursively merging sorted sequences. Unlike mergesort a k-merger stops working on a merging subproblem when the merged output sequence becomes "long enough" and it resumes working on another merging subproblem (see Figure (5)).

Invariant. The invocation of a k-merger outputs the first k^3 elements of the sorted sequence obtained by merging the k input sequences.

Base Case. $k = 2$ producing $k^3 = 8$ elements whenever invoked.

Note. The intermediate buffers are twice in size than the output obtained by a $k^{1/2}$ merger.

To output k^3 elements, a $k^{1/2}$-merger is invoked $k^{3/2}$ times. Before each invocation the $k^{1/2}$-merger fills each buffer that is less than half full so that every buffer has at least $k^{3/2}$ elements—the number of elements to be merged in that invocation.

Frigo et al. [1999] have shown that the above algorithm (that does not make explicit use of the various memory-size parameters) is optimal in the I/O Model. However, the I/O Model does not account for conflict misses since it assumes full associativity. This could be a degrading influence in the presence of limited-associativity (in particular direct-mapping).

5.4.1. *Structure of k-Merger.* It is sufficient to get a bound on cache misses in the Cache Model since the bounds for capacity misses in the Cache Model are the same as the bounds shown in the I/O Model.

Let us get an idea of what the structure of a k-merger looks like by looking at a 16-merger (see Figure 6). A k-merger, unrolled consists of 2-mergers arranged in

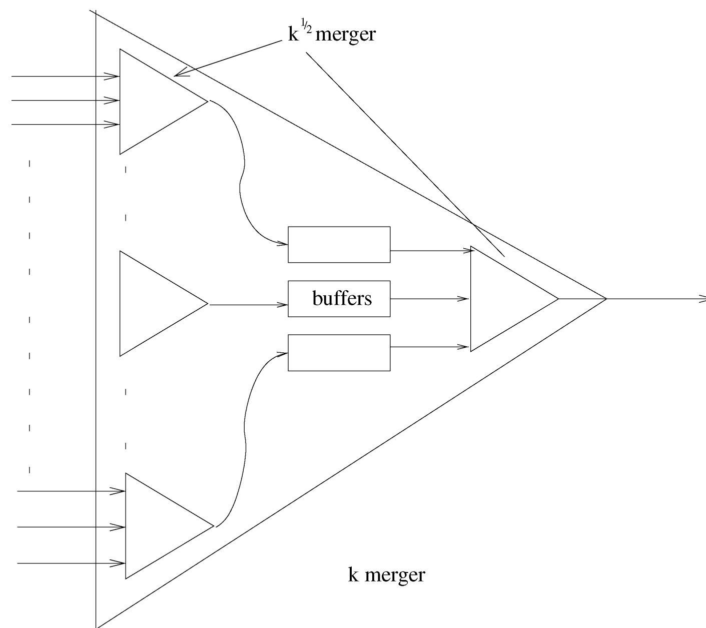

FIG. 5. Recursive definition of a k-merger in terms of $k^{1/2}$-mergers.

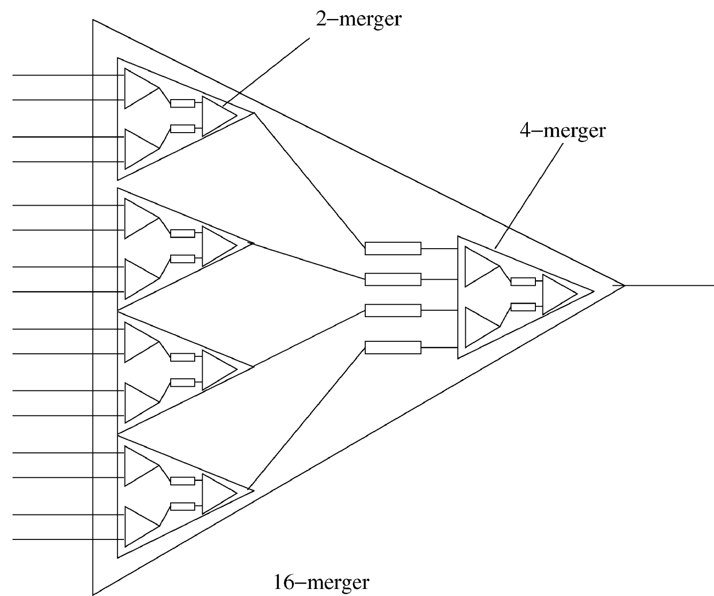

FIG. 6. Expansion of a 16-merger into 2-mergers.

a tree like fashion. Since the number of 2-mergers gets halved at each level and the initial input sequences are k in number there are $\log k$ levels.

LEMMA 5.3. *If the buffers are randomly placed and the starting position is also randomly chosen (since the buffers are cyclic this is easy to do) the probability of conflict misses is maximized if the buffers are less than one cache line long.*

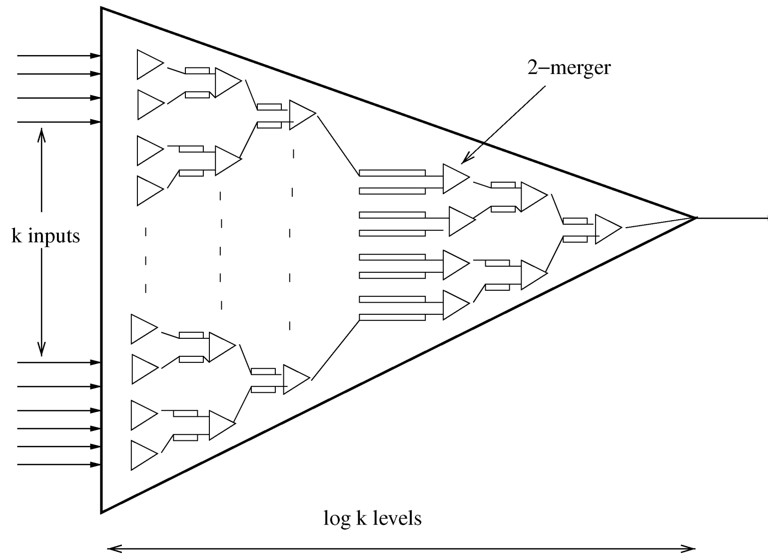

FIG. 7. A k-merger expanded out into 2-mergers.

The worst case for conflict misses occurs when the buffers are less than one cache line in size. This is because, if the buffers collide, then all data that goes through them will thrash. If, however, the size of the buffers were greater than one cache line, then even if some two elements collide the probability of future collisions would depend upon the data input or the relative movement of data in the two buffers. The probability of conflict miss is maximized when the buffers are less than one cache line. Then probability of conflict is $1/m$, where m is equal to the Cache Size M divided by the Cache Line Size B, that is, the number of Cache Lines.

5.4.2. *Bounding Conflict Misses.* The analysis for compulsory and capacity misses goes through without change from the I/O Model to the Cache Model. Thus, Funnel Sort is Cache Model Optimal if the conflict misses can be bounded by

$$\frac{N}{B} \times \frac{\log N/B}{\log M/B}.$$

LEMMA 5.4. *If the cache is* 3*-way or more set associative, there will be no conflict misses for a* 2*-way merger.*

PROOF. The two input buffers and the output buffer, even if they map to the same cache set can reside simultaneously in the cache. Since, at any stage only one 2-merger is active, there will be no conflict misses at all and the cache misses will only be in the form of capacity or compulsory misses. □

5.4.3. *Direct-Mapped Case.* For an input of size N, a $N^{1/3}$-merger is created. The number of levels in such a merger is $\log N^{1/3}$ (i.e., the number of levels of the tree in the unrolled merger). Every element that travels through the $N^{1/3}$-merger sees $\log N^{1/3}$ 2-mergers (see Figure 7). For an element passing through a 2-merger, there are three buffers that could collide. We *charge* an element for a conflict miss if it is swapped out of the cache before it passes to the output buffer or collides

with the output buffer when it is being output. So the expected number of collisions is ${}^3C_2$ times the probability of collision between any two buffers (two input and one output). Thus, the expected number of collisions for a single element passing through a 2-merger is ${}^3C_2 \times 1/m \leq 3/m$ where $m = M/B$.

If $x_{i,j}$ is the probability of a cache miss for element i in level j, then, summing over all elements and all levels, we get

$$\begin{aligned} E\left(\sum_{i=1}^{N}\sum_{j=1}^{\log N^{1/3}} x_{i,j}\right) &= \sum_{i=1}^{N}\sum_{j=1}^{\log N^{1/3}} E(x_{i,j}) \\ &\leq \sum_{i=1}^{N}\sum_{j=1}^{\log N^{1/3}} \frac{3}{m} = \frac{3N}{m} \times \log N^{1/3} \\ &= O\left(\frac{N}{m} \times \log N\right). \end{aligned}$$

LEMMA 5.5. *The expected performance of Funnel Sort is optimal in the direct-mapped Cache Model if* $\log M/B \leq M/(B^2 \log B)$. *It is also optimal for a* 3*-way associative cache.*

PROOF. If M and B are such that

$$\log \frac{M}{B} \leq \frac{M}{B^2 \log B},$$

we have the total number of conflict misses

$$\frac{N \log N}{m} = \frac{N \log N}{B \log B \dfrac{M}{B^2 \log B}} \leq \frac{N}{B} \times \frac{\log N/B}{\log M/B}.$$

Note that the condition is satisfied for $M \geqslant B^{2+\epsilon}$ for any fixed $\epsilon > 0$, which is similar to the *tall-cache* assumption made by Frigo et al. [1999].

The set associative case is proved by Lemma 5.4. □

The same analysis is applicable between successive levels $\mathcal{L}_i$ and $\mathcal{L}_{i+1}$ of a multilevel Cache model since the algorithm does not use the parameter values explicitly.

6. *Conclusions*

We have presented a cache model for designing and analyzing algorithms. Our model, while closely related to the I/O model of Aggarwal and Vitter, incorporates four additional salient features of cache: lower miss penalty, limited associativity, fixed replacement policy, and lack of direct program control over data movement. We have established an emulation scheme that allows us to systematically convert an I/O-efficient algorithm into a cache-efficient algorithm. This emulation provides a generic starting point for cache-conscious algorithm design; it may be possible to further improve cache performance by problem-specific techniques to control conflict misses. We have also established the relevance of the emulation scheme by demonstrating that a direct mapping of an I/O-efficient algorithm does not guarantee

a cache-efficient algorithm. Finally, we have extended our basic cache model to multiple cache levels.

Our single-level cache model is based on a blocking cache that does not distinguish between reads and writes. Modeling a nonblocking cache or distinguishing between reads and writes would appear to require queuing-theoretic extensions and does not appear to be appropriate at the algorithm design stage. The *translation lookaside buffer* (TLB) is another important cache in real systems that caches virtual-to-physical address translations. Its peculiar aspect ratio and high miss penalty raise different concerns for algorithm design. Our preliminary experiments with certain permutation problems suggests that TLBs are important to model and can contribute significantly to program running times. It also appears that the presence of prefetching in the memory hierarchy can have a profound effect on algorithm design and analysis.

We have begun to implement some of these algorithms to validate the theory on real machines, and also using cache simulation tools like *fast-cache*, ATOM, or *cprof*. Preliminary observations indicate that our predictions are more accurate with respect to miss ratios than actual running times (see Chatterjee and Sen [2000]). We have traced a number of possible reasons for this. First, because the cache miss latencies are not astronomical, it is important to keep track of the constant factors. An algorithmic variation that guarantees lack of conflict misses at the expense of doubling the number of memory references may turn out to be slower than the original algorithm. Second, our preliminary experiments with certain permutation problems suggests that TLBs are important to model and can contribute significantly to program running times. Third, several low-level details hidden by the compiler related to instruction scheduling, array address computations, and alignment of data structures in memory can significantly influence running times. As argued earlier, these factors are more appropriate to tackle at the level of implementation than algorithm design.

Several of the cache problems we observe can be traced to the simple array layout schemes used in current programming languages. It has shown elsewhere [Chatterjee et al. 1999a, 1999b; Thottethodi et al. 1998] that nonlinear array layout schemes based on quadrant-based decomposition are better suited for hierarchical memory systems. Further study of such array layouts is a promising direction for future research.

Appendix

A. *Approximating Probability of Conflict*

Let μ be the number of elements between $S_{i,j}$ and $S_{i,j+1}$, that is, one less than the difference in ranks of $S_{i,j}$ and $S_{i,j+1}$. (μ may be 0, which guarantees event E1.) Let E_m denote the event that $\mu = m$. Then $\Pr[E1] = \sum_m \Pr[E1 \cap E_m]$, since E_m's are disjoint. For each m, $\Pr[E1 \cap E_m] = \Pr[E1|E_m] \cdot \Pr[E_m]$. The events E_m correspond to a geometric distribution, that is,

$$\Pr[E_m] = \Pr[\mu = m] = \frac{1}{k}\left(1 - \frac{1}{k}\right)^m. \tag{2}$$

To compute $\Pr[E1|E_m]$, we further subdivide the event into cases about how the m numbers are distributed into the sets S_j, $j \neq i$. Without loss of generality, let

$i = 1$ to keep notations simple. Let $m_2, \ldots, m_k$ denote the case that m_j numbers belong to sequence S_j ($\sum_j m_j = m$). We need to estimate the probability that for sequence S_j, b_j does not conflict with $\mathbb{S}(b_1)$ (recall that we have fixed $i = 1$) during the course that m_j elements arrive in S_j. This can happen only if $\mathbb{S}(b_j)$ (the cache set position of the leading block of S_j right after element $S_{1,t}$) does not lie roughly $\lceil m_i/B \rceil$ blocks from $\mathbb{S}(b_1)$. From Assumption A1 and some careful counting, this is $1-(m_j - 1 + B)/sB$ for $m_j \geqslant 1$. For $m_j = 0$, this probability is 1 since no elements go into S_j and hence there is no conflict.[9] These events are independent from our Assumption A1 and hence these can be multiplied. The probability for a fixed partition $m_2, \ldots, m_k$ is the multinomial $m!/(m_2! \cdots m_k!) \cdot (1/(k-1))^m$ (m is partitioned into $k-1$ parts). Therefore, we can write the following expression for $\Pr[E1|E_m]$.

$$\Pr[E1|E_m] = \sum_{m_2+\cdots+m_k=m} \frac{m!}{m_2! \cdots m_k!} \cdot \left(\frac{1}{k-1}\right)^m \prod_{m_i \neq 0} \left(1 - \frac{m_j - 1 + B}{sB}\right). \quad (3)$$

In the remainder of this section, we obtain an upper bound on the right hand side of Eq (3). Let $nz(m_2, \ldots, m_k)$ denote the number of js for which $m_j \neq 0$ (nonzero partitions). Then, Eq. (3) can be rewritten as the following inequality:

$$\Pr[E1|E_m] \leqslant \sum_{m_2+\cdots+m_k=m} \frac{m!}{m_2! \cdots m_k!} \cdot \left(\frac{1}{k-1}\right)^m \left(1 - \frac{1}{s}\right)^{nz(m_2 \ldots m_k)}, \quad (4)$$

since $(1 - (m_j - 1 + B)/sB) \leqslant (1 - (1/s))$ for $m_j \geqslant 1$. In other words, the right side is the expected value of $(1-(1/s))^{NZ(m,k-1)}$, where $NZ(m, k-1)$ denotes the number of nonempty bins when m balls are thrown into $k-1$ bins. Using Eq. (2) and the preceding discussion, we can write down an upper bound for the (unconditional) probability of $E1$ as

$$\sum_{m=0}^{\infty} \frac{1}{k}\left(1 - \frac{1}{k}\right)^m \cdot E\left[\left(1 - \frac{1}{s}\right)^{NZ(m,k-1)}\right]. \quad (5)$$

We use known sharp concentration bounds for the occupancy problem to obtain the following approximation for the expression (5) in terms of s and k.

THEOREM A.1 ([KAMATH ET AL. 1994]). *Let $r = m/n$, and Y be the number of empty bins when m balls are thrown randomly into n bins. Then*

$$E[Y] = n\left(1 - \frac{1}{m}\right)^m \sim n\exp(-r)$$

and for $\lambda > 0$

$$\Pr[|Y - E[Y]| \geqslant \lambda] \leqslant 2\exp\left(-\frac{\lambda^2(n-1)/2}{n^2 - \mu^2}\right). \quad \square$$

[9] The reader will soon realize that this case leads to some nontrivial calculations.

COROLLARY A.2. *Let NZ be the number of nonempty bins when m balls are thrown into k bins. Then*

$$E[NZ] = k(1 - \exp\left(-\frac{m}{k}\right)$$

and

$$\Pr[|NZ - E[NZ]| \geqslant \alpha\sqrt{2k \log k}] \leqslant \frac{1}{k^{\alpha}}.$$ □

So in Eq. (5), $E[(1-(1/s))^{NZ(m,k-1)}]$ can be bounded by

$$\left(\frac{1}{k^{\alpha}}\right)\left(1-\frac{1}{s}\right) + \left(1-\frac{1}{s}\right)^{k(1-\exp(-m/k)-\alpha\sqrt{2k\log k}/k)} \quad (6)$$

for any α and $m \geqslant 1$.

PROOF (OF LEMMA 4.1). We will split up the summation of (5) into two parts, namely, $m \leqslant e/2 \cdot k$ and $m > e/2 \cdot k$. One can obtain better approximations by refining the partitions, but our objective here is to demonstrate the existence of ϵ and δ and not necessarily obtain the best values.

$$\begin{aligned} &\sum_{m=0}^{\infty} \frac{1}{k}\left(1-\frac{1}{k}\right)^{m} \cdot E\left[\left(1-\frac{1}{s}\right)^{NZ(m,k-1)}\right] \\ &\quad = \sum_{m=0}^{ek/2} \frac{1}{k}\left(1-\frac{1}{k}\right)^{m} \cdot E\left[\left(1-\frac{1}{s}\right)^{NZ(m,k-1)}\right] \\ &\quad + \sum_{m=ek/2+1}^{\infty} \frac{1}{k}\left(1-\frac{1}{k}\right)^{m} \cdot E\left[\left(1-\frac{1}{s}\right)^{NZ(m,k-1)}\right] \end{aligned} \quad (7)$$

The first term can be upper bounded by

$$\sum_{m=0}^{ek/2} \frac{1}{k}\left(1-\frac{1}{k}\right)^{m}$$

which is $\sim 1 - \frac{1}{\exp(e/2)} \sim 0.74$.

The second term can be bounded using Eq. (6) using $\alpha \geqslant 2$.

$$\begin{aligned} &\sum_{1+ek/2}^{\infty} \frac{1}{k}\left(1-\frac{1}{k}\right)^{m} \cdot E\left[\left(1-\frac{1}{s}\right)^{NZ(m,k-1)}\right] \\ &\quad \leqslant \sum_{1+ek/2}^{\infty} \frac{1}{k}\left(1-\frac{1}{k}\right)^{m} \cdot \frac{1}{k^2}\left(1-\frac{1}{s}\right) \\ &\quad + \sum_{1+ek/2}^{\infty} \frac{1}{k}\left(1-\frac{1}{k}\right)^{m} \cdot \left(1-\frac{1}{s}\right)^{k(1-\exp(-m/k)-\alpha\sqrt{2k\log k}/k)} \end{aligned} \quad (8)$$

The first term of the previous equation is less than $1/k$ and the second term can be bounded by

$$\sum_{1+ek/2}^{\infty} \frac{1}{k}\left(1-\frac{1}{k}\right)^{m} \cdot \left(1-\frac{1}{s}\right)^{0.25k}$$

for sufficiently large k ($k > 80$ suffices). This can be bounded by $\sim 0.25 \exp(-0.25k/s)$, so Eq. (8) can be bounded by $1/k + 0.25 \exp(0.25k/s)$. Adding this to the first term of Eq. (7), we obtain an upper bound of $0.75 + 0.25 \exp(-0.25k/s)$ for $k > 100$. Subtracting this from 1 gives us $(1 - \exp(-0.25k/s))/4$, that is, $\delta \geqslant (1 - \exp(-0.25k/s))/4$. □

ACKNOWLEDGMENTS. We are grateful to Alvin Lebeck for valuable discussions related to present and future trends of different aspects of memory hierarchy design. We would like to acknowledge Rakesh Barve for discussions related to sorting, FFTW, and BRP. The first author would also like to thank Jeff Vitter for his comments on an earlier draft of this article. The second author would like to thank Erin Parker for generating the experimental results in Section 4.2.

REFERENCES

AGARWAL, A., HOROWITZ, M., AND HENNESSY, J. 1989. An analytical cache model. *ACM Trans. Comput. Syst. 7*, 2 (May), 184–215.

AGGARWAL, A., ALPERN, B., CHANDRA, A., AND SNIR, M. 1987a. A model for hierarchical memory. In *Proceedings of ACM Symposium on Theory of Computing*. ACM, New York, 305–314.

AGGARWAL, A., CHANDRA, A., AND SNIR, M. 1987b. Hierarchical memory with block transfer. In *Proceedings of IEEE Foundations of Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif., 204–216.

AGGARWAL, A., AND VITTER, J. 1988. The input/output complexity of sorting and related problems. *Commun. ACM 31*, 9, 1116–1127.

ALPERN, B., CARTER, L., FEIG, E., AND SELKER, T. 1994. The uniform memory hierarchy model of computation. *Algorithmica 12*, 2, 72–109.

BARVE, R., GROVE, E., AND VITTER, J. 1997. Simple randomized mergesort on parallel disks. *Parallel Computing 23*, 4, 109–118. (A preliminary version appeared in *SPAA 96*.)

BILARDI, G., AND PESERICO, E. 2001. A characterization of temporal locality and its portability across memory hierarchies. In *Proceedings of ICALP 2001*. Lecture Notes in Computer Science, vol. 2076. Springer-Verlag, New York, 128–139.

CARTER, L., AND GATLIN, K. 1998. Towards an optimal bit-reversal permutation program. In *Proceeding of IEEE Foundations of Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif.

CHATTERJEE, S., JAIN, V. V., LEBECK, A. R., MUNDHRA, S., AND THOTTETHODI, M. 1999a. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the 1999 ACM International Conference on Supercomputing* (Rhodes, Greece). ACM, New York, 444–453.

CHATTERJEE, S., LEBECK, A. R., PATNALA, P. K., AND THOTTETHODI, M. 1999b. Recursive array layouts and fast parallel matrix multiplication. In *Proceedings of 11th Annual ACM Symposium on Parallel Algorithms and Architectures* (Saint-Malo, France). ACM, New York, 222–231.

CHATTERJEE, S., AND SEN, S. 2000. Cache-efficient matrix transposition. In *Proceedings of HPCA-6* (Toulouse, France). 195–205.

CHIANG, Y., GOODRICH, M., GROVE, E., TAMASSIA, R., VENGROFF, D., AND VITTER, J. 1995. External memory graph algorithms. In *Proceedings of the ACM-SIAM Symposium of Discrete Algorithms*. ACM, New York, 139–149.

CORMEN, T. H., SUNDQUIST, T., AND WISNIEWSKI, L. F. 1999. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. *SIAM J. Comput. 28*, 1, 105–136.

FLOYD, R. 1972. Permuting information in idealized two-level storage. In *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher, Eds. Plenum Press, New York, N.Y., 105–109.

FRICKER, C., TEMAM, O., AND JALBY, W. 1995. Influence of cross-interference on blocked loops: A case study with matrix-vector multiply. *ACM Trans. Program. Lang. Syst. 17*, 4 (July), 561–575.

FRIGO, M., AND JOHNSON, S. G. 1998. FFTW: An adaptive software architecture for the FFT. In *Proceedings of ICASSP'98*, vol. 3 (Seattle, Wash.). IEEE Computer Society Press, Los Alamitos, Calif., 1381.

FRIGO, M., LEISERSON, C. E., PROKOP, H., AND RAMACHANDRAN, S. 1999. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS '99)* (New York, N.Y). IEEE Computer Society Press, Los Alamitos, Calif.

GOODRICH, M., TSAY, J., VENGROFF, D., AND VITTER, J. 1993. External memory computational geometry. In *Proceeding of IEEE Foundations of Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif., 714–723.

HILL, M. D., AND SMITH, A. J. 1989. Evaluating associativity in CPU caches. *IEEE Trans. Comput. C-38*, 12 (Dec.), 1612–1630.

HONG, J., AND KUNG, H. 1981. I/O complexity: The red blue pebble game. In *Proceedings of ACM Symposium on Theory of Computing*. ACM, New York.

KAMATH, A., MOTWANI, R., PALEM, K., AND SPIRAKIS, P. 1994. Tail bounds for occupancy and the satisfiability threshold conjecture. In *Proceeding of IEEE Foundations of Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif., 592–603.

LADNER, R., FIX, J., AND LAMARCA, A. 1999. Cache performance analysis of algorithms. In *Proceedings of the ACM-SIAM Symposium of Discrete Algorithms*. ACM, New York.

LAM, M. S., ROTHBERG, E. E., AND WOLF, M. E. 1991. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*. 63–74.

LAMARCA, A., AND LADNER, R. 1997. The influence of cache on the performance of sorting. In *Proceedings of the ACM-SIAM Symposium of Discrete Algorithms*. ACM, New York, 370–379.

LEBECK, A. R., AND WOOD, D. A. 1994. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer 27*, 10 (Oct.), 15–26.

MEHLHORN, K., AND SANDERS, P. 2000. Scanning multiple sequences via cache memory. citeseen.nj.nec.com/506957.html.

PRZYBYLSKI, S. A. 1990. *Cache and Memory Hierarchy Design: A Performance-Directed Approach*. Morgan-Kaufmann, San Mateo, Calif.

SANDERS, P. 1999. Accessing multiple sequences through set associative caches. In *Proceedings of ICALP*. Lecture Notes in Computer Science, vol. 1644. Springer-Verlag, New York, 655–664.

SAVAGE, J. 1995. Extending the Hong-Kung model to memory hierarchies. In *Proceedings of COCOON*. Lecture Notes in Computer Science, vol. 959. Springer-Verlag, New York, 270–281.

SLEATOR, D., AND TARJAN, R. 1985. Amortized efficiency of list update and paging rules. *Commun. ACM 28*, 2, 202–208.

THOTTETHODI, M., CHATTERJEE, S., AND LEBECK, A. R. 1998. Tuning Strassen's matrix multiplication for memory efficiency. In *Proceedings of SC98 (CD-ROM)* (Orlando, Fla.) (Available from http://www.supercomp.org/sc98).

VITTER, J., AND NODINE, M. 1993. Large scale sorting in uniform memory hierarchies. *J. Paral. Dist. Comput. 17*, 107–114.

VITTER, J., AND SHRIVER, E. 1994. Algorithms for parallel memory. I: Two-level memories. *Algorithmica 12*, 2, 110–147.

RECEIVED OCTOBER 2000; REVISED OCTOBER 2002; ACCEPTED OCTOBER 2002