6.897: Advanced Data Structures	Spring 2003

Lecture 15 — April 16, 2003

Prof. Erik Demaine — *Scribes: Hans Robertson and Daniel Aguayo*

1 Review of the Cache-Oblivious Model

The last lecture introduced the *external-memory model*, which models a two-level memory system. The first level of memory (the *cache*) of size M is partitioned into blocks of size B. The second level (*main memory* or *disk*) is limitless but requires us to pay a charge for each access (*memory/block transfer*). In this model, we measure performance by the number of memory transfers (cache misses).

In the *cache-oblivious model*, we still measure performance by the number of memory transfers, but the algorithm has no knowledge of the values of M and B. Also, we do not have explicit control over the cache (e.g., the ability to request, "fetch block i and put it in cache slot j"). Instead, block fetches occur automatically, triggered by element accesses, and the cache system implements an omnisciently optimal block-replacement strategy.

Efficient algorithms using this model afford us the advantage that they do not need to be "tuned" for a given machine or memory subsystem. Indeed, because the amount of cache memory often varies across the same machine model and is sometimes not exposed to the operating system, this tuning can be difficult in practice. More importantly, such algorithms will perform well when used on machines with multi-level memory hierarchies.

For more information on caches, see [HP03].

1.1 A Simple Example

One example of an operation which translates directly to the cache-oblivious model is scanning N elements in memory. In the external-memory model, if the elements are stored in adjacent positions in memory with the first element on a block boundary, scanning requires $\lceil N/B \rceil$ memory transfers.

In the cache-oblivious model, without knowledge of the block size B, we cannot ensure that the first element lies on a block boundary, so scanning may require an extra block transfer: $\lceil N/B \rceil + 1$ transfers in total. This bound is very close to the external-memory performance.

1.2 Block Replacement

Before continuing, we address some details of the cache behavior in the cache-oblivious model. One important aspect of a cache design is the *block replacement strategy* that determines which current block should be evicted from the cache when we fetch a new block from a lower level of memory.

There are several simple possibilities:

- **Least Recently Used (LRU).** Evict the block which was last accessed the longest time ago.
- **Most Recently Used.** Evict the block which was most recently accessed. (This behavior seems counterintuitive, but might make sense if we know that data recently referenced is less likely to be referenced again soon.)
- **First In, First Out (FIFO).** Evict the block which has been in the cache the longest.
- **Random.** Evict a block chosen at random.
- **Optimal (OPT).** Evict the block to be accessed which will be accessed furthest in the future.

Sleator and Tarjan [ST85] showed that both LRU and FIFO are within a constant factor of the optimal strategy, if we allow changing both the resource bounds (M and B) and the time bound. Specifically:

$$\text{cost}(\text{LRU with cache of size } 2M) \leq 2 \times \text{cost}(\text{OPT with cache of size } M)$$

and

$$\text{cost}(\text{FIFO with cache of size } 2M) \leq 2 \times \text{cost}(\text{OPT with cache of size } M)$$

With this in mind, we can assume an optimal replacement strategy in the cache-oblivious model. For most algorithms, changing M by a constant factor changes the running time by only a constant factor, and we are happy.

1.3 Associativity

The cache-oblivious model further assumes that any memory block can be stored anywhere in the cache. This property is known as *full associativity*. Many caches allow a given block to be stored only in a subset of the available slots. If all sets are of size k, the cache is said to be *k-way set associative*. If $k = 1$, the cache is said to be *direct mapped*. On a new insert, the block replacement strategy is used to determine which of these k blocks should be ejected.

Frigo et al. [FLPR99] showed that automatic replacement on a fully associative cache can be simulated using manual replacement on a direct-mapped cache. The basic idea is to use two-way universal hashing; by hashing the block addresses, you avoid conflicts. Such a scheme is outside the cache-oblivious model, because it requires knowing B. Because of the constant-factor overheads, it is unclear whether this scheme is practical, but at least it is comforting from a theoretical point of view.

2 External-Memory Linked Lists

Consider the problem of storing a linked list on a machine with multiple levels of memory. First we address the problem in the external-memory model. In Section 3, we present a solution in the cache-oblivious model.

Our linked list should support the following operations:

- INSERT: Insert a new element between two given elements.
- DELETE: Delete a given element.
- TRAVERSE(K): Given an element, visit the next K elements.

Our goal is to support traversals in $O(\lceil K/B \rceil)$ memory transfers. A fairly simple data structure achieves this bound and achieves updates in $O(1)$ memory transfers. We cluster the elements of the list into $\Theta(N/B)$ chunks, each of size between $B/2$ and B. Each chunk is stored in one block.

The underlying structure is an ordinary linked list. We only impose that each block stores consecutive elements in the list. As a result, traversals take $O(\lceil K/B \rceil)$ memory transfers.

Updates are also straightforward. An insertion proceeds as in a normal linked list, except if a memory block becomes too full, in which case we split the block in two and put the new block at the end of memory. Similarly, if deleting an element results in a block merge and a resulting hole in memory, we just move the last block in memory into the gap. Each update takes $O(1)$ memory transfers.

In the external-memory model with two levels, cache and disk, elements may be arbitrarily ordered within each block, and the blocks may be stored in arbitrary locations. If we were to try to extend this scheme to multiple levels, the layout would be more restricted in order to avoid higher-level cache misses. In this case, the data structure gets worse because of $O(L)$ levels of recursion to handle L levels of memory.

Knowledge of B is essential to this scheme. In the next section we devise a cache-oblivious scheme.

3 Cache-Oblivious Linked Lists

A linked list in the cache-oblivious model is more challenging.

3.1 Via Ordered-File Maintenance

If we use the ordered-file maintenance structure presented in Lecture 14, we immediately obtain the following bounds:

- INSERT and DELETE in $O((\lg^2 N)/B)$ amortized memory transfers, assuming $M \geq 2B$. When inserting an element, we have two interleaved scans, one to the left and one to the right of the new element. Rebalancing also uses two such scans.
- TRAVERSAL(K) in $O(\lceil K/B \rceil)$ memory transfers. Because of the lower bound we imposed on the density of elements in the structure, the gaps between elements are of $O(1)$ size.

This is the best cache-oblivious update bound known if we want a $O(\lceil K/B \rceil)$ worst-case traversal bound.

3.2 Sacrificing Traversals Slightly

Bender et al. [BCDF00] have shown that if we relax the worst-case traversal bound slightly and store the elements out-of-order, we can do updates better. This is somewhat counterintuitive because, by not knowing B, we don't know just how disordered the elements can be.

Nonetheless, the solution supports updates in $O((\lg\lg n)^{2+\epsilon}/B)$ amortized memory transfers, for any $\epsilon > 0$. TRAVERSAL(K) uses $O(\lceil K/B \rceil + [B^\epsilon \text{ if } K \geq B^{1-\epsilon}])$ transfers. This traversal bound is suboptimal only when K is in the range from $B^{1-\epsilon}$ to B.

The algorithm which provides these bounds is somewhat complicated, so we will describe another.

3.3 Self-Adjusting Data Structure

We will describe a data structure which achieves the same bounds as the external-memory data structure, except that both bounds become amortized:

- INSERT and DELETE in $O(1)$ amortized memory transfers.
- TRAVERSAL(K) in $O(\lceil K/B \rceil)$ amortized memory transfers.

Updates with a constant number of memory transfers are simple. For INSERT, we just put the new element at the end of memory, and update the pointers. For DELETE, we just update the pointers, and erase the element. We don't make any effort to fill the gap. Both operations run in $O(1)$ transfers.

For TRAVERSAL, we simply visit the elements. If we're lucky, and the elements were inserted in order, then we have a *run* of consecutive elements and the traversal uses the desired number of memory transfers. In the worst case, however, we might end up with K disjoint runs.

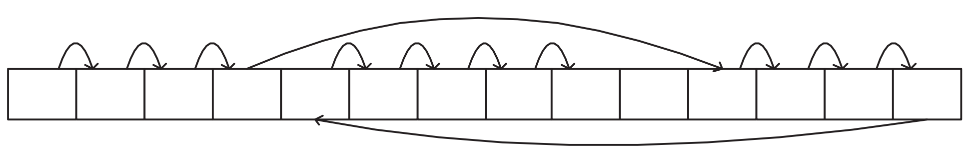

Figure 1: A linked list with 3 runs.

We define r to be the actual number of runs. Then the cost of the traversal is no more than $2r + O(\lceil K/B \rceil)$. To improve the amortized performance, we can easily merge $r - 2$ runs—all but the first and last—into one run at the end of memory. We omit the first and last runs because moving these might break an existing run, negating our attempted improvement.

3.4 Analysis of Traversals

Every run in the data structure must have been created by an update. The cost of a traversal is therefore $O(\lceil K/B \rceil)$ amortized, because we can charge $r - 3$ of the $r + O(\lceil K/B \rceil)$ cost to the updates which created those runs and were removed by the TRAVERSAL.

3.5 Recompactification

In this scheme, each traversal might increase the size of the data structure, leaving long gaps in the middle. Thus, when the size of the structure grows by a constant factor, it is necessary to "recompactify" by shifting elements left to fill the holes, and updating all the pointers. Recompactifying requires $O(N)$ memory transfers, but it too amortizes to $O(1)$.

References

[BCDF00] Michael A. Bender, Richard Cole, Erik D. Demaine, and Martin Farach-Colton, "Scanning and Traversing: Maintaining Data for Traversals in a Memory Hierarchy," in Proceedings of the 10th Annual European Symposium on Algorithms (ESA 2002), Lecture Notes in Computer Science, volume 2461, Rome, Italy, September 17-21, 2002, pages 139-151.

[FLPR99] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in Proc. 40th IEE Sympos. Found. Comp. Sci., New York, Oct. 1999, pages 285-297.

[HP03] John Hennessy and David Patterson, "Computer Architecture: A Quantitative Approach.3rd Ed." Morgan Kaufmann, 2003.

[ST85] Daniel Sleator and Robert Tarjan,"Amortized efficiency of list updates and paging rules," Communications of the ACM, 28:202–208, 1985.