Chapter 22 Beyond Physical Memory: Policies

Memory pressure — when little free memory

- Forces the OS to start paging out pages
  - Make room for actively-used pages
- Deciding which pages to evict: **replacement policy**

22.1 Cache Management

We can view main memory (PM) as a **cache** for VM pages in the system

Goal of replacement policy: *minimize cache misses*

- That is, minimize number of times we fetch a page from disk
- Or: vice versa: maximize cache hits

Knowing cache hits and misses let us calculate:

Average memory access time (AMAT):

$$AMAT = T_M + (P_{miss} * T_D)$$

T_M = cost of accessing memory
T_D = cost of accessing disk
P_{miss} = probability of not finding the data in cache

Note: *always pay cost of accessing data in memory* (T_M).

Example. Assume T_M = 100 ns, T_D = 10 ms, P_{miss} = 10% = 0.1

AMAT = 100 ns + (0.1 * 10 ms) = 100 ns + 1 ms = 1.0001 ms ≈ 1 ms

Example. If the hit rate 99.9 %: P_{miss} = 0.001

AMAT = 100 ns + 0.001*10 ms = 100 ns + 0.01 ms = 10.1 microseconds

100x faster: with near 100 % hit rate: AMAT approaches 100 ns

22.2 The Optimal Replacement Policy

Optimal policy: MIN - fewest misses overall

- Replace the page that will be accessed furthest in the future
- Not possible to implement — can’t predict the future
- Still useful to know what the optimal would for a set of cases
  - Can compare your new replacement policy algorithm with the optimal for certain workloads

Example. Cache size = 3, #Pages = 4.

- Stream of virtual page references:
  - 0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1

Cold start:

- Filling the cache — first three misses

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**

Q: Which page to replace/evict to load page 3?

- 0 used immediately after
- 3 used again
- 1 used next
- 2 used next (2 must be evicted since there is no room for it, and it is used furthest into the future)

Q: Which page to evict to load page 2?

- 1 used immediately
- No more pages used: so can evict page 3 or 0. We chose 3 in this example.

Hit rate for the cache:

Hit rate = Hits / (Hits+Misses)

Example. 6 hits, 5 misses:

Hit rate = 6 / (6+5) = 6/11 = 54.5 %

Hit rate modulo compulsory misses:

Hit rate = Hits / (Hits+Misses-#Pages)

Example. There are 4 pages in total. Ignore four misses due to first load of each page:

Hit rate = 6 / (6+5-4) = 6/7 = 85.7 %

22.3 Simple Policy: FIFO

- Pages placed in a queue when entering the system
- Replacement:
  - Evict page on the tail of the queue (the first-in page)

Which page to evict to load page 3?

- Easy: page 0 since it was loaded first

Which page to evict for page 0?

- Easy: page 1 since it was loaded after page 0

Compare FIFO to optimal policy (MIN):

Hit rate = 4/(4+7) = 4/11 = 36.4 % (54.5 % for MIN)

Hit rate excluding compulsory misses = 4/(4+7-4) = 4/7 = 57.1 % (85.7 % for MIN)

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**

Aside: Belady's Anomaly

In general: *expect cache hit rate to increase (get better) when caches gets larger.*

Example. Memory-reference stream: 1, 2, 3, 4, 1, 2, 5, 1, 3, 4, 5.

Cache Size = 3					Cache Size = 4			
Access	**Hit/Miss?**	**Evict**	**Cache State**		**Access**	**Hit/Miss?**	**Evict**	**Cache State**
1	Miss		1		1	Miss		1
2	Miss		1,2		2	Miss		1,2
3	Miss		1,2,3		3	Miss		1,2,3
4	Miss	1	2,3,4		4	Miss		1,2,3,4
1	Miss	2	3,4,1		1	Hit		1,2,3,4
2	Miss	3	4,1,2		2	Hit		1,2,3,4
5	Miss	4	1,2,5		5	Miss	1	2,3,4,5
1	Hit		1,2,5		1	Miss	2	3,4,5,1
2	Hit		1,2,5		2	Miss	3	4,5,1,2
3	Miss	1	2,5,3		3	Miss	4	5,1,2,3
4	Miss	2	5,3,4		4	Miss	5	1,2,3,4
5	Hit		5,3,4		5	Miss	1	2,3,4,5

With 3 pages:

Hit rate = 3 / (3+9) = 25 %

With 4 pages:

Hit rate = 2 / (2+10) = 16.7 %

FIFO may sometimes get worst with larger cache sizes.

22.5 Using History: LRU

Use history as guide:

- If page access in near past — likely to be accessed again in near future
- Historical info
  - Frequency — page access many times (page must have “value”; let’s keep it in cache)
  - Recency — page was recently accessed

Family of policies: rely on **principle of locality**

- Programs tend to access code/data in sequence, e.g., in a loop
- Keep those pages in memory instead of evicting them
- Heuristic: *often very good*, but may exhibit random access that limit the use of the locality principle!

Least-Frequently-Used (LFU)

- Replaces least-frequently used page

Least-Recently-Used (LRU)

- Replaces least-recently used page

Example. LRU Fig 22.5

Hit rate = 6/(6+5) = 6/11 = 54.5 % (same as MIN)

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**

22.6 Workload Examples

In practice: need to study more complex **workloads**.

- ideally: application traces

Example. Workload without locality

- Each reference is a random page (within the set of accessed pages)
  - Workload accessed 100 unique pages
  - 10,000 pages accessed overall
- Varied the cache size: 1 page - 100 pages

Fig 22.6 shows:

- When no locality in workload
  - Doesn't matter much which policy you use
  - All perform the same
- When cache is large enough to fit entire workload
  - Doesn't matter which policy is used
  - All policies converge to 100 % hit rate
- Optimal performs much better than any realistic policies
  - Peeking into the future does a much better job of replacement

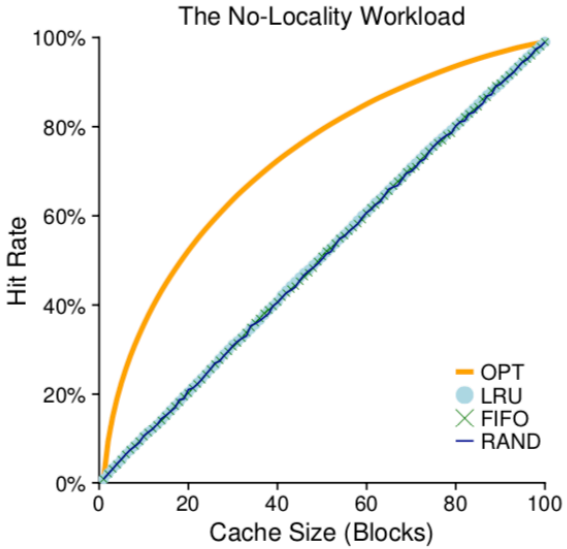

Figure 22.6: **The No-Locality Workload**

Example. 80-20 Workload

- Again: 100 unique pages
- Exhibit locality:
  - 80 % of references are made to 20 % of the pages (hot pages)
- Remaining 20 % of references are made to 80 % of the pages (cold pages)

Fig 22.7 shows:

- FIFO and random does reasonably well
- LRU does better: more likely to hold on to hot pages
- Optimal is quite a bit better than LRU
  - Showing that LRU's historical info is not perfect!

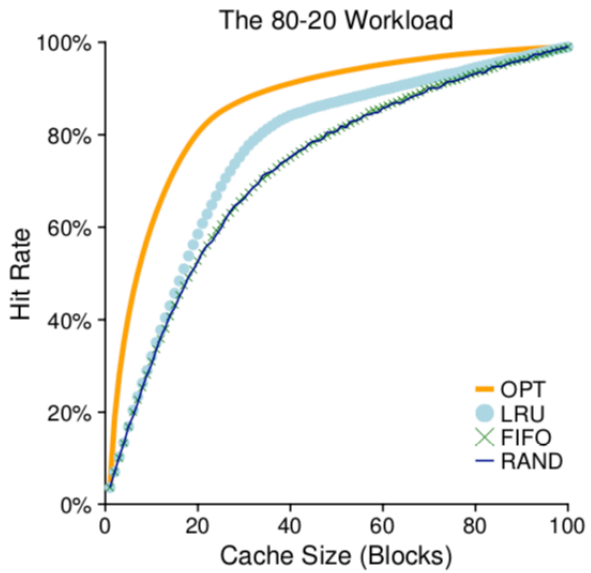

Figure 22.7: **The 80-20 Workload**

Example. Looping Sequential Workload

- 50 unique pages in the hot loop
  - Refer to 50 pages in sequence: 0, 1, 2, … 49 and
  - Then loop for 10,000 accesses
- Common in many applications (e.g. databases)
  - Worst-case for LRU and FIFO
  - Kick out older pages
    - But due to the looping nature of the program
    - Those are the “older” pages that will soon be accessed again
  - Cache size of 49 — with 50 pages in a looping workload:
    - Hit rate of 0 %
  - Random much better for this case, but not as good as optimal
    - Random doesn’t have corner cases

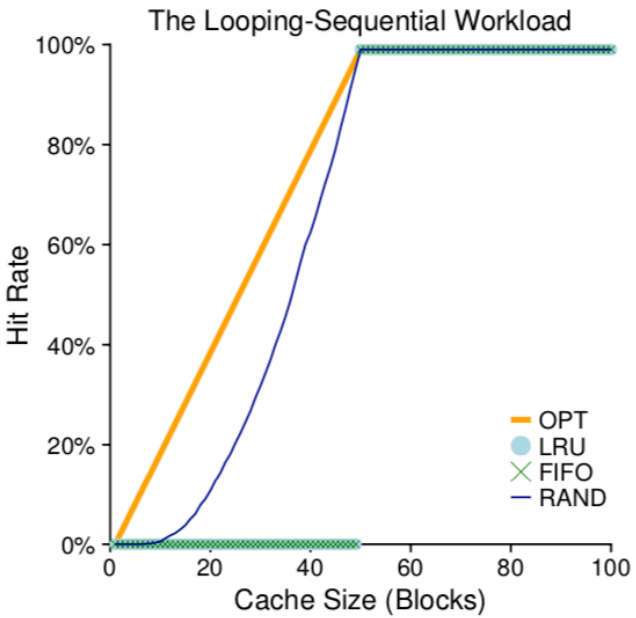

Figure 22.8: **The Looping Workload**

22.7 Implementing Historical Algorithms

(FIFO and random are easy to implement)

Challenge: How do we implement LRU?

To implement LRU perfectly / accurately:

- On each page access (instruction fetch or data load/store)
  - Update data structure to move “page” to front of the list
  - Accounting work on every memory reference
  - Great care must be taken: such accounting can greatly reduce performance

Could we do this in HW?

- On each access, update a time field in PTE
- When replacing page, OS scan all time fields to find least-recently-used page

Prohibitively expensive to scan a large table to find the LRU page.

22.8 Approximate LRU

HW support: **use bit (reference bit)** in the PTE

- When page referenced (read or write)
- HW set **use bit** to 1.
- OS is responsible for clearing the **use bit** to 0.

Clock Algorithm:

- Arrange all pages in a circular list
- A “clock hand” points to some page *P*
- When a replacement must occur
  - **CheckUseBit**:
    - OS checks if *P’s* **use bit** is 0 or 1.
    - If **use bit** == 1: implies that *P* was recently used
      - (Not a good candidate for replacement)
      - Clear *P’s* use bit.
      - Advance clock hand to *P*+1.
      - GOTO CheckUseBit
    - If **use bit** == 0: implies that *P* has not recently been used
      (or we have searched all pages, and all are used)
      - Replace page *P.*

Fig shows that the clock algorithm performs quite well compared to perfect LRU

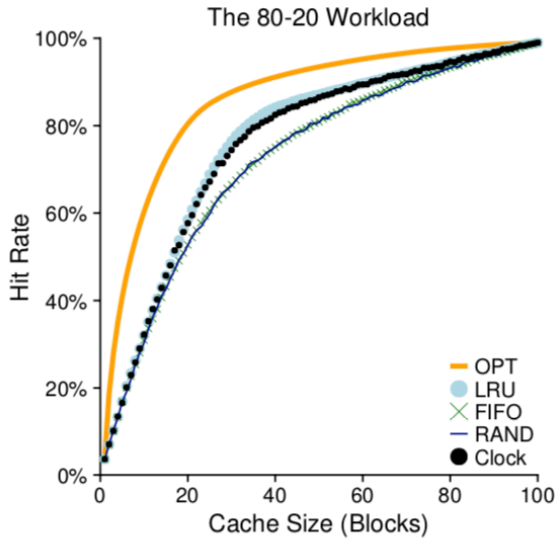

Figure 22.9: **The 80-20 Workload With Clock**

22.9 Considering Dirty Pages

- If page has been modified (dirty)
  - Must be written to disk when being evicted (expensive)
- Else (page is clean)
  - Eviction is free (no need for expensive IO)

The VM system prefer to evict clean pages over dirty pages

HW has dirty bit (or modified bit) in TLB and PT.

Modified Clock Algorithm:

- Scan for pages that are both unused and clean (to evict first)
- Failing to find any such pages
  - Evict unused pages that are dirty (taking the cost of IO)

22.10 Other VM Policies

Page selection policy:

OS must decide when to bring page into memory.

Two approaches:

- **Demand paging**:
  - OS brings in a page when it is accessed (on demand)
  - This means that if the page is not present in PM,
    we will take a page fault and load the page from SS.
- **Prefetching:**
  - If OS already fetching page *P*, it may be likely that page *P*+1 will be needed next.
  - Should only be done if there is a reasonable chance of success…

Policies for writing pages out to disk:

- **One-at-a-time**:
  - Does what you expect, write one page from PM to disk at a time
- **Grouping:**
  - Grouping multiple writes into one is more effective
    because of the nature of disk drives (also called batching or clustering)

22.11 Thrashing

Q: What should OS do when memory is oversubscribed?

Def. Oversubscribed (for the memory case)
the memory demands of the set of running processes exceeds the available PM.

System will constantly be paging: *condition is called **thrashing***

Def. Working set: *set of pages a process is actively using*

Admission control:

- Reduce the set of processes that gets to run
  - Hope the remaining processes's **working set** fit in memory

Linux approach to memory overload:

Out-of-memory (OOM) killer:

- Daemon chooses a memory-intensive process and kills it!
- Problem:
  - If daemon kills the X server — the program that renders stuff on the screen…

22.12 Summary

Modern page replacement algorithms:

- Try to support LRU approximations (like clock algorithm)
- **Scan-resistant**:
  - Avoid worst-case behavior of LRU,
    e.g., for the looping-sequential workload

Importance of page replacement algorithms has decreased

- Discrepancy between memory-access and disk-access times has increased
- Cost of frequent paging: prohibitive

Best solution: buy more memory!