Caching in theory and practice

Pavel Panchekha | October 16, 2012

Hello, my name is Pavel Panchekha. I was an intern at Dropbox back in '11, and one thing I've investigated are various caching algorithms. The Dropbox mobile client caches frequently-accessed files, so that viewing them doesn't require a network call. Both our Android and iOS clients use the LRU caching algorithm, which often selects good files to cache. But while this is the usual algorithm for caching, I wondered: are there better algorithms, and if not, why is LRU the best option?

Caches

Let's formalize the problem. We have a large set of files, and we'd like an algorithm to determine which (k) files to keep at any point. We're assuming all files have the same size. In fact, Dropbox stores files in 4MB blocks, so this simplification isn't too far off (we'll see later how to avoid it). To determine which files to keep, every time we want to use a file, we tell the cache to fetch it for us. If the cache has a file, it will give us its copy; if not, the cache has to fetch the new file, and it might also want to remove a file from its cache to make room for the new file.

Note also that we're stuck with an on-line algorithm: we can't predict what files a user will want in the future.

The cache needs to be fast, along two metrics. First, The cache should ensure that as many of the requests for files go to it (cache hit), not over the network (cache miss). Second, the overhead of using a cache should be small: testing membership and deciding when to replace a file should be as

fast as possible. Maximizing cache hits is the goal of the first part of this post; quickly implementing the cache will be the topic of the second part.

Competitive Analysis

How do we measure the worst case number of cache misses? Unlike a normal algorithm, our runtime is driven by the user's actions. So our worst-case performance corresponds to our worst-case user: one who maximally breaks our cache at every step.

But a pure adversarial analysis won't work, since a user can always make our cache perform badly by just requesting lots of files –eventually, some of them won't be cached.

The key is to compare how well our algorithm performs with how well our algorithm could possibly perform. We need some benchmark. Zero cache misses is a lower bound but is usually impossible. So instead, let's compare our algorithm with one that can "plan ahead" perfectly: let's compare our algorithm – which at any point only has the requests from the past – with some sort of optimal magic "future-seeing" algorithm.

More specifically, we're going to find the ratio of cache misses from our algorithm to the number of cache misses for the optimal algorithm. And then we're going to try to minimize this ratio across all possible sequences of file requests from the user. Generally, we'll argue that the algorithm we're analyzing will have at most (A) misses during any particular sequence of instructions, during which the optimal algorithm must have at least (O) misses; thus the "competitive ratio" is at most (A / O). This type of analysis is called competitive analysis.

In general, our method will be to pick a sequence that a chosen algorithm performs very poorly on. We find how many cache misses, (A), that algorithm sees for that sequence of requests. Usually, we'll be able to calculate (A) precisely. Then, we'll try to think up the cleverest possible way to cache files for that specific sequence; the number of cache misses we see we'll call (O). We'll usually find *some* possible way of caching files and calculate the number of cache misses for that, so we'll get an upper bound on (O). The competitive ratio is (A / O), and since we had an upper bound on (O), we get a lower bound on the competitive ratio. Furthermore, our algorithm could perform even worse

on a different sequence, so (A / O) is definitely a lower bound. This lets us say that some algorithm is really bad, but doesn't let us say that some algorithm is really good. We'll also prove some upper bounds on the competitive ratio, which will let us claim that some algorithms are optimal. Together, these will give us a way to compare caching algorithms.

Caching Algorithms

Before we go ahead to analyze a bunch of caching algorithms, we need caching algorithms to analyze. So let's quickly list a bunch of popular ones:

Most Recently Used

When we need to get rid of a file, we trash the one we just recently accessed. This algorithm incorporates information about how often a file is accessed in a perverse way – it prefers to keep around old data that is rarely used instead of data that is frequently accessed. But if you use many files, without using the same files over and over again (such as, say, when viewing a photo gallery), this algorithm works very well, since you're kicking out files you're unlikely to see again. In effect, browsing through a complete photo gallery can take up only one "slot" in the cache, since each access you kick out the previous photo in that gallery.

Least Recently Used

When we need to get rid of a file, we get rid of the one we haven't used in the longest time. This only requires keeping the access order of the files in the cache. By keeping files that we've recently accessed, it too tends to keep around files used more often; on the other hand, if a user's interest changes, the entire cache can relatively quickly become tuned to the new interests. But this cache tends to work poorly if certain files are accessed every once in a while, consistently, while others are accessed very frequently for a short while and never again.

Least Frequently Used

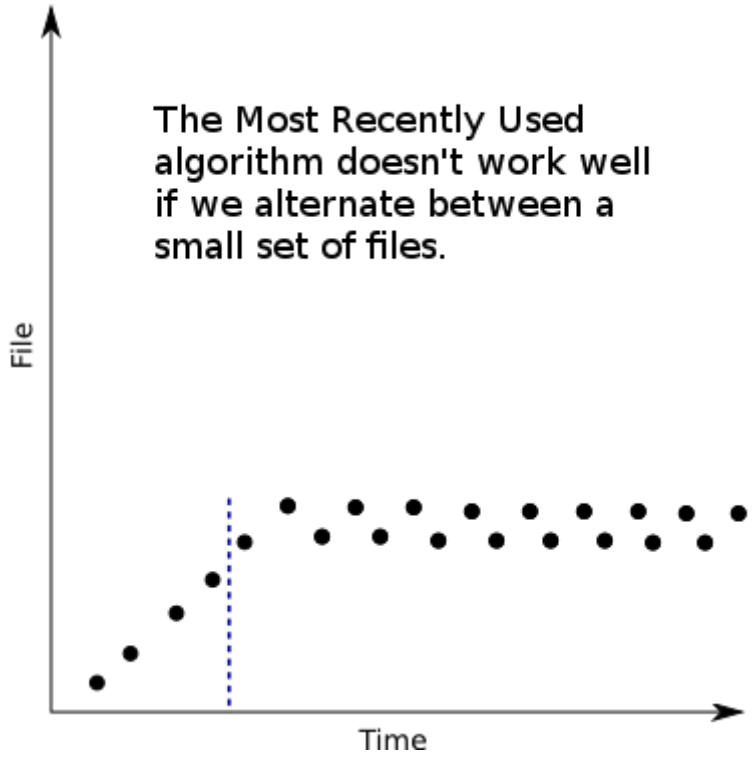
When we need to get rid of a file, we get rid of the one that is least frequently used. This requires keeping a counter on each file, stating how many times it's been accessed. If a file is accessed a lot for a while, then is no longer useful, it will stick around, so this algorithm

probably does poorly if access patterns change. On the other hand, if usage patterns stay stable, it'll (we hope) do well.

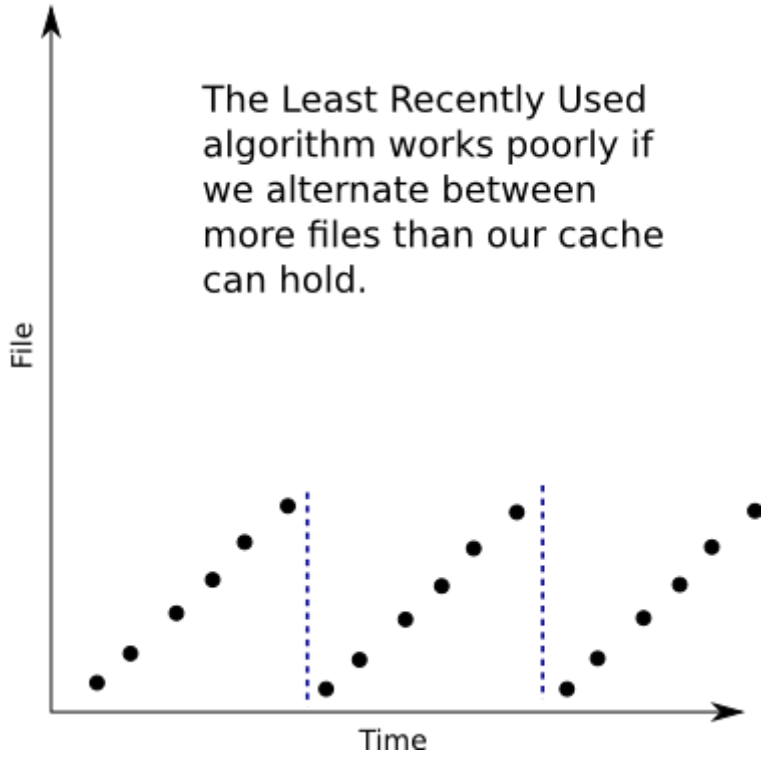
This is a nice batch of the more common and simple caching algorithms, so let's look at how we perform using the competitive analysis above.

The Most Recently Used and the Least Recently Used Algorithms

We can easily construct sequences to stump the Most Recently Used algorithm. For example, consider the sequence of file accesses (1, 2, dots, k, k+1, k, k+1, k, dots). Most Recently Used will kick out (k) to make room for (k+1), then kick out (k+1) to make room for (k), and so on. It will have a cache miss on every file lookup for this sequence of files. And it's so easy to do better: an optimal algorithm might, for example, kick out (1) to make room for (k+1), and never have a cache miss after that (since both (k) and (k+1) are in the cache after that). The optimal algorithm sees at most (k+1) cache misses, while Most Recently Used sees (N) cache misses, making it ((N / (k+1)))-competitive. Since we can make (N) as large as we want, this can be arbitrarily large – we might call the Most Recently Used algorithm (infty)-competitive. So, really bad.



The Least Recently Used algorithm is better. For example, on that input sequence, it does precisely what the optimal algorithm might do. But it still doesn't do that well. Imagine if our sequence of requests is for files (1, 2, dots, k, k+1, 1, 2, dots, k, k+1, dots). The Least Recently Used algorithm will miss every time, since for every request, the file requested was just kicked out. And the optimal algorithm can always just swap for the most-recently-requested file. So first, it would fail to find (k + 1) in the cache and replace (k) with it. Then it would fail to find (k) and replace (k – 1) with it. Then (k – 1) with (k – 2), and so on. This yields one cache miss every (k) requests; so if there are (N) requests total, the optimal algorithm would face (k + frac{N}{k}) failures (the "k" for populating the cache with (1, dots, k)), while the Least Recently Used algorithm would face (N) failures. Thus the Least Recently Used algorithm is at best (N / (k + (N / k)))-competitive, which for large (N) works out to be (k)-competitive.



This doesn't show that the Least Recently Used algorithm *is* (k)-competitive; it tells us that the Least Recently Used algorithm isn't better than (k)-competitive. But with a bit more effort, we can prove that the Least Recently Used algorithm is precisely (k)-competitive.

To do that, we'll have to make an argument about all possible input sequences. The core of the proof is to look at what must happen for LRU to fail. If the Least Recently Used algorithm has (k+1) cache misses, it must be because (k+1) new files were requested. But if this happens, at least one of those files wasn't in the cache that the optimal algorithm had (since it, too, can only cache (k) files).

To capture this property precisely, let's divide the sequence of files into phases – during each phase, only some (k) specific files are requested. LRU may fail on at most each of these new files before they are all in the cache – at most (k) times. Meanwhile, the optimal algorithm fails at least once, since at least one of those files isn't yet in the cache (if they all are, then we never ended the previous phase). So LRU is precisely (k)-competitive.

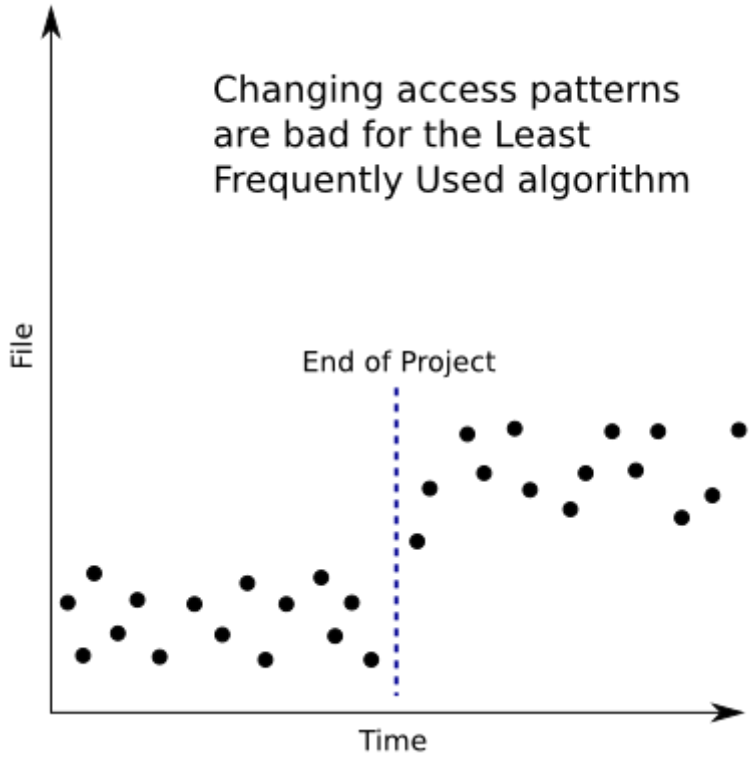
This doesn't sound that good. In a way, the larger our cache, the less impressively LRU performs. But in fact, our argument that Least Recently Used is (k)-competitive is applicable to any algorithm for which we can predict what files it will cache. So while (k) times worse than perfect seems pretty poor, it is in fact the best we can do (unless we use randomized algorithms; I'll discuss why not to do that in a bit).

LRU only made use of very basic timing information. A smarter algorithm, you might imagine, might actually maintain some popularity information: which files you use often, and which more rarely. Does it do any better?

Least Frequently Used

It seems that Least Frequently Used should do much better than LRU, since it incorporates actual information about how popular various files are, instead of just timing information.

But let's do the analysis proper, just in case. To make LFU perform poorly, we'd need to make it keep switching between two files, each time kicking one out to make room for the other. This might happen if we use, say, files (1, 2, dots, k-1) very frequently, and files (k) and (k+1) infrequently, but equally so. If we just request (1, dots, k-1) once or twice and then alternate between (k) and (k+1), this isn't too much of a problem, since eventually both (k) and (k+1) will be more frequently used than (1, dots, k-1) and will both be cached. But if we first use (1, dots, k-1) a bunch, so that neither (k) nor (k+1) are ever more popular than them, we can create a lot of cache misses. What we are setting up is a case where usage patterns change. First, we used (1) through (k-1) a lot, and then we changed to using (k) and (k+1) a lot.



An example such sequence requests (1, dots, k-1) (N) times, and then alternates between (k) and (k+1) (N) times. Both (k) and (k+1) are less frequently used than any of (1, dots, k-1), so each is kicked out to make room for the other. This leads to (k-1) cache misses to load (1) through (k-1) into the cache, and then (2N) cache misses for the requests to (k) and (k+1). On the other hand, the optimal algorithm could do so much better. For example, it could kick out (1) to make room for (k+1) when it stops being used, leading to (k+1) total cache misses. So the LFU algorithm had (2N + k – 1) misses, and the optimal algorithm had (k + 1). The quotient of these can be made arbitrarily large by increasing (N), so LFU can be arbitrarily bad.

This result is curious. LFU semi-intelligently made use of popularity data, but fared so much worse than LRU, which just made use of basic timing data. But, the cases that make LFU perform poorly are relatively real-world. For example, suppose you have a large project that you're working on, and then you finish said project and no longer access those files. Your cache would be storing those old

files instead of the new ones you're using. So our analysis told us something surprising: that LFU, which looked so promising, could actually be absurdly bad, in perhaps real-world situations.

In fact, if you think about it, LRU does make some use of popularity information. If a file is popular enough to be used more often than once every (k) times, it will always be in an LRU cache. But by forgetting any information more than (k) files ago, the LRU algorithm prevents really old files from taking precedence over new ones.

So, Why LRU?

You'd think it'd be possible to combine the best of LRU and LFU to make an algorithm that performs better than either. Turns out, yes and no.

When we proved LRU no better than (k)-competitive, we choose a sequence where the next file requested was always the file not in the cache. But we can do this for any deterministic algorithm! This means that the worst-case behavior of any deterministic algorithm is guaranteed to be no better than (k)-competitive.

But in a practical sense, better algorithms do exist. For reference, the ARC[1] and CAR[2] algorithms tend to outperform Least Recently Used caches. Of course, each has the same worst-case behavior that the Least Recently Used algorithm has, but they manage to trade off between frequent and recent items in a way that often leads to better performance in practice. Of course, both are more complex than the Least Recently Used algorithm.

We can get around the theoretical deficiencies of deterministic algorithms – that the user can predict which files aren't in the cache and thus keep requesting those – by having our algorithm make partially-random choices. This will make it harder for users to hit the worst case, but it often makes the algorithm perform worse in practice. The best a randomized algorithm can do is (O(log k)) (in fact, approximately the natural log of (k)); see Fiat et al.[3]. Randomized caching algorithms have the downside of behaving in unexpected ways for the user – "Why is that file taking so long to open, I just looked at it!". So in practice, they're rarely used.

Tangent: while randomized algorithms cannot be used directly in practice, they do tell us something about the expected performance of deterministic algorithms. This comes from a beautiful theorem by John von Neumann, called the Minimax Theorem. Imagine that the algorithm designer and his adversary play a game: the designer chooses a caching algorithm, the adversary a sequence of files, and then the winnings are decided based on how many cache misses the cache had. Phrased this way, algorithm design falls under the purview of game theory. We can represent a randomized algorithm as a strategy that involves choosing an algorithm at random from some set, and we can represent a randomized sequence of files as a random choice from a set of possible sequences.

Continuing the tangent, let's consider what the Minimax Theorem tells us about this game. The Minimax Theorem tells us that there exists an equilibrium strategy, where the worst-case winnings for each player is maximized. Since they're the worst-case winnings for each player, they're minimum winnings, so we have a minimum maximized – hence the theorem's name. Such an equilibrium strategy might be a randomized strategy. In fact, since randomized algorithms can deliver guaranteed (O(log k)) performance, better than any deterministic algorithm, we might suppose that the maximum worst-case winnings for the adversary are at most (O(log k)). Similarly, the adversary will likely want to play some manner of randomized input sequence, since otherwise there would be added structure for a cache to possibly extract.

Still on tangent, note that if the algorithm designer is committed to a randomized algorithm, there may be no reason to play a randomized input sequence. This is a consequence of the second part of the Minimax Theorem (which, sadly, is not as well-known): if one player is committed to a strategy, there is an optimal, deterministic response, which attains results at least as good as those from the equilibrium strategy. In particular, if the randomized algorithm being used is well-known, there must be a sequence of inputs that has the most expected cache misses; but this can't take longer than with a randomized input sequence (otherwise, we would have chosen this deterministic sequence as our "randomized" one). But we can turn this around: if the input sequence is pre-chosen, there is an optimal deterministic response. This option better describes the usual human user, who will not actively try to thwart the Dropbox caching algorithm, but simply accesses files in a normal fashion. In this case, the sequence of files is random and pre-determined, so there is an optimal deterministic response. And the expected number of cache misses from such is at most (O(log k)). So a good deterministic algorithm, while it has a worst-case competitiveness of (O(k)),

may have an expected competitiveness of at most (O(log k)). And, in fact, LRU is one of these good deterministic algorithms.

Another way to convince yourself that the (k)-competitiveness of LRU is not that bad is compare an LRU cache not with an optimal cache of the same size, but with an optimal but smaller cache. In this case, you can prove a better result. For example, an LRU cache is at most twice as bad as an optimal cache half its size. Compared to an optimal cache of 100 files, an LRU cache for 200 files is at most twice as bad.

Overall, the caching algorithm you want to use is usually LRU, since it is theoretically very good and in practice both simple and efficient. For example, the Dropbox iOS and Android clients both use LRU caches. The Linux kernel uses a variant called segmented LRU.

On to some code.

Caching Algorithms in Practice

Our LRU implementation needs to do two things quickly. It needs to access each cached page quickly, and it needs to know which files are most and least recent. The lookup suggests a hash table, maintaining recency suggests a linked list; then each step can be done in constant time. A hash table can point to its file's node in the list, which we can then go ahead and move around. Here goes.

```
class DoubleLinkedNode:
    def __init__(self, prev, key, item, next):
        self.prev = prev
        self.key = key
        self.item = item
        self.next = next

class LRUCache:
    """ An LRU cache of a given size caching calls to a given function """

    def __init__(self, size, if_missing):
```

```
        """
        Create an LRUCache given a size and a function to call for missing keys
        """

        self.size = size
        self.slow_lookup = if_missing
        self.hash = {}
        self.list_front = None
        self.list_end = None
    def get(self, key):
        """ Get the value associated with a certain key from the cache """

        if key in self.hash:
            return self.from_cache(key)
        else:
            new_item = self.slow_lookup(key)

            if len(self.hash) >= self.size:
                self.kick_item()

            self.insert(key, new_item)
            return new_item
```

To look up an item that's already in the cache, we just need to move its node in the list to the front of the list.

```
    def from_cache(self, key):
        """ Look up a key known to be in the cache. """

        node = self.hash[key]
        assert node.key == key, "Node for LRU key has different key"

        if node.prev is None:
            # it's already in front
            pass
        else:
            # Link the nodes around it to each other
            node.prev.next = node.next
            if node.next is not None:
```

```
            node.next.prev = node.prev
        else: # Node was at the list_end
            self.list_end = node.prev

        # Link the node to the front
        node.next = self.list_front
        self.list_front.prev = node
        node.prev = None
        self.list_front = node

    return node.item
```

To kick an item, we need only take the node at the end of the list (the one that's least recently used) and remove it.

```
def kick_item(self):
    """ Kick an item from the cache, making room for a new item """

    last = self.list_end
    if last is None: # Same error as [].pop()
        raise IndexError("Can't kick item from empty cache")

    # Unlink from list
    self.list_end = last.prev
    if last.prev is not None:
        last.prev.next = None

    # Delete from hash table
    del self.hash[last.key]
    last.prev = last.next = None # For GC purposes
```

Finally, to add an item, we can just link it to the front of the list and add it to the hash table.

```
def insert_item(self, key, item):
    node = DoublyLinkedNode(None, key, item, None)
```

```
        # Link node into place
        node.next = self.list_front
        if self.list_front is not None:
            self.list_front.prev = node
        self.list_front = node

        # Add to hash table
        self.hash[key] = node
```

There it is, a working (k)-competitive LRU cache.

You'll note that we've been assuming so far that all files are the same size. But in practice, this is of course untrue. How do we deal with bigger and smaller files? Well, it turns out, Dropbox naturally subdivides files into blocks (4MB big for the client). So instead of caching particular files, we can cache particular blocks, which are close enough in size that the Least Recently Used algorithm above works. Equivalently, we just kick out files until there is enough room for whatever file we want to load.

Another problem that a real-world cache needs to solve is the issue of cache invalidation – that is, since the files we are caching can change on the server, how do we tell that our cache is out of date? A simple way is to always download an index, which tells you the file's revision number, but not the file data itself. You can do this on a per-directory basis, so that it's not too much data by itself. Then every time you find a file in the cache, you simply check when your copy was last modified and when the server's copy was last modified. This lets you know whether to renew your copy. Going even further, you can cache these indices for each directory, and use the same logic to determine whether they need to be downloaded again. This is what the Android and iOS clients do.

Conclusions

Caches can be used in front of any slow part of your application -- communication over a network, reads from disk, or time-intensive computation. Caching is especially important in mobile

programs, where network communication is often both necessary and costly, so it's good to know the theory and do it right. Luckily, the best solution for caching problems is usually the Least Recently Used algorithm, which is both efficient and simple to implement.

Thanks to Dan Wheeler, Tido the Great, Aston Motes, Albert Ni, Jon Ying, and Rian Hunter for proofreading.

Footnotes:

[2] S. Bansal & D. Modha (2004), "CAR: Clock with Adaptive Replacement".

[3] A. Fiat, R. Karp, M. Luby, M. McGeoch, D. Sleator & N. Young (1991), "Competitive paging algorithms".

Filed under: Performance