Towards a Theory of Cache-Efficient Algorithms (Extended Abstract)*

Sandeep Sen† Siddhartha Chatterjee‡

Submitted for publication

Abstract

We describe a model that enables us to analyze the running time of an algorithm in a computer with a memory hierarchy with limited associativity, in terms of various cache parameters. Our 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-optimal algorithms for some fundamental problems like sorting, FFT, and an important subclass of permutations in the single-level cache model. We also show that ignoring associativity concerns could lead to inferior performance, by analyzing the average-case cache behavior of mergesort. Our techniques may be used for systematic exploitation of the memory hierarchy starting from the algorithm design stage, and dealing with the hitherto unresolved problem of limited associativity.

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. While 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 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 [5]. 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 paper, we address the issue of better and systematic utilization of caches starting from the algorithm design stage.

*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.

†Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175, USA and Department of Computer Science and Engineering, IIT Delhi, New Delhi 1100116, India. E-mail:ssen@cs.unc.edu and ssen@cse.iitd.ernet.in.

‡Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175, USA. E-mail: sc@cs.unc.edu.

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. Memory hierarchy models used by computer architects to design caches have numerous parameters and suffer from the first shortcoming [1, 24]. Existing memory hierarchy models [2–5] 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.[1] They introduce *conflict misses* (see Appendix A for more details) that make analysis of algorithms much more difficult [14]. Carter and Gatlin [8] 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 paper, 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 [4] 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. The work in this paper 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.

The remainder of this paper 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 provably inferior, even in the average case, in the cache model. Section 5 describes a natural extension of our model to multiple levels of caches, although the bounds we prove are not as tight as in the two-level model. Finally, Section 6 presents conclusions, possible refinements to the model, and directions for future work.

2 Related work

The I/O model 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/Os. Floyd [13] defined a formal model and proved tight bounds on the number of I/Os required to transpose a matrix using two internal memory pages. Hong and Kung [19] extended this model and studied the I/O complexity of FFT when the internal memory size is bounded by M. Aggarwal and Vitter [4] 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 [11] and computational geometry [17].

Researchers have also modeled multiple levels of memory hierarchy. Aggarwal *et al.* [2] 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 multi-level hierarchy. Aggarwal *et al.* [3] added the capability of block transfer to the HMM, which enabled them to obtain faster algorithms. Alpern *et al.* [5] described the *Uniform Memory Hierarchy* (UMH) model, where the access costs differ in discrete steps. Very recently, Frigo *et al.* [16] presented an alternate strategy of

[1] See the discussion in [8] on a simple matrix transpose program.

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.[2] Other attempts were directed towards extracting better performance by parallel memory hierarchies [12, 28, 29], where several blocks could be transferred simultaneously.

Ladner *et al.* [21] 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 [23].

3 The cache model

The (two-level) I/O model of Aggarwal and Vitter [4] 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*. The use of the model is meaningful when the problem size $N \gg M$.

The I/O model contains the following further assumptions.

1. A datum can be used in a computation iff it is present in fast memory. All data initially resides in slow memory. Data can be transferred between slow and fast memory (in either direction) by I/O operations.

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 designed for such restricted memory access patterns. We denote the map from a memory address to its block address by $\mathbb{B}$. The internal memory can hold at least three blocks, i.e., $M \geqslant 3 \cdot B$.

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.

4. A block of data from slow memory can be placed in any block of fast memory.

5. I/O operations are explicit in the algorithm.

The goal of algorithm design in this model is to minimize 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 this case, the cache is the fast memory, while main memory is 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.

- 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 will use L to denoted the *normalized* cache latency. This cost function assigns a cost of 1 for accessing an element in cache and L for accessing an element in the main memory. This way, we also account for the computation in cache.

[2] However, none of these models address the problem of limited associativity in cache.

- Main memory blocks are mapped into cache sets using a *fixed* and pre-determined mapping function that is implemented in hardware (see Appendix A for details). Typically, this is a modulo mapping based on the low-order address bits. However, the results of this section will hold as long as there is a *fixed* address mapping function that distributes the main memory evenly in the cache. We denote this mapping from main memory blocks to cache sets by $\mathbb{S}$. We will occasionally slightly abuse this notation and apply $\mathbb{S}$ directly to a memory address x rather than to $\mathbb{B}(x)$.

- The cache is not visible to the programmer (not even at the assembly level). When a program issues a reference to a memory location x, an *image* (copy) of the main memory block $b = \mathbb{B}(x)$ 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 another block b' that is mapped to the same cache set (*i.e.*, $\mathbb{S}(b) = \mathbb{S}(b')$). In other words, a cache set c contains the latest memory block referenced that is mapped to this set.

To summarize, we use the notation $\mathfrak{C}(M, B, L)$ to denote our three-parameter cache model, and the notation $\mathfrak{I}(M, B)$ to denote the I/O model with parameters M and B. We will use n and m to denote N/B and M/B respectively. The assumptions of our cache model parallel those of the I/O model, except as noted above.[3] The goal of algorithm design in the cache model is to minimize *running time*, defined as the number of cache accesses plus L times the number of main memory accesses.

3.1 Emulating I/O algorithms

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. Our main result in this section establishes a connection between the I/O model and the cache model using a very simple emulation scheme.

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

Proof: Note that I is usually not accounted for in the I/O model, but we will keep track of the internal memory computation done in A in our emulation. The idea behind the emulation is as follows. We will mimic the behavior of the I/O algorithm A in the cache model, using an array *Buf* of m blocks to play the role of the fast memory. We will view the main memory in the cache model as an array *Mem* of B-element blocks. Although *Buf* is also part of the memory, we are using different notations to make their roles explicit in this proof. Likewise, we will view the cache as an array of sets and denote the ith set by $C[i]$.

As discussed above, we do not have explicit control on the contents of the cache locations. However, we can control the memory access pattern through a level of indirection so as to maintain a 1-1 correspondence between *Buf* and the cache. Wlog, we assume that $\mathbb{S}$ maps block i of *Buf* to cache set $C[i]$ for $i \in [1, m]$.

We divide the I/O algorithm into rounds, where in each round, the I/O algorithm A transfers a block between the slow memory and the fast memory and (possibly) does some computations. The cache algorithm A^c transfers the same blocks between *Mem* and *Buf* and then does the identical computations in *Buf*. Figure 1 formally describes the procedure. Note that the B elements must be explicitly copied in the cache model.

It must be obvious that the final outcome of algorithm A^c is the same as algorithm A. A more interesting issue is the cost of the emulation.

A block of size B is transferred into cache if its image does not exist in the cache at the time of reference. The invariant that we try to maintain at the end of each round is that there is a 1-1 correspondence between *Buf* and C. This will ensure that all the I operations are done within the cache at minimal cost.

[3] Frigo *et al.* [16] independently arrive at a very similar parameterization of their model.

Round t of the emulation

I/O Algorithm A	*Cache Emulation A^c*
1. Transfer block b_t from slow memory to block a_t of the fast memory 2. Perform computations in fast memory	1. Copy contents of the B locations of *Mem*$[b_t]$ into *Buf*$[a_t]$ 2. Perform identical computations in *Buf*

Figure 1: The emulation scheme used in the proof of Theorem 3.1.

Assume that we have maintained the above invariant at the end of round $t-1$. In round t, we transfer block *Mem*$[b_t]$ into *Buf*$[a_t]$. Accessing the memory block *Mem*$[b_t]$ will displace the existing block in cache set $C[q]$, where $q = \mathbb{S}(b_t)$. From the invariant, we know that the block displaced from $C[q]$ is *Buf*$[q]$, which must be restored to cache to restore the invariant. We can bring it back by a single memory reference and charge this to the round t itself, which is L. (Actually it will be brought back during the subsequent reference, so the previous step is only to simplify the accounting.)

The cost of copying *Mem*$[b_t]$ to *Buf*$[a_t]$ is $L+B$ assuming that *Mem*$[b_t]$ and *Buf*$[a_t]$ are not mapped to the same cache set ($\mathbb{S}(b_t) \neq \mathbb{S}(a_t)$). Otherwise it will cause alternate cache misses (*thrashing*) of the blocks *Mem*$[b_t]$ and *Buf*$[a_t]$ leading to $L \cdot B$ steps for copying. This can be prevented by transferring through an intermediate memory block *Mem*$[Y]$ such that $\mathbb{S}(Y) \neq \mathbb{S}(b_t)$. Having two such intermediate buffers that map to distinct cache sets would suffice in all cases. So, we first transfer *Mem*$[b_t]$ to *Mem*$[Y]$ followed by *Mem*$[Y]$ to *Buf*$[j]$. The first copying has cost $2L+B$ since both blocks must be fetched from main memory. The second transfer is between blocks, one of which is present in the cache, so it has cost $L+B$. To this we must also add cost L for restoring the block of *Buf* that was mapped to the same cache set as *Mem*$[Y]$. So, the total cost of the *safe* method is $4L+2B$.

The internal processing remains identical. If I_t denotes the internal processing cost of step t, the total cost of the emulation is at most $\sum_{t=1}^{T}(I_t + 2(L+B) + 2L) = I + 4L \cdot T + 2B \cdot T$. □

Remark 1

- A possible alternative to using intermediate memory-resident buffers to avoid thrashing is to use registers, since register access is much faster. In particular, if we have B registers, then we can save two extra memory accesses, bringing down the emulation cost to $2L+2B$.
- We can make the emulation somewhat simpler by using a randomized mapping scheme. That is, if we choose the starting location of array *Buf* randomly, then the probability that *Mem*$[b_t]$ and *Buf*$[a_t]$ have the same image is $1/M$. So the expected emulation cost is $I + 2L \cdot T + (B + (LB)/M) \cdot T$ without using any intermediate copying.
- 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 [22] and bit-reversal permutation [8]. 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 will call such I/O algorithms *block-efficient*.

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

Remark 2 The algorithms for sorting, FFT, matrix transposition, and matrix multiplication described in Aggarwal and Vitter [4] are block-efficient.

3.2 Extension to set-associative cache

Some of the modern architectures (such as the SGI machines) allow limited flexibility in the address mapping between memory blocks and cache sets. The k-way set-associative cache has the property that a memory block can reside in any (one) of k cache-sets. Thus, $k = 1$ corresponds to the direct-mapped cache we have considered so far, while $k = m$ corresponds to a fully associative cache. Values of k for data caches are generally small, usually in the range 1–4.

If all the k sets are occupied, a replacement policy like LRU is used (by the hardware) to find an assignment for the referenced block. The emulation technique of the previous section would extend to this scenario easily if we had explicit control on the replacement. This not being the case, we shall tackle it indirectly by making use of an useful property of LRU that Frigo *et al.* [16] exploited in the context of designing cache-oblivious algorithms for a fully associative cache.

Lemma 3.1 (Sleator-Tarjan[26]) *For any sequence s, F_{LRU}, the number of misses incurred by LRU with cache size n_{LRU} is no more than $(n_{LRU}/(n_{LRU} - n_{OPT} + 1)F_{OPT})$, where F_{OPT} is the minimimum 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, *i.e.*, we choose the buffer to be of size $m/2$, such that for every k cache lines in a set, we have only $k/2$ buffer blocks. From Lemma 3.1, 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.

Theorem 3.3 (Generalized Emulation Theorem) *An algorithm A in $\mathfrak{I}(M/2, B)$ using T block transfers and I processing time can be converted to an equivalent algorithm A^c in the k-way set-associative cache model with parameters M, B, L that runs in $O(I + (L + B) \cdot T)$ steps. The memory requirement of A^c is an additional $m/2 + 2$ blocks beyond that of A.*

3.3 The cache complexity of sorting and other problems

Aggarwal and Vitter [4] prove the following lower bound for sorting and FFT in the I/O model.

Lemma 3.2 ([4]) *The average-case and the worst-case number of I/O's required for sorting N records and for computing the N-input FFT graph in $\mathfrak{I}(M, B)$ is $\Omega\left(\frac{N}{B}\frac{\log(1+N/B)}{\log(1+M/B)}\right)$.*

Theorem 3.4 *The lower bound for sorting in $\mathfrak{C}(M, B, L)$ is $\Omega(N \log N + L\frac{N}{B}\frac{\log N/B}{\log M/B})$.*

Proof: Any lower bound in the number of block transfers in $\mathfrak{I}(M, B)$ carries over to $\mathfrak{C}(M, B, L)$. Since the lower bound is the maximum of the lower bound on number of comparisons and the bound in Lemma 3.2, the theorem follows by dividing the sum of the two terms by 2. □

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

Proof: The M/B-way mergesort algorithm described in Aggarwal and Vitter [4] has an I/O complexity of $O(\frac{N}{B}\frac{\log N/B}{\log M/B})$. The processing time involves maintaining a heap of size M/B and $O(\log M/B)$ 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 2, the cost of this algorithm in the cache model is $O(N \log N + L \cdot \frac{N}{B} \cdot \frac{\log N/B}{\log M/B})$. Optimality follows from Theorem 3.4. □

Remark 3 The M/B-way distribution sort (multiway quicksort) also has the same upper bound.

We can prove a similar result for FFT computations.

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

Remark 4 The FFTW algorithm [15] is optimal only for $B = 1$. Barve [6] 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. Combining the work of Cormen *et al.* [12] with our emulation scheme, we obtain the following result.

Theorem 3.7 *The class of BMMC permutations for N elements can be achieved in $\Theta\left(N + L \cdot \frac{N}{B} \frac{\log M}{\log(M/B)}\right)$ steps in $\mathfrak{C}(M, B, L)$.*

Remark 5 Many known geometric [11] and graph algorithms [17] in the I/O model, such as convex hull and graph connectivity, can be transformed optimally into the cache model. The final paper will contain a more comprehensive list of such applications.

4 Average-case performance of mergesort in the cache model

In this section, we analyze the average-case performance of k-way mergesort in the cache model. Of the three architectural classes of misses described in Appendix A, 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. It is easy to construct a worst-case input permutation where there will be a conflict miss for every input element (a cyclic distribution suffices), so the average case is more interesting.

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 j-th 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, *i.e.*, $S_{i,j}$ 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 will 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 6 A recent variation of the mergesort algorithm (see [7]) actually satisfies **A1** by its very nature. So, the present analysis is directly applicable to its average-case performance in 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 will compute an upper bound on $\Pr[E1]$, under the 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 B. □

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 7 Intuitively, by choosing k to be orders of magnitude smaller than s (say $k = \sqrt{s}$), we can minimize the probability of conflict misses. However, the number of merge phases increases, thereby increasing the running time. This underlines the critical role of conflict misses which forces us to use a very small fraction of the available cache, even when capacity is not a bottleneck. We need to achieve a good balance between the two factors—conflict misses and the number of merge phases for achieving the best running times.

Remark 8 Using a relatively small fraction of available cache is considered to be a good heuristic to minimize the expected number of conflict misses. On the other hand, use of the emulation theorem guarantees minimal conflict misses while making better use of cache.

5 Multi-level cache models

Many modern architectures, such as the UltraSparc and the Alpha. have a memory hierarchy consisting of multiple cache levels. Consider two cache levels L_1 and L_2 preceding main memory, with 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 L_1. If it is not present in L_1, then it is searched for in L_2 (these can be overlapped with appropriate hardware support). If the item is not present in L_1 but it is in L_2, then it is brought into L_1. In case that it is not in L_2, then a cache line is brought into L_2 and into L_1. The size of cache line brought into L_2 (denoted by B_2) is usually larger than the one brought into L_1 (denoted by B_1). The expectation is that the more frequently used items will remain in the faster cache.

We now extend our previous model to accommodate a multi-level hierarchy of caches. We denote the cache levels by $L_1, \ldots, L_k$ for some fixed constant k and the corresponding cache capacities by $M_1, \ldots, M_k$. We assume that $M_1 \leqslant \cdots \leqslant M_k$. We represent the cost of transferring a block from L_i by ℓ_i and the block sizes by $B_1, \ldots, B_k$. Further, we assume that $B_1 \leqslant \cdots \leqslant B_k$.

We first consider a restriction of the model described above where data cannot be transferred simultaneously across non-consecutive cache levels.

Theorem 5.1 *The lower bound for sorting in the restricted multi-level cache model is* $\Omega(N \log N + \sum_{i=1}^{k} \ell_i \cdot \frac{N}{B_i} \frac{\log N/B_i}{\log M_i/B_i})$.

Proof: The proof of Aggarwal and Vitter 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. □

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

Lemma 5.1 *The lower bound for sorting in the multi-level cache model is*

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

□

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 multi-level 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, $\frac{N}{B_k} \frac{\log N/B_k}{\log M_k/B_k}$. For level $k-1$, we will subtract this number from the lower bound of $\frac{N}{B_{k-1}} \frac{\log N/B_{k-1}}{\log M_{k-1}/B_{k-1}}$. Continuing in this fashion, we obtain the following lower bound.

Theorem 5.2 *The lower bound for sorting in the multi-level cache model is*

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

□

If we further assume that $\frac{M_i}{M_{i-1}} \geqslant \frac{B_i}{B_{i-1}} \geqslant 3$, we obtain a relatively simple expression that resembles[4] Theorem 5.1.

[4]This is because the consecutive terms in the expression in the second summation of the previous lemma decrease by a factor of 3.

Corollary 5.3 *The lower bound for sorting in the multi-level cache model with geometrically decreasing cache sizes and cache lines is* $\Omega(N \log N + \frac{1}{2}\sum_{i=1}^{k} \ell_i \cdot \frac{N \cdot \log N/B_i}{B_i \log M_i/B_i})$. □

These lower bounds are in the same spirit as those of Nodine and Vitter [28] and Savage [25], that is, the lower bounds do not capture the simultaneous interaction of the different levels.

Theorem 5.4 *In a multi-level cache, where the* B_i *blocks are composed of* B_{i-1} *blocks, we can sort in* ***expected*** *time* $O\left(N \log N + \left(\frac{\log N/B_1}{\log M_1/B_1}\right) \cdot \sum_{i=1}^{k} \ell_i \cdot \frac{N}{B_i}\right)$.

Proof: We perform a M_1/B_1-way mergesort using the variation proposed by Barve *et al.* [7] 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.* [7], 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 $O\left(N/B_1(\frac{\log N/B_1}{\log M_1/B_1})\right)$ B_1 block transfers suffice for $(M_1/B_1)^{1/3}$-way mergesort. □

Remark 9 This bound is reasonably close to Corollary 5.3 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 *sub-block* property. This is satisfied by the mergesort (and quicksort) and a number of other algorithms but cannot be assumed in general.

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 three additional salient features of cache: lower miss penalty, limited associativity, 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 interference misses. We have also demonstrated 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 direct-mapped cache that does not distinguish between reads and writes. Modeling a non-blocking 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* or 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.

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. 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 in the beginning, these factors may be easier to tackle at the level of implementation than algorithm design.

Some of the cache problems we observe can be traced to the simple array layout schemes used in current programming languages. It has shown elsewhere [9, 10, 27] 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.

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 paper.

References

[1] A. Agarwal, M. Horowitz, and J. Hennessy. An analytical cache model. *ACM Trans. Comput. Syst.*, 7(2):184–215, May 1989.

[2] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of ACM Symposium on Theory of Computing*, pages 305–314, 1987.

[3] A. Aggarwal, A. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proceedings of IEEE Foundations of Computer Science*, pages 204–216, 1987.

[4] A. Aggarwal and J. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(5):1118–1127, 1988.

[5] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2):72–109, 1994.

[6] R. Barve. Private communication.

[7] R. Barve, E. Grove, and J. Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4):109–118, 1997. A preliminary version appeared in SPAA 96.

[8] L. Carter and K. Gatlin. Towards an optimal bit-reversal permutation program. In *Proceeding of IEEE Foundations of Computer Science*, 1998.

[9] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, pages 444–453, Rhodes, Greece, June 1999.

[10] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi. Recursive array layouts and fast parallel matrix multiplication. In *Proceedings of Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 222–231, Saint-Malo, France, June 1999.

[11] Y. Chiang, M. Goodrich, E. Grove, R. Tamassia, D. Vengroff, and J. Vitter. External memory graph algorithms. In *Proceedings of the ACM-SIAM Symposium of Discrete Algorithms*, pages 139–149, 1995.

[12] T. H. Cormen, T. Sundquist, and L. F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. *SIAM Journal of Computing*, 28(1):105–136, 1999.

[13] R. Floyd. Permuting information in idealized two-level storage. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 105–109. Plenum Press, New York, NY, 1972.

[14] C. Fricker, O. Temam, and W. Jalby. Influence of cross-interference on blocked loops: A case study with matrix-vector multiply. *ACM Trans. Prog. Lang. Syst.*, 17(4):561–575, July 1995.

[15] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of ICASSP'98*, volume 3, page 1381, Seattle, WA, 1998. IEEE.

[16] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of IEEE Foundations of Computer Science*, 1999. To appear.

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

[18] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Trans. Comput.*, C-38(12):1612–1630, Dec. 1989.

[19] J. Hong and H. Kung. I/O complexity: The red blue pebble game. In *Proceedings of ACM Symposium on Theory of Computing*, 1981.

[20] A. Kamath, R. Motwani, K. Palem, and P. Spirakis. Tail bounds for occupancy and the satisfiability threshold conjecture. In *Proceeding of IEEE Foundations of Computer Science*, pages 592–603, 1994.

[21] R. Ladner, J. Fix, and A. LaMarca. Cache performance analysis of algorithms. In *Proceedings of the ACM-SIAM Symposium of Discrete Algorithms*, 1999.

[22] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Apr. 1991.

[23] A. LaMarca and R. Ladner. The influence of cache on the performance of sorting. In *Proceedings of the ACM-SIAM Symposium of Discrete Algorithms*, pages 370–379, 1997.

[24] S. A. Przybylski. *Cache and Memory Hierarchy Design: A Performance-Directed Approach*. Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[25] J. Savage. Extending the Hong-Kung model to memory hierarchies. In *Proceedings of COCOON*, volume LNCS 959, pages 270–281. Springer Verlag, 1995.

[26] D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985.

[27] M. Thottethodi, S. Chatterjee, and A. R. Lebeck. Tuning Strassen's matrix multiplication for memory efficiency. In *Proceedings of SC98 (CD-ROM)*, Orlando, FL, Nov. 1998. Available from http://www.supercomp.org/sc98.

[28] J. Vitter and M. Nodine. Large scale sorting in uniform memory hierarchies. *Journal of Parallel and Distributed Computing*, 17:107–114, 1993.

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

A A primer on caches

This section provides a quick review of terms associated with caches that are used in the paper. We assume a simplified memory hierarchy that processes one memory access at a time, with no distinction between memory reads and writes.

The structure of a single level of a memory hierarchy—called a *cache*—is generally characterized by three parameters: **A**ssociativity, **B**lock size, and **C**apacity. 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 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.*, $A = \frac{C}{B}$), the cache is said to be *fully associative*; if $A = 1$, the cache is *direct-mapped*; otherwise, the cache is A*-way set associative*.

For a given memory access, the hardware inspects the cache to determine if the corresponding memory element 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. If the memory block is resident, a cache hit is said to occur, and the cache satisfies the access after its *access latency*. If the memory block is not resident, a cache miss is said to occur.

From an architectural standpoint, cache misses fall into one of three classes [18].

- A *compulsory miss* is one that is caused by referencing a previously unreferenced memory block. Eliminating a compulsory miss requires prefetching the data, either by an explicit prefetch operation or by placing more data items in a single memory block.

- A reference that is not a compulsory miss but misses in a fully-associative cache with LRU replacement is classified as a *capacity miss*. Capacity misses are caused by referencing more memory blocks than can fit in the cache. Restructuring the program to re-use blocks while they are in cache can reduce capacity misses.

- A reference that hits in a fully-associative cache but misses in an A-way set-associative cache is classified as a *conflict miss* (or *interference miss*). A conflict miss to block X indicates that block X has been referenced in the recent past, since it is contained in the fully-associative cache, but at least A other memory blocks that map to the same cache set have been accessed since the last reference to block X. Eliminating conflict misses requires transforming the program to change either the memory allocation and/or layout of the two arrays (so that contemporaneous accesses do not compete for the same sets) or the manner in which the arrays are accessed.

  At the program source level, interference misses can be further subdivided based on whether the interfering blocks come from different parts of a single array, or from different arrays. The miss is called a *self-interference miss* in the former case and a *cross-interference miss* in the latter case [22].

Another important component of the memory hierarchy is the *translation lookaside buffer* (TLB). This is a small highly-associative (often fully associative) cache of address translations that is used in converting virtual memory addresses to physical memory addresses.

B Approximating probability of conflict

Let μ be the number of elements between $S_{i,j}$ and $S_{i,j+1}$, *i.e.*, 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, *i.e.*,

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

To compute $\Pr[E1|E_m]$, we further subdivide the event into cases about how the m numbers are distributed into the sets $S_j, j \neq i$. Wlog, 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 $1 - \frac{m_j - 1 + B}{sB}$ 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.[5] These events are independent from our assumption A1 and hence these can

[5]The reader will soon realize that this case leads to some non-trivial calculations.

be multiplied. The probability for a fixed partition $m_2, \ldots, m_k$ is the multinomial $\frac{m!}{m_2! \cdots m_k!} \cdot \left(\frac{1}{k-1}\right)^m$ (m is partitioned into $k-1$ parts). Therefore we can write the following expression for $\Pr[E1|E_m]$.

$$\Pr[E1|E_m] = \sum_{m_2+\cdots+m_k=m} \frac{m!}{m_2! \cdots m_k!} \cdot \left(\frac{1}{k-1}\right)^m \prod_{m_i \neq 0} \left(1 - \frac{m_j - 1 + B}{sB}\right) \tag{2}$$

In the remainder of this section, we will obtain an upper bound on the right hand side of equation (2). Let $nz(m_2, \ldots, m_k)$ denote the number of js for which $m_j \neq 0$ (non-zero partitions). Then equation (2) can be rewritten as the following inequality.

$$\Pr[E1|E_m] \leqslant \sum_{m_2+\cdots+m_k=m} \frac{m!}{m_2! \cdots m_k!} \cdot \left(\frac{1}{k-1}\right)^m \left(1 - \frac{1}{s}\right)^{nz(m_2 \ldots m_k)} \tag{3}$$

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

$$\sum_{m=0}^{\infty} \frac{1}{k}\left(1 - \frac{1}{k}\right)^m \cdot E\left[\left(1 - \frac{1}{s}\right)^{NZ(m,k-1)}\right] \tag{4}$$

We use known sharp concentration bounds for the occupancy problem to obtain the following approximation for the expression (4) in terms of s and k.

Theorem B.1 ([20]) *Let $r = m/n$, and Y be the number of empty bins when m balls are thrown randomly into n bins. Then*

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

and for $\lambda > 0$

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

□

Corollary B.2 *Let NZ be the number of non-empty bins when m balls are thrown into k bins. Then*

$$E[NZ] = k(1 - e^{-m/k})$$

and

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

□

So in equation (3), $E[(1 - \frac{1}{s})^{NZ(m,k-1)}]$ can be bounded by

$$1/k^{\alpha}\,(1 - 1/s) + \left(1 - \frac{1}{s}\right)^{k(1 - e^{-m/k} - \alpha\sqrt{2k \log k}/k)} \tag{5}$$

for any α and $m \geqslant 1$.

Proof: (of Lemma 4.1): We will split up the summation of (4) 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.

$$\begin{aligned}\sum_{m=0}^{\infty} \frac{1}{k}\left(1-\frac{1}{k}\right)^m \cdot E[\left(1-\frac{1}{s}\right)^{NZ(m,k-1)}] &= \sum_{m=0}^{ek/2k} \frac{1}{k}\left(1-\frac{1}{k}\right)^m \cdot E[\left(1-\frac{1}{s}\right)^{NZ(m,k-1)}] \\ &+ \sum_{m=ek/2+1}^{\infty} \frac{1}{k}\left(1-\frac{1}{k}\right)^m \cdot E[\left(1-\frac{1}{s}\right)^{NZ(m,k-1)}] \quad (6)\end{aligned}$$

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 $\sim 1 - \frac{1}{e^{e/2}} \sim 0.74$.

The second term can be bounded using equation (5) using $\alpha \geqslant 2$.

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

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

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

for sufficiently large k ($k > 80$ suffices). This can be bounded by $\sim 0.25e^{-0.25k/s}$, so equation (7) can be bounded by $1/k + 0.25e^{0.25k/s}$. Adding this to the first term of equation (6), we obtain an upper bound of $0.75 + 0.25e^{-0.25k/s}$ for $k > 100$. Subtracting this from 1 gives us $\frac{1-e^{-0.25k/s}}{4}$, *i.e.*, $\delta \geqslant \frac{1-e^{-0.25k/s}}{4}$. $\square$