#uni/semester3/Betriebssysteme/chapter22/chapter-notes

➜ When little memory is free, things get interesting. This **memory pressure** forces the OS to start paging out pages. The **replacement policy** decides which page to evict.

1. Cache Management

   ➜ Memory is also known as a cache for virtual memory pages because it holds some subset of all the pages in the system. Goal of the replacement policy is to minimise the number of cache misses (minimise the number of times we have to fetch a page from disk)

   ➜ If we know the number of cache hits and misses we cal calculate the **average memory access time (AMAT)** for a program

$$AMAT = T_M + (P_{Miss} \cdot T_D)$$

   ➜ Tm represents the cost of accessing memory. Td the cost of accessing the disk. Pmiss the probability of a miss. You always have to pay the cost of accessing memory, but you can additionally pay the cost of accessing the disk.

   ➜ If we had the sequence hit, hit, hit, miss, hit, hit, hit, hit, hit, hit. The hit rate would be 0.9, and miss rate = 0.1

   ➜ Going from miss rate of 0.1 to 0.001, makes it 100 times faster. As the hit rate approaches 100% AMAT approaches the cost of accessing memory. As the miss rate gets bigger, it approaches the cost of accessing the disk.

2. The Optimal Replacement Policy

   ➜ Belady developed the optimal replacement policy that leads to the fewest number of misses. The optimal policy is to replace the page that will be accessed furthest in the future. This is simple but difficult to implement.

   ➜ So basically throw the page the one which is needed furthest from now.

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	3	0, 1, 2
1	Hit		0, 1, 2

Figure 22.1: **Tracing The Optimal Policy**

➜ Assumption, we have a cache with only 3 entries, so with 4 pages we need to replace one at one point

➜ the first 3 accesses are misses and such a miss is called cold-start miss (or compulsory miss)

➜ hit rate is: 6 / (6 + 5) = 54.5%

➜ or ignore the compulsory misses (ignore the first miss to a given page) (hit rate modulo compulsory misses). 6 / (6 + 1) = 85.7%

➜ Future is not generally known, so we can't build the optimal policy. The optimal policy will only serve us a comparison point, to know how close we are to perfect.

➜ There are 3 types of misses: compulsory, capacity and conflict.

➜ Compulsory is when cache is empty and this is the first reference to the item.

➜ Capacity miss is when the cache ran out of space and we had to evict one item.

➜ Conflict miss arises in hardware because if limits where an item can be placed in hardware cache, due to set associativity. It does not arise in the OS page cache because such caches are fully associative, meaning there are no restrictions where in memory a page can be placed.

3. A Simple Policy: FIFO

➜ FIFO has one strength, it is quite simple to implement. The pages are placed in a queue and when replacement occcurs, the page on the tail of the queue is evicted.

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		First-in→ 0
1	Miss		First-in→ 0, 1
2	Miss		First-in→ 0, 1, 2
0	Hit		First-in→ 0, 1, 2
1	Hit		First-in→ 0, 1, 2
3	Miss	0	First-in→ 1, 2, 3
0	Miss	1	First-in→ 2, 3, 0
3	Hit		First-in→ 2, 3, 0
1	Miss	2	First-in→ 3, 0, 1
2	Miss	3	First-in→ 0, 1, 2
1	Hit		First-in→ 0, 1, 2

Figure 22.2: **Tracing The FIFO Policy**

➜ It has a hit rate of 36.4% (or 57.1% excluding compulsory misses)

➜ FIFO can't determine the importance of blocks

➜ One interesting reference stream for FIFO 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

➜ When you increase the cache size from 3 to 4, the hit rate doesn't get better (as someone would expect) but worse! This odd behaviour is called Belady's Anomaly.

➜ LRU doesn't have this problem, because it has a stack property. For algorithms with this property, a cache of size N + 1 naturally includes the contents of a cache of size N. When increasing the cache size, hit rate will either stay the same or get better. FIFO and Random do not obey the cache property and thus have some anomalous behaviour.

Frame	0	1	2	3	0	1	4	0	1	2	3	4
0	0	0	0	3	3	3	4	4	4	4	4	4
1		1	1	1	0	0	0	0	0	2	2	2
2			2	2	2	1	1	1	1	1	3	3

Frame	0	1	2	3	0	1	4	0	1	2	3	4
0	0	0	0	0	0	0	4	4	4	4	3	3
1		1	1	1	1	1	1	0	0	0	0	4
2			2	2	2	2	2	2	1	1	1	1
3				3	3	3	3	3	3	2	2	2

➜ weird behaviour of FIFO when increasing cache size to 4

4. Another Simple Policy: Random

   ➜ Random is similar to FIFO. Easy to implement but not trying to be too intelligent in picking which page to evict.

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	0	1, 2, 3
0	Miss	1	2, 3, 0
3	Hit		2, 3, 0
1	Miss	3	2, 0, 1
2	Hit		2, 0, 1
1	Hit		2, 0, 1

Figure 22.3: **Tracing The Random Policy**

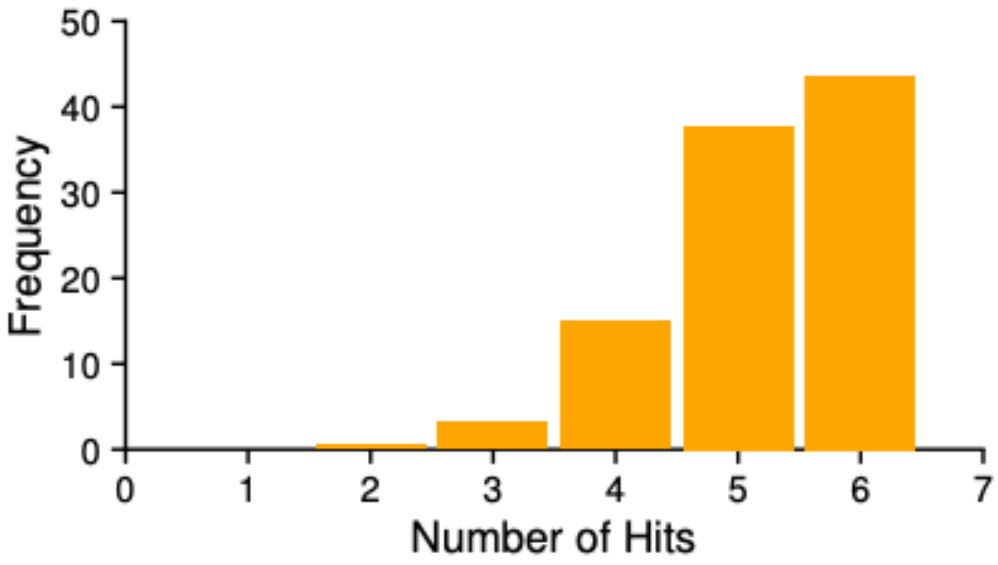


Figure 22.4: **Random Performance Over 10,000 Trials**

➜ If we run Random for 10000 with the above stream, we see that in about 40% of time Random is as good as optimal with 6 hits. Sometimes it is much worse with only 2 hits. How Random does depends on luck.

5. Using History: LRU

➜ FIFO or Random might throw out an important page

➜ as we did with scheduling policy, in order to improve the future we need to lean on the past and use it as our guide. If a program has accesses a page in the near past, it might access it again in the near future.

➜ One type of historical information in order do decide how important page is, is its frequency.

➜ Another one is the recency of an access, because the more recently a page has been accessed, perhaps the more likely it will be accessed again.

➜ This family of policies is based on the principle of locality. Thus we should use history to figure out which pages are important and keep those pages in memory when it comes to eviction time.

➜ There is also Least-Frequently-Used (LFU) policy.

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		LRU→ 0
1	Miss		LRU→ 0, 1
2	Miss		LRU→ 0, 1, 2
0	Hit		LRU→ 1, 2, 0
1	Hit		LRU→ 2, 0, 1
3	Miss	2	LRU→ 0, 1, 3
0	Hit		LRU→ 1, 3, 0
3	Hit		LRU→ 1, 0, 3
1	Hit		LRU→ 0, 3, 1
2	Miss	0	LRU→ 3, 1, 2
1	Hit		LRU→ 3, 2, 1

Figure 22.5: **Tracing The LRU Policy**

➜ LRU uses history to do better than stateless policies.

➜ There are also opposites of these algorithms. Most-Frequently-Used (MFU) and Most-Recently-Used (MRU).

➜ Spatial locality means if a page P is accessed, it is likely that the pages around it (P - 1 or P + 1)

➜ Temporal locality states that pages which have been recently accesses, will be accessed in the near future.

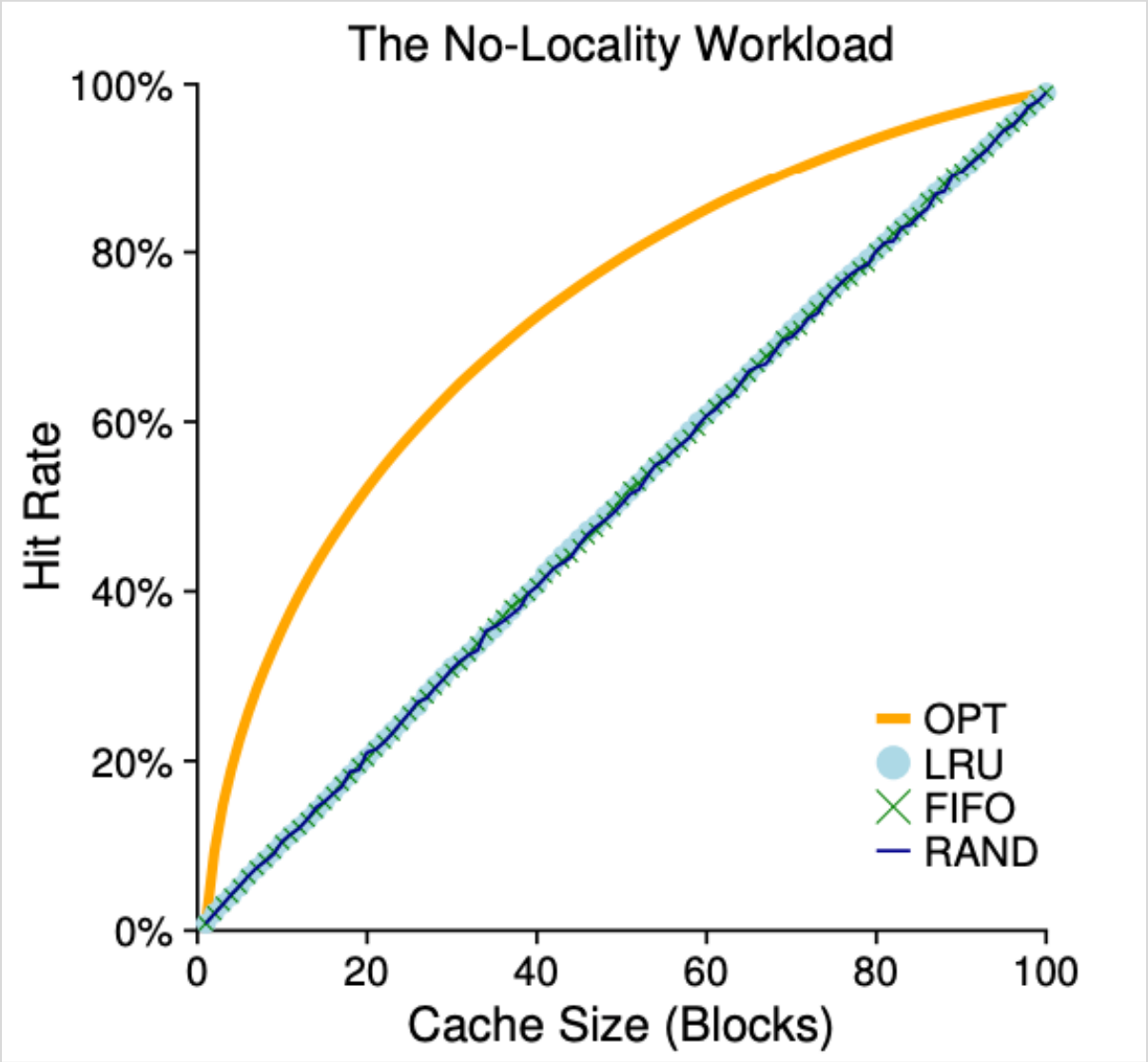
➜ The assumptions of these properties play a big role in how the OS, thus helping programs which obey to this properties, to run fast.

➜ This **principle of locality** is not a hard rule that all programs must obey. Some indeed access memory in a random fashion.

6. Workload Examples

➜ next more complex **work loads** are being analysed

➜ First one has no locality. Each reference is to a random page. It accesses 100 unique pages; overall 10 000 pages are accessed



➜ When there is no locality in the workload, all policies perform the same, with performance increasing by the page size. When cache size is as big to fit the entire workload, all policies get 100% hit rate

➜ The next workload is called "80-20". 80% of the references are made to 20% of the pages (hot pages) and 20% of the references are made to the 80% of the pages (cold pages). There are in total 100 unique pages.

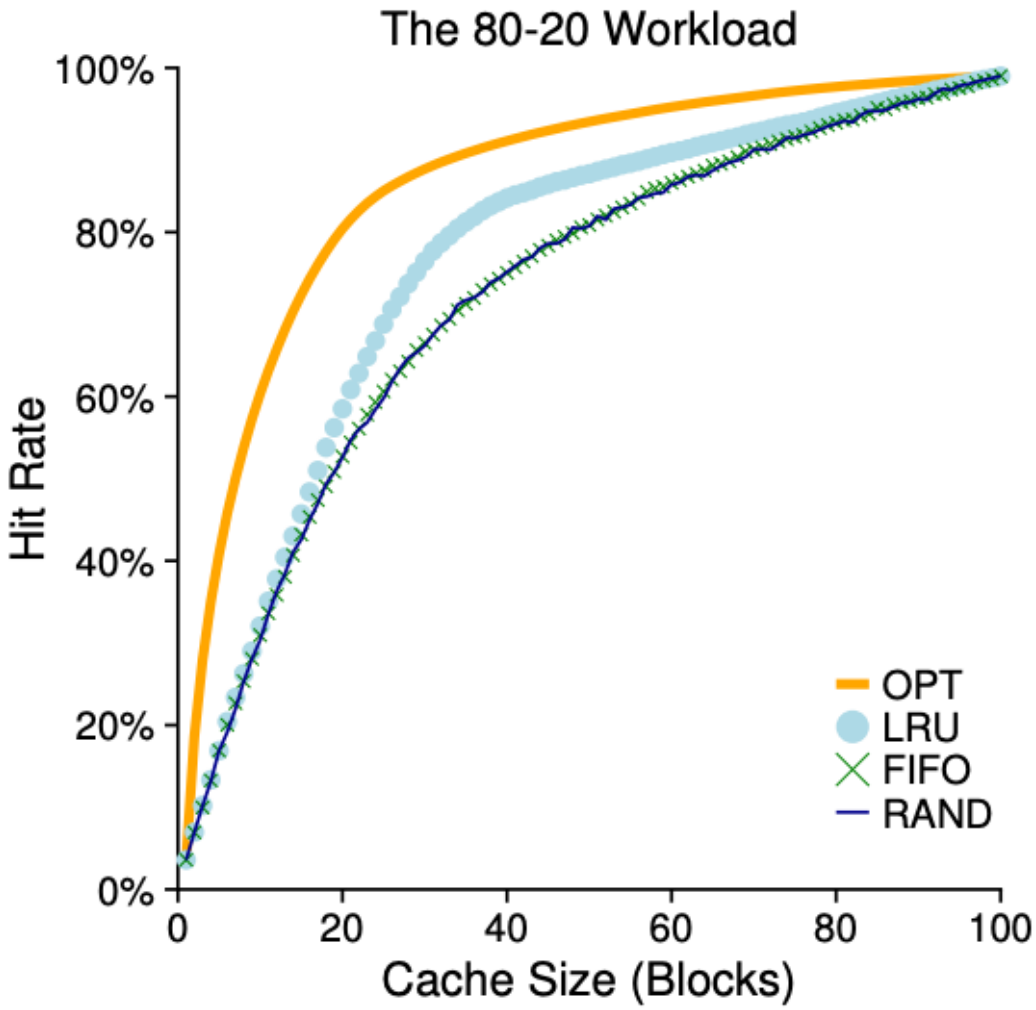


Figure 22.7: **The 80-20 Workload**

➜ LRU does better than RAND and FIFO because it holds onto the hot pages. They have been referred frequently in the past, they are likely to be referred in the future.

➜ Hit rate improvement is very important, when the cost of each miss is very expensive. If it isn't that expensive, then the improvement also isn't that important.

➜ Final workload is a looping sequence. We refer to 50 pages in sequence, starting at 0 to page 49. Then we loop repeating those accesses for a total of 10 000 accesses to 50 unique pages.

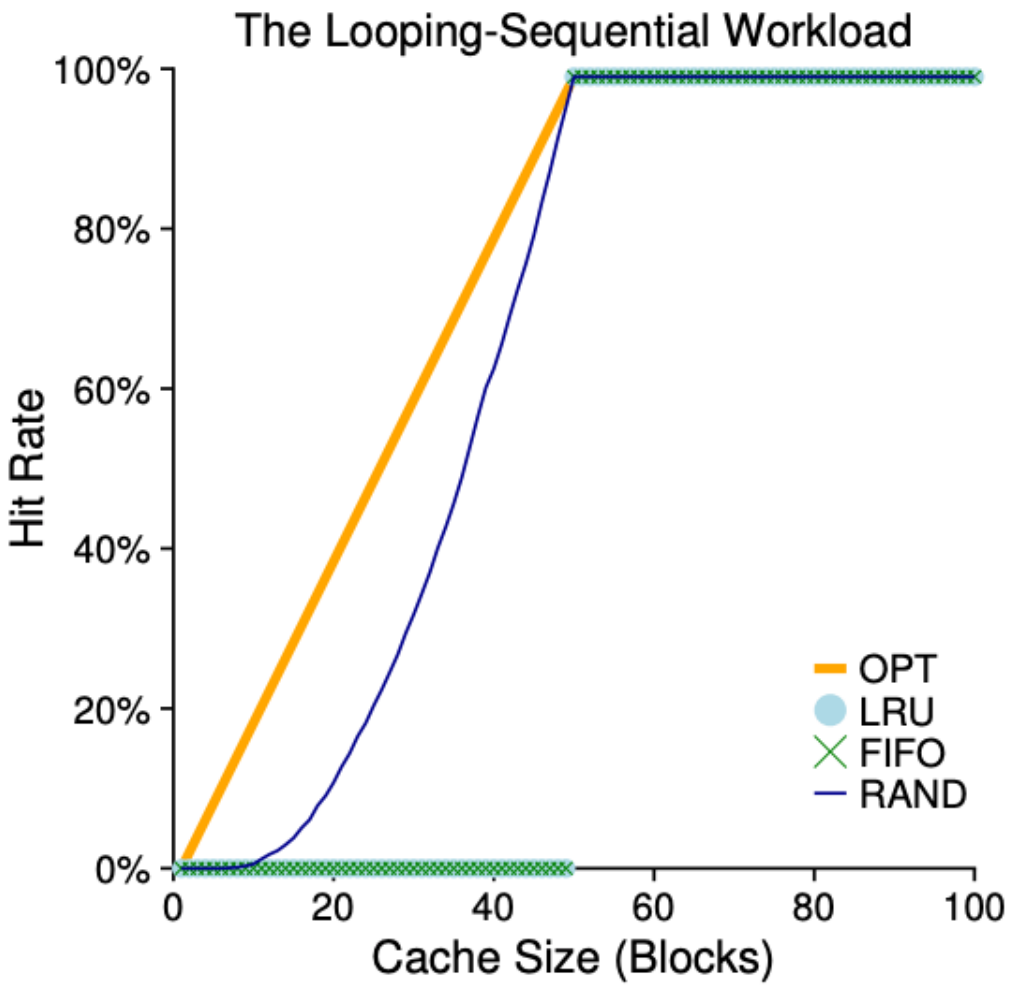


Figure 22.8: **The Looping Workload**

➜ This is a worst case for LRU and FIFO. They both kick out older pages, and because of the looping, these will are going to be accessed sooner than the pages the policies prefer to keep in cache. Random performs much better, but not quite approaching the optimal. Random has the nice property of avoiding weird corner case behaviours.

7. Implementing Historical Algorithms

➜ How to implement LRU? We need to do a lot of work, because on each page access we must update some data structure to move this page to the front of the list (i.e. MRU policy). For FIFO you only need to access the list of pages when a page is evicted or when a new page is added to the list. With LRU the system has to do some accounting work on every memory reference. This greatly reduces performance.

➜ To speed this up we could use a bit of hardware help. The machine could update on each page access a time field in memory (maybe in the per process page table or in some array with one entry per physical page of the system). Thus when page accessed hardware sets the time field to current time. When replacing a page, the OS scans all the time fields to find the LRU page.

➜ This would be expensive with a lot of pages, which is the case in modern system. Thus, do we really have to find the absolut oldest page to replace, or instead survive with an approximation?

8. Approximating LRU

➜ This idea requires some hardware support in the form of a use bit (referenced bit). This is one use bit per page of the system and its location is in the per process page table or in an array somewhere.

➜ When a page is referenced (read or written) the use bit is set by the hardware to 1. The hardware never clears the bit, this is the responsibility of the OS.

➜ The clock algorithm is a simple approach. All the pages of the system are arranged in a circular list. A clock hand points to some particular page to begin with. When a replacement occurs, the OS looks at the currently pointed page P if it has a use of 1 or 0. If 1 then page was recently used and clears it to 0. The hand clock is incremented and points to the next page P + 1. This algorithm continues, till a page of use bit 0 has been found. Worst case is when going through the whole list, and set all bits to 0.

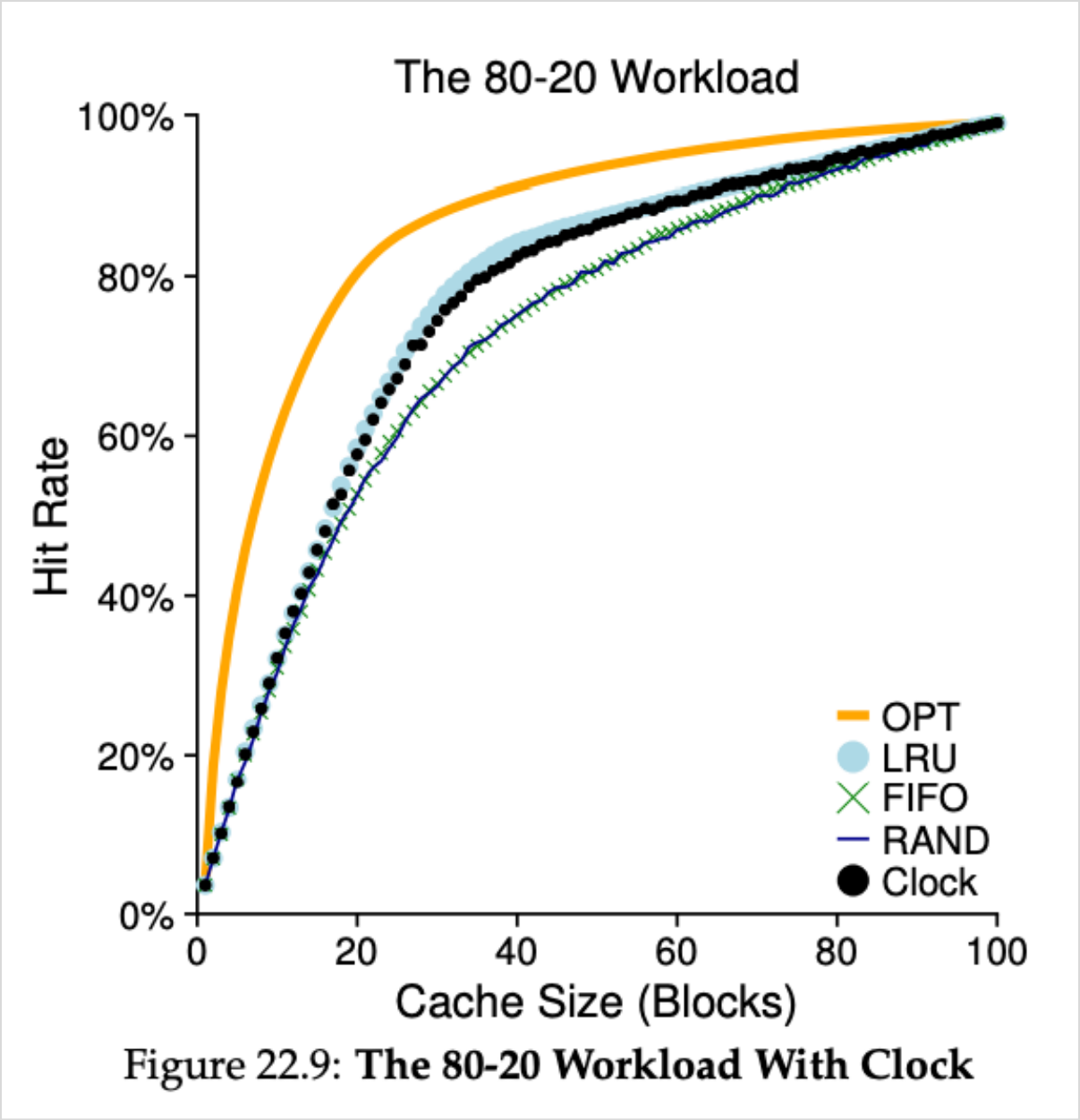


Figure 22.9: **The 80-20 Workload With Clock**

➜ Clock isn't as good as perfect LRU, but still better than RAND and FIFO.

9. Considering Dirty Pages

   ➜ Modification of the clock algorithm, is the additional consideration fo whether a page has been modified or not while in memory

   ➜ If a page has been **modified** and thus being **dirty** it must be written back to disk to evict it which is expensive. If page isn't dirty then is **clean** which means the eviction is free. The physical frame can simply be reused without additional I/O. Thus we want to evict clean pages over dirty ones.

   ➜ Hardware support in this is the **modified bit** or **dirty bit**, which is set any time a page is written. The clock algorithm can now scan for pages that are both unused and clean to evict first. If there aren't any, then search for unused pages that are dirty, and so forth.

10. Other VM Policies

➜ Page replacement isn't the only policy. The OS also has to decide when to bring a page into memory. This policy is called **page selection policy**.
➜ **Demand paging** is when the OS brings the page into memory, when it is accessed.
➜ **Prefetching** is when OS guesses that a page is about to be used and should be only done when the chance of success is reasonable. If a code page P is brought into memory then it assumes P + 1 will also be accessed soon and should be brought into memory too.
➜ Another policy determines how to write pages out to disk. One at a time or collect many pending write in memory and write them to disk at once. This is called **clustering** or **grouping** of writes. This is effective because disks perform a single write more efficiently.

11. Thrashing

➜ **Thrashing** is a condition, when the system is constantly paging. This happens because the memory demands of the processes exceeds the available physical memory.
➜ To cope with this condition there is **admission control** which given a set of processes, a system could decide not to run a subset of process in order to make little progress. It is better to make a little progress then do many things at once poorly.
➜ Another approach is running an **out of memory killer** which is used by some versions of Linux. This once chooses a memory intensive process and kills it. This approach can have problems because it can kill something, which other things depend on, thus leading to instability in applications.

12. Summary

➜ Modern systems tweak the LRU approximation with **scan resistance** algorithms such as ARC. These algorithms which is like LRU but try to avoid the worst case behaviour of LRU, which we saw with the loop sequence.
➜ Because of the gap speed between RAM and disk the importance of these algorithms has decreased, because paging disk is always too expensive. So the best solution to excessive paging is to buy more memory.