CS787: Advanced Algorithms

Scribe: Dalibor Zeleny and Xiaoyong Chai **Lecturer:** Shuchi Chawla
Topic: Caching Algorithms **Date:** October 31 2007

Today we discussed the issue of caching—a problem that is well suited for the use of online algorithms. We talked about deterministic and randomized caching algorithms, and proved bounds on their competitive ratios.

20.1 Caching

In today's computers, we have to make tradeoffs between the amount of memory we have and the speed at which we can access it. We solve the problem by having a larger but slower *main memory*. Whenever we need data from some page of the main memory, we must bring it into a smaller section of fast memory called the *cache*. It may happen that the memory page we request is already in the cache. If this is the case, we say that we have a cache *hit*. Otherwise we have a cache *miss*, and we must go in the main memory to bring the requested page. It may happen that the cache is full when we do that, so it is necessary to *evict* some other page of memory from the cache and replace it with the page we read from main memory, or we can choose not to place the page we brought from main memory in the cache. Cache misses slow down programs because the program cannot continue executing until the requested page is fetched from the main memory. Managing the cache in a good way is, therefore, necessary in order for programs to run fast. The goal of a *caching algorithm* is to evict pages from the cache in a way that minimizes the number of cache misses. A similar problem, called *paging*, arises when we bring pages from a hard drive to the main memory. In this case, we can view main memory as a cache for the hard drive.

For the rest of this lecture, assume that the cache has a capacity of k pages and that main memory has a capacity of N pages. For example consider a cache with $k = 3$ pages and a main memory with $N = 5$ pages. A program could request page 4 from main memory, then page 1, then 2 etc. We call the sequence of memory pages requested by the program the *request sequence*. One request sequence could be 4, 1, 2, 1, 5, 3, 4, 4, 1, 2, 3 in our case. Initially, the cache could contain pages 1, 2, and 3. When we execute this program, we need access to page number 4. Since it's not in the cache, we have a cache miss. We could evict page 2 from the cache and replace it with page 4. Next, our program needs to access page 1, which is in the cache. Hence, we have a cache hit. Now our program needs page 2, which is not in the cache, so we have a miss. We could choose to evict page 1 from the cache and put page 2 in its place. Next, the program requests page 1, so we have another miss. This time we could decide to just read page 1 and not place it in the cache at all. We continue this way until all the memory requests are processed.

In general, we don't know what the next terms in the request sequence are going to be. Thus, caching is a place where we can try to apply an online algorithm. For a request sequence σ and a given algorithm ALG, we call $\text{ALG}(\sigma)$ the *cost* of the algorithm ALG on request sequence σ, and define it to be the number of cache misses that happen when ALG processes the memory requests and maintains the cache.

20.2 Optimal Caching Algorithm

If we knew the entire request sequence before we started processing the memory requests, we could use a greedy algorithm that minimizes the number of cache misses that occur. Whenever we have a cache miss, we go to main memory to fetch the memory page p we need. Then we look at this memory page and all the memory pages in the cache. We evict the page for which the next request occurs the latest in the future from among all the pages currently in the cache, and replace that page with p in the cache. If the next time p is requested comes after the next time all the pages in the cache are requested again, we don't put p in the cache.

Once again, take our sample request sequence 4, 1, 2, 1, 5, 3, 4, 4, 1, 2, 3 and assume that pages 1 through 3 are in the cache. The optimal algorithm has a cache miss when page 4 is requested. Since page 4 is requested again later than pages 1 through 3 are, the algorithm doesn't put 4 in the cache and doesn't evict any page. The next miss occurs when page 5 is requested. Once again, the next times pages 1 through 3 are requested occur before page 5 is requested the next time, so page 5 is not brought in the cache. The next cache miss happens when page 4 is requested. At that point, page 3 gets evicted from the cache and is replaced with page 4 because pages 1, 2 and 4 get requested again before page 3 does. Finally, the last cache miss occurs when page 3 is requested (last term in the request sequence). At this point, the algorithm could choose to evict page 1 and put page 3 in the cache instead. Thus, the optimal algorithm has 4 cache misses on this request sequence.

Notice that in the case of caching, we have an optimal algorithm assuming that we know the entire input (in our case the input is the request sequence). Last lecture, we discussed the list update problem. We do not have an algorithm that runs in polynomial time and can find an optimal solution to the list update problem, even when it has access to the entire input at the very beginning. This fact makes the analysis of online algorithms for caching easier because we know what algorithm we compare the online algorithms against.

For the remainder of this lecture, we will assume that a caching algorithm always has to bring the memory page in the cache on a cache miss, and doesn't have the option of just looking at it and not putting it in the cache, as it will make our proofs simpler. This version is also much closer to what happens in hardware.

20.3 Deterministic Caching

There are many deterministic online algorithms for caching. We give some examples below. In each of the cases, when a cache miss occurs, the new memory page is brought into the cache. The name of the algorithm suggests which page should be evicted from the cache if the cache is full.

LRU (Least Recently Used) The page that has been in the cache for the longest time without being used gets evicted.

FIFO (First In First Out) The cache works like a queue. We evict the page that's at the head of the queue and then enqueue the new page that was brought into the cache.

LFU (Least Frequently Used) The page that has been used the least from among all the pages in the cache gets evicted.

LIFO (Last In First Out) The cache works like a stack. We evict the page that's on the top of the stack and then push the new page that was brought in the cache on the stack.

The first two algorithms, LRU and FIFO have a competitive ratio of k where k is the size of the cache. The last two, LFU and LIFO have an *unbounded competitive ratio*. This means that the competitive ratio is not bounded in terms of the parameters of the problem (in our case k and N), but rather by the size of the input (in our case the length of the request sequence).

First we show that LFU and LIFO have unbounded competitive ratios. Suppose we have a cache of size k. The cache initially contains pages 1 through k. Also suppose that the number of pages of main memory is $N > k$. Suppose that the last page loaded in the cache was k, and consider the request sequence $\sigma = k+1, k, k+1, k, \ldots, k+1, k$. Since k is the last page that was put in the cache, it will be evicted and replaced with page $k+1$. The next request is for page k (which is not in the cache), so we have a cache miss. We bring k in the cache and evict $k+1$ because it was brought in the cache last. This continues until the entire request sequence is processed. We have a cache miss for each request in σ, whereas we have only one cache miss if we use the optimal algorithm. This cache miss occurs when we bring page $k+1$ at the beginning and evict page 1. There are no cache misses after that. Hence, LIFO has an unbounded competitive ratio.

To demonstrate the unbounded competitive ratio of LFU, we again start with the cache filled with pages 1 through k. First we request each of the pages 1 through $k-1$ m times. After that we request page $k+1$, then k, and alternate them m times. This gives us $2m$ cache misses because each time k is requested, $k+1$ will be the least frequently used page in the cache so it will get evicted, and vice versa. Notice that on the same request sequence, the optimal algorithm makes only one cache miss. This miss occurs during the first request for page $k+1$. At that point, the optimum algorithm evics page 1 and doesn't suffer any cache misses afterwards. Thus, if we make m large, we can get any competitive ratio we want. This shows that LFU has an unbounded competitive ratio.

We now show that no deterministic algorithm can have a better competitive ratio than the size of the cache, k. After that, we demonstrate that the LRU algorithm has this competitive ratio.

Claim 20.3.1 *No deterministic online algorithm for caching can achieve a better competitive ratio than k, where k is the size of the cache.*

Proof: Let ALG be a deterministic online algorithm for caching. Suppose the cache has size k and that it currently contains pages 1 through k. Suppose that $N > k$. Since we know the replacement policy of ALG, we can construct an adversary that causes ALG to have a cache miss for every element of the request sequence. To do that, we simply look at the contents of the cache at any time and make a request for the page in $\{1, 2, \ldots, k+1\}$ that is currently not in the cache.

The only page numbers requested by the adversary are 1 through $k+1$. Thus when the optimal algorithm makes a cache miss, the page it evicts will be requested no sooner than after at least k other requests. Those requests will be for pages in the cache. Thus, another miss will occur after

at least k memory requests. It follows that for every cache miss the optimal algorithm makes, ALG makes at least k cache misses, which means that the competitive ratio of ALG is at least k. ■

Claim 20.3.2 *LRU has a competitive ratio of k.*

Proof: First, we divide the request sequence σ into phases as follows:

- Phase 1 begins at the first page of σ;
- Phase i begins at the first time we see the k-th distinct page after phase $i-1$ has begun.

As an example, suppose $\sigma = 4, 1, 2, 1, 5, 3, 4, 4, 1, 2, 3$ and $k = 3$. We divide σ into three phases as in Figure 20.3.1 below. Phase 2 begins at page 5 since page 5 is the third distinct page after phase 1 began (pages 1 and 2 are the first and second distinct pages, respectively).

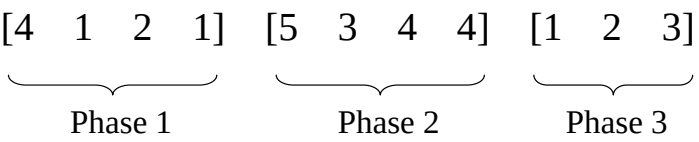

Figure 20.3.1: Three phases of sequence σ.

Next, we show that OPT makes at least one cache miss each time a new phase begins. Denote the j-th distinct page in phase i as p_j^i. Consider pages p_2^i-p_k^i and page p_1^{i+1}. These are k distinct pages by the definition of a phase. Then if none of the pages p_2^i-p_k^i incur a cache miss, p_1^{i+1} must incur one. This is because pages p_2^i-p_k^i and p_1^{i+1} are k distinct pages, page p_1^i is in the cache, and only k pages can reside in the cache. Let N be the number of phases. Then we have $\text{OPT}(\sigma) \geq N - 1$. On the other hand, LRU makes at most k misses per phase. Thus $\text{LRU}(\sigma) \leq kN$. As a result, LRU has a competitive ratio of k. ■

This proof relies on the fact that on a page fault, OPT (and any other caching algorithm) has to bring the page from memory into the cache, and evict another page if the cache is full. If this were not the case, we could only demonstrate a competitve ratio of $2k$. It is also possible to show that this bound on the competitive ratio is tight.

20.3.1 Deterministic 1-Bit LRU

As a variant of the above LRU algorithm, the 1-bit LRU algorithm (also known as the marking algorithm) associates each page in the cache with 1 bit. If a cache page is recently used, the corresponding bit value is 1 (marked); otherwise, the bit value is 0 (unmarked). The algorithm works as follows:

- Initially, all cache pages are unmarked;
- Whenever a page is requested:
    - If the page is in the cache, mark the page;

- Otherwise:
  - If there is at least one unmarked page in the cache, evict an arbitrary unmarked page, bring the requested page in, and mark it;
  - Otherwise, unmark all the pages and start a new phase.

Following the proof of Claim 20.3.2, we can easily see that the above deterministic 1-bit LRU algorithm also has a competitive ratio of k.

20.4 Randomized 1-Bit LRU

Given a deterministic LRU algorithm, an adversary can come up with a worst-case scenario by always requesting the page which was just evicted. As a way to improve the competitive ratio, we consider a randomized 1-bit LRU algorithm in which a page is evicted randomly. Before we go into the algorithm, we shall define the competitive ratio for a randomized online algorithm.

Definition 20.4.1 *A randomized online algorithm* ALG *has a* competitive ratio r *if for* $\forall\sigma$,

$$E[\mathrm{ALG}(\sigma)] \leq r \cdot \mathrm{OPT}(\sigma) + c,$$

where c is a constant independent of σ.

Note that the competitive ratio of a randomized algorithm is defined over the expected performance of the algorithm rather than the worst-case performance.

The randomized 1-bit LRU algorithm is rather simple: at every step, run the 1-bit LRU algorithm and evict an unmarked page uniformly at random when necessary.

Claim 20.4.2 *The randomized 1-bit LRU algorithm has a competitive ratio of $2H_k$, where H_k is the k-th harmonic number, which is defined as*

$$H_k = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{k} \in (\log k, \log(k) + 1).$$

Proof: Let us first analyze the expected number of cache misses in phase i. Recall that at the end of phase $i-1$, there are k pages in the cache. Let m_i denote the number of "new" pages in phase i, *i.e.*, those pages that were not requested in phase $i-1$. Call the rest of the pages in phase i "old" pages. Requesting a new page causes an eviction while requesting an old page may not. Assume for simplicity that the m_i new pages appear the first in the request sequence of phase i. Actually, one should convince oneself that this assumption indicates the worst-case scenario (think about how the number of cache misses changes if any one of these new page requests is delayed). Once the m_i new pages are requested, they are marked in the cache. The remaining $k - m_i$ pages in the cache are old pages. Now let us consider the probability of a cache miss when we request an old page for the first time, *i.e.*, when we request the (m_i+1)-th distinct page in phase i. This old page by definition was in the cache at the end of phase $i-1$. When requested, however, it may not be in the cache since it can be evicted for caching one of the m_i new pages. Then,

$$Pr[(m_i+1)\text{-th distinct page suffers a cache miss}] = \frac{\binom{k-1}{k-m_i}}{\binom{k}{k-m_i}} = \frac{m_i}{k}.$$

By the same reasoning, we have

$$Pr[(m_i+2)\text{-th distinct page suffers a cache miss}] = \frac{\binom{k-2}{k-m_i-1}}{\binom{k-1}{k-m_i-1}} = \frac{m_i}{k-1},$$

and so on.

Then the expected total number of cache misses in phase i is at most

$$m_i + \frac{m_i}{k} + \frac{m_i}{k-1} + \cdots + \frac{m_i}{k-(k-m_i)+1} \leq m_i H_k.$$

Let N be the number of phases. Then

$$E[\text{RAND-1-BIT-LRU}(\sigma)] \leq H_k \sum_{i=1}^{N} m_i. \tag{20.4.1}$$

Let us now analyze the number of cache misses of $\text{OPT}(\sigma)$. Note that requesting a new page p in phase i may not cause an eviction. This is because OPT may choose to fix p in the cache when p was requested in some earlier phase. In contrast, the randomized algorithm unmarks all pages in the cache at the beginning of phase i, and evicts randomly an unmarked page in case of a cache miss. Therefore, at the end of phase i, p cannot be in the cache. Although we cannot bound the number of cache misses in a single phase, we can bound the number in two consecutive phases. Let n_i be the number of cache misses of OPT in phase i. Since in total there are $k+m_i$ distinct pages in phases $i-1$ and i, at least m_i of them cause a miss. Thus we have

$$n_{i-1} + n_i \geq m_i,$$

and

$$\text{OPT}(\sigma) \geq \frac{1}{2} \sum_{i=1}^{N} m_i. \tag{20.4.2}$$

Combining Eq. (20.4.1) and Eq. (20.4.2), we have

$$E[\text{RAND-1-BIT-LRU}(\sigma)] \leq 2H_k \times \text{OPT}(\sigma).$$

In other words, the randomized 1-bit LRU algorithm is $2H_k$-competitive. ■

Claim 20.4.3 *No randomized online algorithm has a competitive ratio better than H_k.*

Proof: See the next lecture note. ■