16. State-Aware Resource Management

In this chapter we will take a look at scheduling strategies for resources that are stateful.

16.1 State Sensitive Resources

In the previous chapters we have made a fundamental assumption about our resources: the time it takes to serve a request is *independent* from the history of requests served in the past. In practice, almost no resource is stateless, but for practical purposes a resource can be considered stateless if the impact of past requests on the time it takes to service a new request is negligible, and/or constant. For instance, the CPU is considered a stateless resource because the time to perform context-switch/scheduling in between jobs is typically almost constant and much smaller than the length of jobs.

In this lecture, we will instead take a look at resources whose service time is **significantly** impacted by the history of serviced requests. We will take a look at two main examples: (i) a traditional magnetic disk; (ii) the DRAM memory subsystem.

16.1.1 System-centric Metrics

We have seem in the previous lectures that the behavior of the system can be evaluated from a system-centric, or a process-centric perspective. In the context of stateless resources, we have introduced the notion of utilization. Turned out that for these resources, utilization entirely depends on the rate at which requests arrive at the system and the time it takes to process each request. In other words, utilization in stateless resources only depends on supply and demand.

What happens on stateful resources ? In these resources, depending on its state (determined by

previous requests), it may take *a while* to reconfigure the resource to serve the new request. Hence, stateless resources followed the sequence: (i) fetch new request, (ii) serve request, repeat. **Stateful resources** follow the sequence: (i) fetch new request, (ii) reconfigure resource (overhead), (iii) serve request.

It follows that for stateful requests, the utilization of the resource, defined as the time it spends handling requests (reconfiguration + service), also depends on the **order** in which requests are served. Now, an inefficient scheduler, we cause the reconfiguration time to be high. Hence, an inefficient scheduler will make the resource *more utilized*. It follows that a good way to measure the performance of a scheduler for stateful resources is to look at the resulting resource utilization.

Let us make some examples. Imagine that you are a truck driver, who is given a sequence of pick-up and drop-off points. You are supposed to pick-up package A, drop it somewhere, then pick-up package B, drop it somewhere else and so on. Now, you are doing *useful* work while you are transporting a package from its pick-up point to its drop-off point. However, once you have dropped a package, you are in a different *state* that depends on the particular drop-off point. Your state in this case is the location in the city where you are. The reconfiguration time is the time that it takes for you to go from the last drop-off point to the next pick-up point.

Clearly, the perfect scheduler will try to arrange your trips so that your next pick-up is as close as possible to the last drop-off. By doing so, you would be able to transport more packages during the day. In other words, your **capacity** would increase. Conversely, a bad scheduler will make you cross the whole city every time for the next pick-up.

As it turns out, there are always additional dimensions when stateful resource scheduling is considered. For this particular example, one could think about the amount of wasted gas. One could also reason about the induced congestion on the city traffic that would result from having a bunch of trucks going up and down the city all day. Yet another dimension is the depreciation of the trucks as they accumulate wear and mileage faster with a bad scheduler. Or one could add the amount of frustration for the you (the driver) constantly running behind schedule with deliveries, as well as for the customers who see long delivery times.

A further refinement of the scheduling policy would also suggest the most efficient route to go through the city. Just to give an idea of the impact that better schedulers make, consider that UPS "saves several millions of gallons of fuel each year" by making its trucks turn always right (unless a left turn is forced) on their delivery routes [1].

Elevators represent another example of stateful resource. This time, the reconfiguration time is the time it takes for the elevator to travel between the floor where the last person was dropped and the floor for next pickup. In this case, a good scheduler will try to minimize the up/down amount of traveling for the elevator, while making sure that at some point every person will be transported. This brings us to the system's performance from the perspective of a user/process.

[1] http://www.cnn.com/2017/02/16/world/ups-trucks-no-left-turns/index.html

16.1.2 Process-centric Metrics

Be the person who is awaiting a delivery. All you see is the time from when (i) your Amazon's Orders page says that the package is out for delivery, until (ii) the delivery person knocks at your door (and leaves immediately. Yes, that's an optimization too). I.e., we still care about the **average response time**.

Clearly, some schedulers may decide to minimize the reconfiguration time by systematically delay requests that require long resource reconfiguration. Thus, **fairness** is something that needs to be taken into account. For stateless resources, we used the response time of a task to compute the slowdown as a metric of fairness. For a job j_i, the slowdown was: $\frac{R_i}{C_i}$. The problem with stateful resources is that the response time, and hence the slowdown would depend on previously handled requests.

As it will be clear in the next few sections, a key aspect in designing stateful resource schedulers is request re-ordering. Suppose that we have 3 requests arriving in the order: A, then B, then C. After serving A, the resource is in a state such that serving C from there takes much less then serving B next. Hence, B and C are re-ordered for service. In light of this, a measurement of fairness for a generic request Q could be the maximum number of requests that could be re-ordered before Q is served. In other words, we wonder if there is a **bound** on the unfairness to which request Q is subject.

16.2 Hard Disk Scheduling

Consider a traditional magnetic disk, whose structure is depicted in Figure 16.1. For this type of disks, the time to handle each request can be decomposed in the following:

- **Seek delay:** this is the time that it takes for the disk head to align with the cylinder of the disk containing the block needing to be read/written. The larger the diameter of the disk, the longer the seek time. By performing a number of approximations, one can say that the seek time is linear with respect to the distance that the disk head needs to travel. This time is highly dependent on the technology used for the disk head. Modern drives have worst-case seek times of about 20 ms.
- **Rotational delay:** this is the time that it takes for the sector of the disk that contains the target block to rotate until it is under the disk head. If the sector has just passed the head, it may require a full revolution of the disk to bring the sector in the right position for a read/write operation. With standard 7200 RPM (revolutions per minute) disks, the worst-case rotational delay is about 8 ms.
- **Transfer time:** this is the time it takes to read/write the desired block(s) once the head and the disk platter is in the right position. Once again, this depends on the RPM of the disk, as well as the amount of data that needs to be read/written.

If follows that a fundamental dimension to optimize when scheduling disk transactions is the seek time, i.e. how much the disk head should travel before it can start to satisfy the next read/write request. Let us now consider few alternatives.

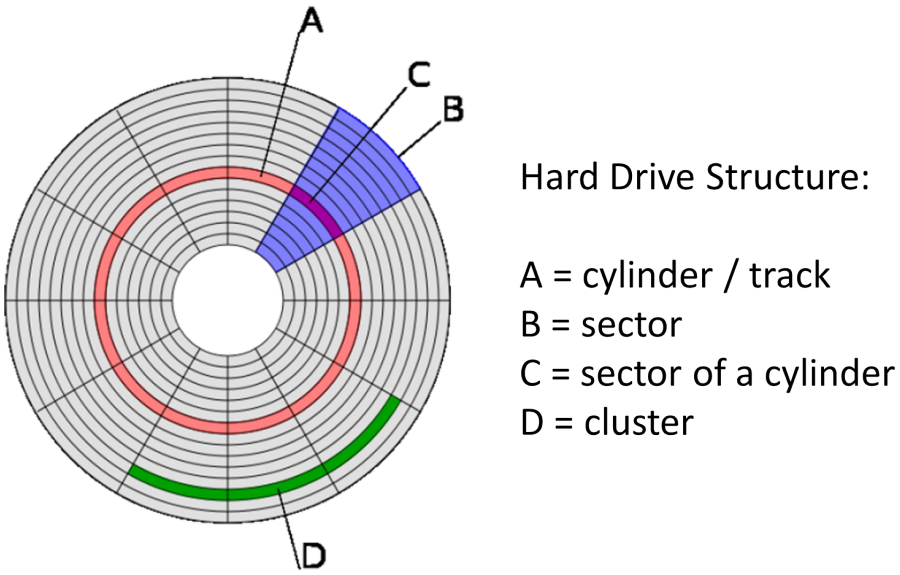

Figure 16.1: Structure of a single platter of a magnetic hard disk.

16.2.1 FCFS Scheduling

In FCFS, disk requests are satisfied in order of arrival. Let us review the key properties:

- **Ease of implementation:** requests are placed in a traditional queue and served one after the other. FCFS is quite simple to implement.
- **Fairness:** there is no chance of request re-ordering, hence fairness is guaranteed.
- **Head movement:** There is nothing in the logic of FCFS that attempts to optimize disk head movement. If requests are spatially independent, the head will move randomly up and down the disk. Typically, a task accesses data that is spatially co-located. In this case FCFS would perform fairly well. But consider what happens if two processes, say A and B, execute submitting interleaved requests. Requests from A $a_1, a_2, \ldots$ are close to each other. Requests from B $b_1, b_2, \ldots$ are also close to each other, but on a cylinder that is far away from the one accessed by A. If the requests are interleaved, we have the FCFS sequence: $a_1, b_1, a_2, b_2, \ldots$. So the disk head is being moved all the way up/down before each new request.

16.2.2 Shortest Scan First – SSF

We have discussed the importance of minimizing disk head movements. SSF attempts at doing just that. In SSF, requests are kept in a queue. Then, the position of the head after the last processed request is considered, and from the queue SSF extracts the pending request for the disk block that is closer to the disk head position.

- **Ease of implementation:** In this case, the geometry of the disk needs to be known. Also, we need to keep track / predict of the position of the disk head after the last read/write request. Next, we need to find out which one of the pending requests addresses a location that is closer

in space on the physical drive.

- **Fairness:** clearly, SSF is unfair as a large burst of spatially co-located requests can delay requests for far-away blocks indefinitely. Consider a block of requests that loop over the same couple of cylinders over and over again. Not only this would delay indefinitely any other request, but it would also cause the disk head to remain "stuck" for long periods of time. This phenomenon goes under the name of "head stickiness", and its occurrence is often correlated to disk malfunctioning.
- **Head movement:** the main point of SSF was to minimize head movement. As it turns out, there are approaches that could achieve even lower head movement than SSF. How ? Here is the intuition. Suppose that you again have the sequence above: a_1, b_1, a_2, b_2. If the arrival of requests is fast enough, then SSF works well. But what if a_2 has not arrived yet by the time a_1 has completed. Then the head would move to serve b_2 (on the other side of the disk), and an ε of time after a_2 arrives. Maybe it would have been better to wait a bit before jumping to process b_2. But oh! This is not a **work-conserving** policy anymore.

16.2.3 SCAN Scheduling

The problem with SSF is that depending on the load you could have the movement of the head entirely localized between few close-by cylinders. The SCAN algorithm, also known as the Elevator algorithm, imposes a constraint on the direction in which the disk head can move.

The SCAN scheduler will pick as next the request that is closer in space to the position of the head, but only in one direction. For instance, suppose that the disk head is currently at position 30, that the current direction is "up", and that the pending requests are for positions: 29, 27, 25, and 40. SCAN would pick the request at 40 to serve next, as opposed to 29, because it is moving up. Once no more requests are available in the "up" direction, SCAN would reverse direction, going "down" instead. Hence the alternative name of Elevator scheduling.

- **Ease of implementation:** the complexity of the implementation is not much different from SSF. We need to keep track of the disk head, and be able to compute/predict the desired head location for each pending request.
- **Fairness:** SCAN solved the problem that requests circulating over a pair of close-by cylinders would monopolize the disk. SCAN forces the head to keep moving up/down. Nonetheless, if the load insists always on the same cylinder, head stickiness is still a possibility.
- **Head movement:** SCAN prevents the head from jumping all over the place on the disk, hence mitigating head reconfiguration overheads.

16.2.4 C-SCAN Scheduling

One of the main problems with SCAN is that if the head has just left cylinder 0, and a new request for cylinder 0 arrives, in the worst-case it would have to wait for a bunch of requests the cover all the cylinders $0 \rightarrow N$, and back $N \rightarrow 0$. Just think about how frustrated you get when you are at floor 1 (out of 10) and you have just missed the elevator. Now, not only the elevator stops at every single floor on its way up, but it also keeps stopping at every single floor on its way down.

C-SCAN addresses this issue by limiting the direction of the head always on one direction only, e.g. "up". Once no more requests are available for higher cylinder numbers, the head resumes from cylinder 0. It is as if once the elevator reaches the 10-th floor, it travels all the way down ignoring requests for people who want to go down.

Other than mitigating the worst-case waiting time for a request, C-SCAN is identical to SCAN.

16.2.5 Complete Fair Queuing Scheduler – CFQ

Ok, so which one of the previous disk scheduling algorithms a real operating system (say Linux) uses ? None of the above. Linux (by default) uses something called Complete Fair Queuing (CFQ) scheduler[2].

The real CFQ algorithm distinguished between *synchronous* requests (requests that need to be satisfied, otherwise the process would block) and *asynchronous* requests (things that can be done later without impacting the application performance). Let us take a look at how CFQ works only for synchronous requests.

The main objective of CFQ is to fairly partition the available disk bandwidth among applications. For this, CFQ works in two stages. In the first stage, CFQ maintains one queue of requests per task/application. Then, it assigns a *quantum of requests* q_{CFQ} (not time!) to be taken from each queue, before moving to the next queue, in a RR fashion. If a queue is empty, or there are less pending requests than the quantum, CFQ will temporarily **wait** on that queue before moving to the next queue.

The requests taken from the per-process queues are not immediately sent to the disk. Instead, they are further queued on a single *dispatch queue*. On this queue, they are re-ordered to minimize disk head movement before (finally) being sent to the disk. Figure 16.2 illustrates the interaction between per-process queues and dispatch queue.

CFQ teaches one important lesson: maybe a best way to achieve disk head movement minimization *and* fairness is to decouple the two things. With CFQ, fairness is handled with a set of round-robin queues (one per application); disk head movement is minimized by re-ordering requests on the dispatch queue. Boom!

- **Ease of implementation:** it is not exactly easy to implement CFQ because apart from having to keep track of the position of the disk head, we have to maintain a set of per-process queues, a dispatch queue, and perform reordering on the latter queue.
- **Fairness:** essentially, once the requests are in the dispatch queue, requests are re-ordered with a C-SCAN-like criteria, i.e. by moving always "up". However, it is not possible for a process to monopolize the disk because requests are accepted into the dispatch queue in RR. So the maximum number of requests that in the worst-case can be re-ordered before a generic request Q is bounded by a factor $q_{CFQ} \cdot N$, where N is the number of tasks with pending requests in the system.
- **Head movement:** CFQ effectively compromises head movement minimization and fairness.

[2] https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt

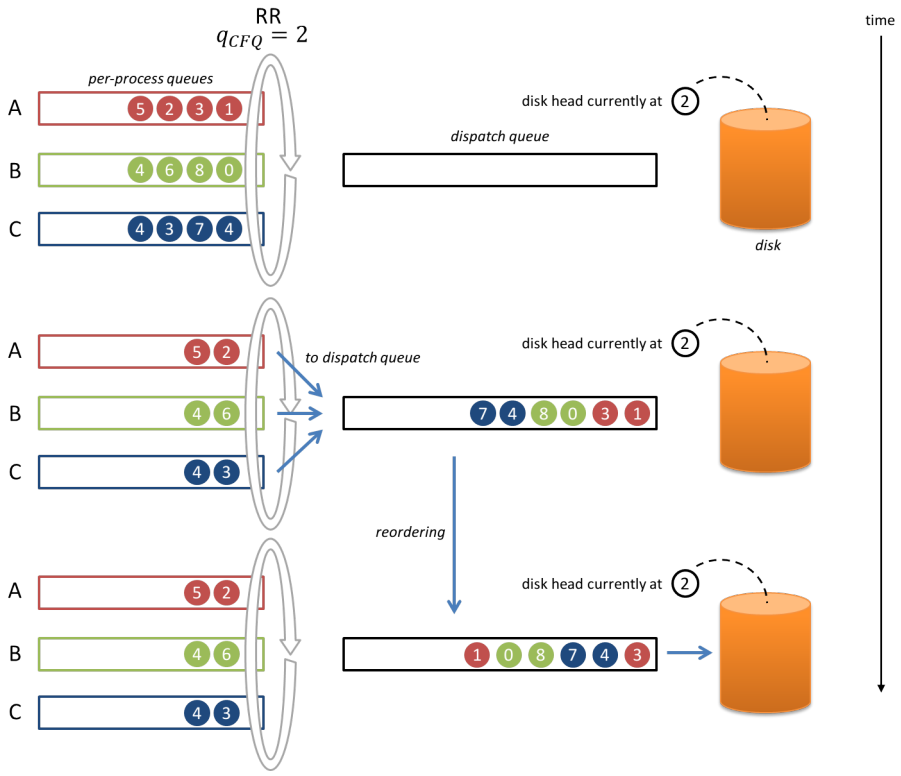

Figure 16.2: CFQ in action: RR among per-process queues (A, B, C) plus reordering on dispatch queue.

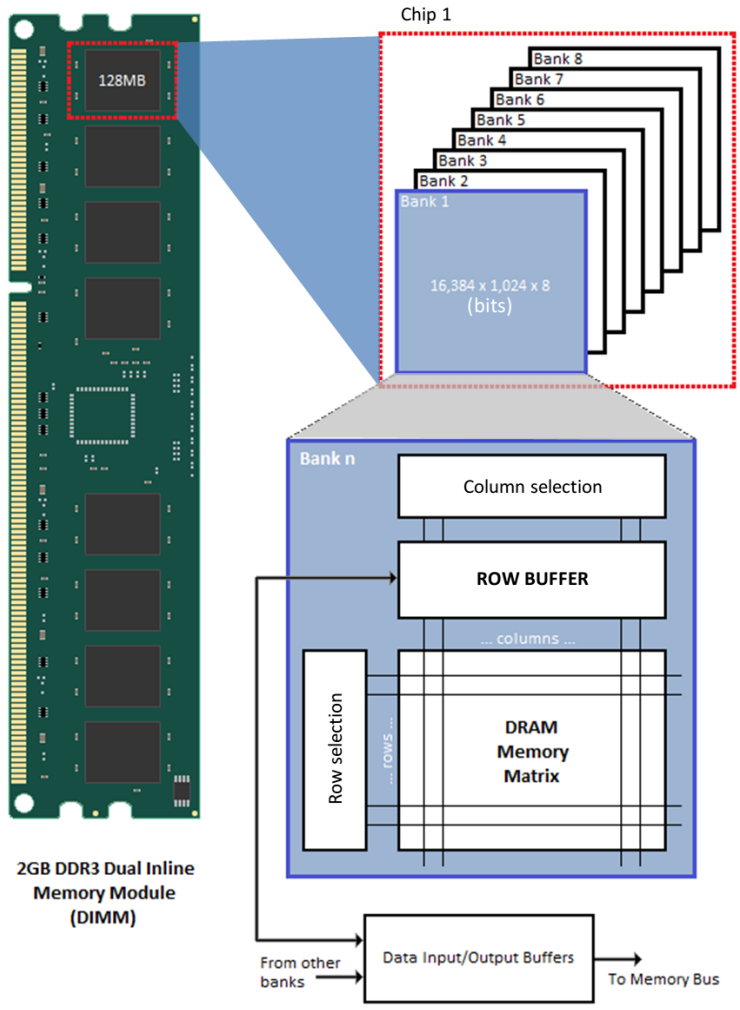

Figure 16.3: Structure of a commercial DRAM memory module.

In other words, it minimizes head movement for requests already in the dispatch queue (not for all the ready requests). Interestingly enough, in the real implementation, if a per-application queue has less ready requests than q_{CFQ}, CFQ *waits* a little bit on that queue before moving to the next, just in case some request arrives. This makes the actual implementation of CFQ **non-work-conserving**.

16.3 The DRAM Subsystem

The typical view of main memory is a uniform space where accesses can be performed randomly and take the same time to satisfy, no matter what was the history of performed memory requests. This is very far from the truth.

When memory is implemented using Dynamic Random Access Memory (DRAM), the organiza-

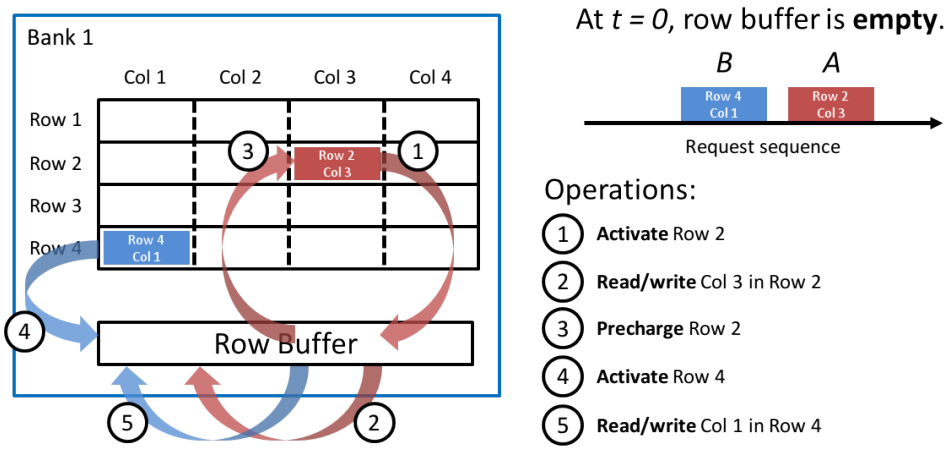

Figure 16.4: Sequence of operations for a DRAM row miss.

tion of memory block is very far from being "uniform". Figure 16.3 depicts the main components of the structure of a commercial DRAM memory module. In its basic formulation, DRAM is divided in a number of *chips*. If you even handled a DRAM module, you have probably noticed the chips soldered onto the printed circuit (left side of the figure). Each chip is internally organized as a group of *banks*. Typically 8 or more. Each bank holds few megabytes of data. The typical size of a bank is 16 MB. Each bank is internally organized as a 2-dimensional array of *rows* and *columns*. The typical number of rows per bank is 16344. The typical number of columns is 1024. So each bank holds 16 MB because: bank total = rows $\times$ columns.

When you want to read a given byte at address A in DRAM, the address needs to be **decoded** in terms of coordinates of the type: $A \rightarrow$ chip X, bank Y, row Z, column Q. While banks can work in parallel, two requests targeting the same bank need to be satisfied in sequence (serialized). The problem is that you cannot read or write a single byte inside a bank, but you can only read/write entire rows. There is a buffer for that. When you request a certain piece of data at address A corresponding to row Z, column Q, the entire row Z is copied onto a *row buffer*. This operation is called **activate**. However, if a row different from the one currently in the row buffer needs to be activated, then the old row needs to be put back in place. This operation is called **pre-charge**. Once row Z is in the row buffer, we can read/write the desired column Q.

If follows that if two consecutive requests A and B target two different rows of the same bank, say Z_A and Z_B, while the row buffer is empty, the resulting sequence of operations would be: (i) activate Z_A, (ii) read/write some column in Z_A, (iii) pre-charge Z_A, (iv) activate Z_B, (v) read/write on Z_B. This situation is depicted in Figure 16.4. Conversely, if A and B were requests targeting the same row, we would have: (i) activate Z_A, (ii) read/write some column in Z_A, (iii) read/write some column in Z_B. The former case is called a case of *row miss*, while the latter is a case of *row hit*. As it turns out, the time required to satisfy a request in case of a row miss is **10 times** more than the time for a row hit.

It makes a lot of sense to schedule memory requests trying to ordering them so to maximize the number of row hits, as this would dramatically improve the capacity of the DRAM subsystem

(which is typically the bottleneck in computing systems). It would be impossible however to perform scheduling at the OS. For this reason, modern DRAM controllers embed a transaction scheduler in hardware.

16.3.1 First Ready, First Come First Served – FR-FCFS

The hardware scheduler in a DRAM controller implements a First Ready, First Come First Served (FR-FCFS) scheduling policy. The idea is simple: a different queue is defined for each bank. Memory requests are queued in each queue depending on the targeted bank. The FR-FCFS keeps track of which one is the row currently in the row buffer. With this piece of information, it re-orders requests in the bank queue so to prioritize requests that will result in row hits. Apart from this re-arrangement, requests are handled in their order of arrival (FCFS).

- **Ease of implementation:** fairly easy to implement (in hardware). All we need is to keep track of the currently open row. We also need the capability to re-order the elements in a queue.
- **Fairness:** this way of doing things could lead to starvation: think about what would happen if an application keeps requesting data in the same row over and over again. For this reason, the FR-FCFS scheduler imposes a bound of re-ordering. So FR-FCFS is not fair to applications that have poor locality in their accesses.
- **Memory throughput:** clearly FR-FCFS aims at optimizing the throughput by processing in bursts requests for the same row.

16.3.2 Capped First Ready, First Come First Served – FR-FCFS+Cap

In order to solve the problem of fairness in FR-FCFS, a cap on reordering is introduced, call it K. The idea is to ensure that no more than K younger row-hits can be re-ordered (effectively jump the line) before older requests.

- **Ease of implementation:** this is trickier to implement because now we also have to keep track of how much re-ordering a row-miss request has been suffering.
- **Fairness:** clearly with the introduction of the cap K, we are imposing a maximum bound on the unfairness that a row-miss request can suffer in the worst-case.
- **Memory throughput:** FR-FCFS+Cap represents a compromise in terms of fairness and throughput optimization. In practice, its performance in terms of throughput are comparable with respect to FR-FCFS.

FR-FCFS+Cap is the typical request scheduler implemented in commercial DRAM controllers. Of course the real implementation introduces a number of additional small improvements and differential treatment for read vs. write requests, but the main idea is just what has been described.