5 Caching

Processors of modern computers are faster than their memory by several orders of magnitude. To avoid waiting for data most of the time, they use caching. They employ a small, but very fast memory called the *cache,* which stores a copy of frequently used data. Some machines have multiple levels of caches, each slightly larger and slightly slower than the previous one. We can also work with data stored on a slow medium like a hard disk and cache it in our main memory. Even further: data stored on other machines in the network can be cached on a local disk.

In short, caching is ubiquitious and algorithms whose access to data can be cached well are much faster than their cache-unfriendly counterparts. In this chapter, we are going to develop several techniques for designing cache-efficient algorithms and data structures. However, we must start with re-definining our model of computation.

5.1 Models of computation

External memory model

The first model we will consider is the *external memory model,* also known as the *I/O model.* It was originally designed for study of algorithms working with data on disks, which do not fit in the machine's main memory.

The model possesses two types of memory:

- *External memory* of potentially infinite size, organized in blocks of size B — since we are going to study asymptotics, units do not matter, so we will measure all sizes in arbitrary *items.*
- *Internal memory* for M items, also organized in B-item blocks.

All computations are performed in the internal memory, similarly to the Random Access Machine. External memory can be accessed only indirectly: a block of data has to be loaded to the internal memory first and if it is modified, it has to be written back to the external memory later. We can assume that the machine has special instructions for that.

In addition to time and space complexity, we are going to study the *I/O complexity* of algorithms: the maximum number of input/output operations (reads and writes of blocks) performed by the algorithm for a given size of input.

We will often reduce calculation of I/O complexity to just counting the number of block reads. In most cases, it is easy to verify that the number of writes is bounded by the

number of reads, at least asymptotically. When intermediate results are written, they are read later. Unless the final output is asymptotically larger than the input, the number of writes needed to create the output is bounded by the number of reads needed to obtain the input.

Cache models

Machines, which accelerate access to main memory by using a cache, can be modelled in a similar way. The role of external memory is played by the machine's main memory. The internal memory is the cache. (Or perhaps we are caching data on a disk: then the external memory is the disk and internal memory is the main memory.)

There is one crucial difference, though. Unlike in the I/O model, block transfers between main memory and the cache are not controlled by the program itself, but by the machine. The program accesses data in the main memory and a part of the machine (which will be called the *cache controller*) handles transfers of the particular blocks between the main memory and the cache automatically.

We will assume that the cache controller works optimally — that is, it decides which blocks should be kept in the cache and which should be evicted in a way minimizing the total number of I/O operations. This assumption is obviously unrealistic: an optimal cache controller needs to know the future accesses. However, we will prove in section 5.4 that there exists a deterministic strategy which approximates this controller within a constant factor.

There are two varieties of this cache model: the *cache-aware* model, in which the algorithm knows the parameters of the cache (the block size B and the cache size M), and the *cache-oblivious* model, where it doesn't.

It will be easier to design algorithms for the cache-aware model, but cache-oblivious algorithms are more useful. One such algorithm can work efficiently on many machines with different cache architectures. And not only that: if the machine uses a hierarchy of several caches, the algorithm can be shown to be optimal (within a constant factor) with respect to all caches simultaneously.

5.2 Basic algorithms

Scanning an array

Let us consider a simple example first. We have an array of N items, which we read sequentially from the start to the end. This is called an *array scan.*

In the I/O model, we can make the array start at a block boundary. So we need to read $\lceil N/B \rceil \leq N/B + 1$ consecutive blocks to scan all items. All blocks can be stored at the same place in the internal memory. This strategy is obviously optimal.

A cache-aware algorithm can use the same sequence of reads. Generally, we do not know the sequence of reads used by the optimal caching strategy, but any specific sequence can serve as an upper bound. For example, the sequence we used in the I/O model.

A cache-oblivious algorithm cannot guarantee that the array will be aligned on a block boundary (it does not know B). In the worst case, this can cost us an extra read — imagine an array of size 2 spanning two blocks.

Asymptotically speaking, the I/O complexity of an array scan is $\mathcal{O}(N/B + 1)$ in all three models. Please note that the additive 1 cannot be „hidden inside $\mathcal{O}$“, because for every $\varepsilon > 0$, we have $N/B \leq \varepsilon$ for infinitely many pairs (N, B). Also, in caching models we cannot replace the $\mathcal{O}$ by Θ since some of the blocks could be already present in the cache.

Mergesort

The traditional choice of algorithm for sorting in external memory is Mergesort. Let us analyze it in our models. Merging two sorted arrays involves three sequential scans: one for each source array, one for the destination array. The scans are interleaved, but so we can interleave their I/O operations. We need 1 block of internal memory for each scan, but let us generally assume that $M/B \geq c$ for any fixed constant c. Therefore the merge runs in linear time and it transfers $\mathcal{O}(T/B + 1)$ blocks, where T is the total number of items merged.

Our Mergesort will proceed bottom-up. Items will be stored in a single array as a sequence of same-size *runs* (sorted ranges), except for the last run which can be shorter. We start with 1-item runs. In each pass, we double the size of runs by merging pairs of consecutive runs. After the i-th pass, we have 2^i-item runs, so we stop after $\lceil \log N \rceil$ passes.

Each pass runs in $\Theta(N)$ time, so the whole algorithm has time complexity $\Theta(N \log N)$. This is optimal for comparison-based sorting.

Let us analyze I/O complexity. All merges in a single pass actually form three scans of the whole array, so they perform $\mathcal{O}(N/B + 1)$ block transfers. All passes together transfer $\mathcal{O}(N/B \cdot \log N + \log N)$ blocks. The extra $\log N$ is actually exaggerated: the $+1$ in complexity of a single pass is asymptotically significant only if $N < B$, that is if the whole input is smaller than a block. In that case, all passes happily compute on the same cached block. We can therefore improve the bound to $\mathcal{O}(N/B \cdot \log N + 1)$ in all three models.

Multi-way Mergesort

Could we take advantage of a cache larger than 3 blocks? Yes, with a multi-way Mergesort. A K-way Mergesort combines K runs at once, so the number of passes decreases to $\lceil \log_K N \rceil = \lceil \log N / \log K \rceil$. A K-way merge needs to locate a minimum of K items in every step, which can be done using a heap. Every step therefore takes time $\Theta(\log K)$, so merging T items takes $\Theta(T \log K)$ and the whole Mergesort $\Theta(N \log K \cdot \log N / \log K) = \Theta(N \log N)$ for any K. (For $K = N$, we actually get Heapsort.)

If we have large enough cache during the merge, every input array has its own scan and the heap fits in cache. Then the total I/O complexity is $\mathcal{O}(T/B + K)$. The extra K is significant in the situation when all runs are small and each is located in a different block. This actually does not happen in Mergesort: all the runs are always consecutive in memory. Therefore $\mathcal{O}(T/B + 1)$ transfers are enough. As in ordinary 2-way Mergesort, all merges during a pass perform $\mathcal{O}(N/B + 1)$ transfers. Similarly, multiplying this by the number of passes yields $\mathcal{O}(N/B \cdot \log N / \log K + 1)$.

How large K does our cache allow? Each scan requires its own cached block, which is $K+1$ blocks total. Another $K-1$ blocks are more than enough for the heap, so $M \geq 2BK$ is sufficient. If we know M and B, we can set $K = M/2B$. Then the I/O complexity will reach $\mathcal{O}(N/B \cdot \log N / \log(M/B)+1)$. This is actually known to be optimal — surprisingly, there is a matching lower bound (FIXME: reference) not only for sorting, but for only permuting items.

In the cache-oblivious model, we have no chance to pick the right number of ways. Still, there exists a rather complicated algorithm called Funnelsort (FIXME: ref) which achieves the same I/O complexity, at least asymptotically. We refer readers to the original article.

5.3 Matrix transposition

Another typical class of algorithms working with large data deals with matrices. We will focus on a simple task: transposition of a square matrix.

Matrices are traditionally stored in row-major order.[1] That is, the whole $N \times N$ matrix is stored as a single array of N^2 items read row by row. Accessing the matrix row by row therefore involves a sequential scan of the underlying array, so it takes $\mathcal{O}(N^2/B + 1)$ block transfers.

What about column-by-column access? Reading a single column is much more expensive: if the rows are large, each item in the column is located in a different block. When we

[1] A particular exception is Fortran and MATLAB, which use column-major order for some mysterious reason.

switch to the next column, we usually need the same N blocks. But unless the cache is big (meaning $M \in \Omega(NB)$), most blocks are already evicted from the cache. The total I/O complexity can therefore reach $\Theta(N^2)$.

A simple algorithm for transposing a matrix walks through the lower triangle and swaps each item with the corresponding item in the upper triangle. However, this means that if we are accessing one of the triangles row-by-row, the other will be accessed column-by-column. So the whole transposition will transfer $\Theta(N^2)$ blocks.

Using tiles

We will show a simple cache-aware algorithm with better I/O complexity. We split the matrix to *tiles* — sub-matrices of size $d \times d$, with smaller, possibly rectangular tiles at the border if N is not a multiple of d. The number of tiles is $\lceil N/d \rceil^2 \leq (N/d+1)^2 \in \mathcal{O}(N^2/d^2+1)$.

Each tile can be transposed on its own, then we have to swap each off-diagonal tile with its counterpart in the other triangle. Obviously, the time complexity is still $\Theta(N^2)$. With a bit of luck, a single tile will fit in the cache completely, so its transposition will be I/O-efficient.

First, we shall analyse the case in which the size of the matrix N is a multiple of the block size B. If it is so, we can align the start of the matrix to the beginning of a block, so the start of each row will be also aligned. If we set $d = B$, every tile will be also aligned and each row of the tile will be a complete block. If we have enough cache, we can process a tile in $\mathcal{O}(B)$ I/O operations. As we have N^2/B^2 tiles, the total I/O complexity is $\mathcal{O}(N^2/B)$.

For this algorithm to work, the cache must be able to hold two tiles at once. Since each tile contains B^2 items, this means $M \geq 2B^2$. An inequality of this kind is usually called *tall-cache property* and it is satisfied by most hardware caches (but not when caching a disk with large blocks in main memory). Intuitively, it means that if we view the cache as a rectangle whose rows are the blocks, the rectangle will be higher than it is wide. More generally, a tall cache has $M \in \Omega(B^2)$. For every possible constant in Ω, we can make tiles constant-times smaller to fit in the cache and the I/O complexity of our algorithm will not change asymptotically.

Now, what if N is not divisible by B? We lose all alignment, but we will prove that the algorithm still works. Consider a $B \times B$ tile. In the worst case, each row spans 2 blocks. So we need $2B$ I/O operations to read it into cache, which is still $\mathcal{O}(B)$. The cache must contain at least $4B^2$ items, but this is still within limits of our tall-cache assumption.

To process all $\mathcal{O}(N^2/B^2+1)$ tiles, we need $\mathcal{O}(N^2/B+B)$ operations. As usual, this can be improved to $\mathcal{O}(N^2/B+1)$ if we realize that the additional term is required only in cases where the whole matrix is smaller than a single block.

We can conclude that in the cache-aware model, we can transpose a $N \times N$ matrix in time $\Theta(N^2)$ with $\mathcal{O}(N^2/B+1)$ block transfers. This is obviously optimal.

Divide and conquer

Optimal I/O complexity can be achieved even in the cache-oblivious model, but we have to be a little bit more subtle. Since we cannot guess the right tile size, we will try to approximate it by recursive subdivisions. This leads to the following divide-and-conquer algorithm.

For a moment, we will assume that N is a power of 2. We can split the given matrix A to 2×2 *quadrants* of size $N/2 \times N/2$. Let us call them A_{11}, A_{12}, A_{21}, and A_{22}. The diagonal quadrants A_{11} and A_{22} will be transposed recursively. The other quadrants A_{12} and A_{21} will have to be transposed, but also swapped. (This is the same idea as in the previous algorithm, but with tiles of size $N/2$.)

However, transposing first and then swapping would spoil time complexity (try to prove this). We will rather apply divide and conquer on a more general problem: given two matrices, transpose them and swap them. This problem admits a similar recursive decomposition (see figure 5.1). If we split two matrices A and B to quadrants, we have to transpose-and-swap A_{11} with B_{11}, A_{22} with B_{22}, A_{12} with B_{21}, and A_{21} with B_{12}.

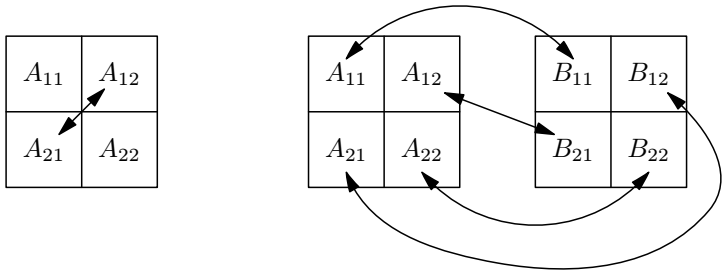

Figure 5.1: Divide-and-conquer matrix transposition

A single transpose-and-swap (TS) therefore recurses on 4-subproblems, which are again TS. A transpose (T) problem recurses on 3 sub-problems, two of which are T and one is TS. All these sub-problems have size $N/2$.

To establish time and I/O complexity, we consider the tree of recursion. Every node corresponds to a T or TS problem. It has 3 or 4 children for its sub-problems. At level i

(counted from the root, which is at level 0), we have at most 4^i nodes with sub-problems of size $N/2^i$. Therefore, the height of the tree is $\log N$ and it has at most $4^{\log N} = N^2$ leaves. Since all internal nodes have least 2 children (in fact, 3 or 4), there are less internal nodes than leaves.

In every TS leaf, we swap a pair of items. In every T leaf, nothing happens. Internal nodes only redistribute work and they do not touch items. So every node takes $\mathcal{O}(1)$ time and the whole algorithm finishes in $\mathcal{O}(N^2)$ steps.

To analyze I/O complexity, we focus on the highest level, at which the sub-problems correspond to tiles from the previous algorithm. Specifically, we will find the smallest i such that the sub-problem size $d = N/2^i$ is at most B. Unless the whole input is small and $i = 0$, this implies $2d = N/2^{i-1} > B$. Therefore $B/2 < d \leq B$.

To establish an upper bound on the optimal number of block transfers, we show a specific caching strategy. Above level i, we cache nothing — this is correct, since we touch no items. (Well, we need a little cache for auxiliary variables like the recursion stack, but this is asymptotically insignificant.) When we enter a node at level i, we load the whole sub-problem to the cache and compute the whole subtree in the cache. Doing this in all nodes of level i is equivalent to running the cache-aware algorithm for $d \in \Theta(B)$, which requires $\mathcal{O}(N^2/B)$ block transfers in total, provided that the cache is tall enough.

Finally, what if N is not a power of 2? When we sub-divide a matrix of odd size, we need to round one half of N down and the other up. This makes off-diagonal quadrants rectangular. Fortunately, it is easy to prove that all matrices we generate are either square, or almost-square, meaning that the lengths of sides differ by at most 1. We are leaving the proof as an exercise to the reader. For almost-square matrices, our reasoning about the required number of block transfers still applies.

We have reached optimal I/O complexity $\mathcal{O}(N^2/B+1)$ even in the cache-oblivious model. The algorithm still runs in time $\Theta(N^2)$. In a real implementation, we can reduce its overhead by stopping recursion at sub-problems of some fixed size greater than 1. This comes at the expense of I/O complexity in very small caches, but these do not occur in practice.

The technique we have used is quite typical for design of cache-oblivious algorithms. We start with a cache-aware algorithm based on decomposing the problem to tiles of some kind. We show that for a particular tile size (typically related to parameters of the cache), the tiles fit in the cache. Then we design a cache-oblivious algorithm, which replaces knowledge of the right tile size by successively refined decomposition. At some level of the refinement — which is not known to the algorithm, but known to us during the analysis — the size of tiles is right up to a constant, so the I/O complexity matches the cache-aware algorithm up to a constant.

5.4 Model versus reality

Warning: We used a different notation at the lecture in April 2021.

Our theoretical models of caching assume that the cache is controlled in an optimal way. This is obviously impossible to achieve since it requires knowledge of all future memory accesses. Surprisingly, we will show that there exists a deterministic strategy which aproximates the optimal strategy well enough.

We will compare caching strategies by the following experiment: The main memory is divided to blocks, each identified by its address. We are given a sequence $a_1, a_2, \ldots, a_n$ of requests for particular blocks. The strategy controls a cache of M block-sized slots. Initially, each slot is either empty or it contains a copy of an arbitrary memory block. Every time the strategy receives a new request, it points to a slot containing the requested block, if there is any. Otherwise the cache *misses* and it has to replace contents of one slot by that block. The obvious measure of cache efficiency is the number C of cache misses. We will call it the *cost* of the strategy for the given access sequence.

The *optimal strategy (OPT)* knows the whole sequence of requests in advance. It is easy to see that the optimal strategy always uses the the cache slot, whose block is needed farthest in the future (at best never). Let us denote the cost of the optimal algorithm by C_{OPT}.

We are looking for *online strategies,* which know only the current state of the cache and the current request. Hypotetically, they could also know the history of past requests, but it does not give any advantage. A typical on-line strategy, which is often used in real caches, is *least-recently used (LRU).* It keeps the cache slots sorted by the time of their last use. When the cache misses, it replaces the slot which is longest unused. The cost of LRU shall be called C_{LRU}.

We would like to prove that LRU is k-competitive, that is $C_{\mathrm{LRU}} \leq k \cdot C_{\mathrm{OPT}}$ for some constant k independent of M. Unfortunately, this is impossible to achieve:

Theorem: For every cache size M and every $\varepsilon > 0$, there exists a sequence of requests for which $C_{\mathrm{LRU}} \geq (1 - \varepsilon) \cdot C \cdot C_{\mathrm{OPT}}$.

Proof: The sequence will consist of K copies of $1, \ldots, M + 1$. The exact value of K will be chosen later.

The number of cache misses on the first M blocks depends on the initial state of the cache. When they are processed, LRU's cache will contain exactly blocks $1, \ldots, M$. The next request for $M + 1$ will therefore miss and the least-recently used block 1 will be replaced by $C + 1$. The next access will request block 1, which will be again a cache miss, so 2 will be replaced by 1. And so on: all subsequent requests will miss.

We will show a better strategy, which will serve as an upper bound on the optimum strategy. We divide the sequence to *epochs* of size M. Two consecutive epochs overlap in $M-1$ blocks. Except for the initial epoch, every other epoch can re-use $M-1$ cached blocks from the previous epoch. When it needs to access the remaining block, it replaces the one block, which is not going to be accessed in the current epoch. Thus it has only one cache miss per M requests.

Except for the first epoch, the ratio between C_{LRU} and C_{OPT} is exactly M. The first epoch can decrease this ratio, but its effect can be diminished by increasing K. □

Still, all hope is not lost. If we give LRU an advantage of larger cache, the competitive ratio can be bounded nicely. Let us denote LRU's cache size M_{LRU} and similarly M_{OPT} for the optimal strategy.

Theorem: For every $M_{\mathrm{LRU}} > M_{\mathrm{OPT}} \geq 1$ and every request sequence, we have:

$$C_{\mathrm{LRU}} \leq \frac{M_{\mathrm{LRU}}}{M_{\mathrm{LRU}} - M_{\mathrm{OPT}}} \cdot C_{\mathrm{OPT}} + M_{\mathrm{OPT}}.$$

Proof: We split the request sequence to *epochs* $E_0, \ldots, E_t$ such that the cost of LRU in each epoch is exactly M_{LRU}, except for E_0, where it is at most M_{LRU}.

Now we consider an epoch E_i for $i > 0$. We distinguish two cases:

a) All blocks on which LRU missed are distinct. This means that the access sequence for this epoch contains at least M_{LRU} distinct blocks. OPT could have had at most M_{OPT} of them in its cache when the epoch began, so it still must miss at least $M_{\mathrm{LRU}} - M_{\mathrm{OPT}}$ times.

b) Otherwise, LRU misses twice on the same block b. After the first miss, b was at the head of the LRU list. Before the next miss, it must have been replaced, so it must have moved off the end of the LRU list. This implies that there must have been at least M_{LRU} other blocks accessed in the meantime. Again, OPT must miss at least $M_{\mathrm{LRU}} - M_{\mathrm{OPT}}$ times.

So in every epoch but E_0 the ratio $C_{\mathrm{LRU}}/C_{\mathrm{OPT}}$ is at most $M_{\mathrm{LRU}}/(M_{\mathrm{LRU}} - M_{\mathrm{OPT}})$. Averaging over all epochs gives the desired bound.

Epoch E_0 is different. First assume that both LRU and OPT start with an empty cache. Then all blocks on which LRU misses are distinct, so OPT must miss on them too and the ratio $C_{\mathrm{LRU}}/C_{\mathrm{OPT}}$ is at most 1. If LRU starts with a non-empty cache, the ratio can only decrease — when the block is already cached at the beginning, we save a cache miss, but the contents of the LRU list stay the same. When OPT starts with non-empty cache,

it can save up to M_{OPT} misses, which is compensated by the extra M_{OPT} term in the statement of the theorem. □

Corollary: If we set $M_{\mathrm{LRU}} = 2 \cdot M_{\mathrm{OPT}}$, then $C_{\mathrm{LRU}} \leq 2C_{\mathrm{OPT}} + M_{\mathrm{OPT}}$. So on a long enough access sequence, LRU is $(2 + \varepsilon)$-competitive.

This comparison may seem unfair: we gave LRU a larger cache, so we are comparing LRU on a cache of size M with OPT on a cache of size $M/2$. However, for all our algorithms the dependency of I/O complexity on the cache size M is such that when we change M by a constant factor, I/O complexity also changes by at most a constant factor. So emulating the optimal cache controller by LRU does not change the asymptotics.