Replacement Policy

How to decide what to evict?

By Matthew Tancreti
For COM S 352
Iowa State University

Replacement Policy

We have seen the issue of **replacement policy** in two contexts: TBL cache and swap

Indeed, it is a common and challenging problem

- Miss rates must be low, every miss is costly
- Data structure must be fast, potentially updated every memory access
- Would like to take advantage of hardware support for anything that happens on every memory access

Our first attempt will be FIFO, which despite its simplicity is not a good choice, it experiences a paradoxical corner-case behavior know as Bélády's anomaly

How to decide which page to evict?

Whoever makes a DESIGN without the Knowledge of PERSPECTIVE will be liable to such Absurdities as are shewn in this Frontispiece.

Absurd perspectives, William Hogarth, 18th Century [source]

Framing the Problem

Main memory can only hold a subset of all pages in the system, the rest are pushed to the disk (swap space)

In a sense, main memory is a cache of the systems “popular pages”

We will discuss replacement policies from the perspective of a cache, but keep in mind the following interpretation for swap

- **cache hit** = page is in main memory
- **cache miss** = **page fault** (page is in swap space)

Reference Strings

We will examine the performance of policies for specific memory access patterns called **page reference strings**

Suppose the memory accesses on a system look like the following

Memory Address	112	58	114	200	220	200	290	58	224
Page Number	0	0	0	1	1	1	2	0	1

Every time there is a page fault the page must be brought to main memory

Consecutive repeated page accesses can never cause additional misses, for this reason they are not interested in them when evaluating policy performance and we remove them from the reference string

Example Reference String (with consecutive repeated pages removed): 0, 1, 2, 0, 1

Comparison to Optimal

To evaluate policies, we need a point of comparison, what are the fewest possible misses for a given cache size

Optimal (OPT) – assume we can predict the future and always chose to evict the page that will be used furthest out (or any page that will never be used again)

Page	0	1	2	0	1	3	0	3	1	2	1
Hit/Miss	M	M	M	H	H	M	H	H	H	M	H
Evict						2				3	
Cache[0]	-	-	0	0	0	0	0	0	0	0	0
Cache[1]	-	0	1	1	1	1	1	1	1	1	1
Cache[2]	0	1	2	2	2	3	3	3	3	2	2

For reference string 0,1,2,0,1,3,0,3,1,2,1 and a cache size of 3, no policy will every have fewer than 5 misses

FIFO

FIFO – evict the page that has been in memory the longest

Advantage: easy to implement and fast (O(1) operations), can be implemented in a fixed size cache using a circular buffer

Page	0	1	2	0	1	3	0	3	1	2	1
Hit/Miss	M	M	M	H	H	M	M	H	M	M	H
Evict						0	1		2	3	
Queue Head	-	-	0	0	0	1	2	2	3	0	0
	-	0	1	1	1	2	3	3	0	1	1
Queue Tail	0	1	2	2	2	3	0	0	1	2	2

Belady’s Anomaly

Intuitively we would expect larger cache to perform better (or at least as well) as smaller cache

FIFO has a strange corner case known as **Belady’s Anomaly**, where sometimes a larger cache does worse

Cache size 3 (9 misses)

Page	1	2	3	4	1	2	5	1	2	3	4	5
Hit/Miss	M	M	M	M	M	M	M	H	H	M	M	H
Evict				1	2	3	4			1	2	
Queue Head	-	-	1	2	3	4	1	1	1	2	5	5
	-	1	2	3	4	1	2	2	2	5	3	3
Queue Tail	1	2	3	4	1	2	5	5	5	3	4	4

Cache size 4 (10 misses)

Page	1	2	3	4	1	2	5	1	2	3	4	5
Hit/Miss	M	M	M	M	H	H	M	M	M	M	M	M
Evict							1	2	3	4	5	1
Queue Head	-	-	-	1	1	1	2	3	4	5	1	2
	-	-	1	2	2	2	3	4	5	1	2	3
	-	1	2	3	3	3	4	5	1	2	3	4
Queue Tail	1	2	3	4	4	4	5	1	2	3	4	5

Random

Random – evict a page at random

Advantages

- easy to implement and fast
- no corner cases such as Belady's anomaly possible because algorithm is not influenced by order of pages (its random)

Page	0	1	2	0	1	3	0	3	1	2	1
Hit/Miss	M	M	M	H	H	M	M	H	M	H	H
Evict						0	1		3		
Cache[0]	-	-	0	0	0	1	2	2	2	2	2
Cache[1]	-	1	1	1	1	2	3	3	0	0	0
Cache[2]	0	0	2	2	2	3	0	0	1	1	1

LRU

LRU – evict the page used the least recently

Advantages

- Better performance than FIFO
- No Belady's anomaly possible

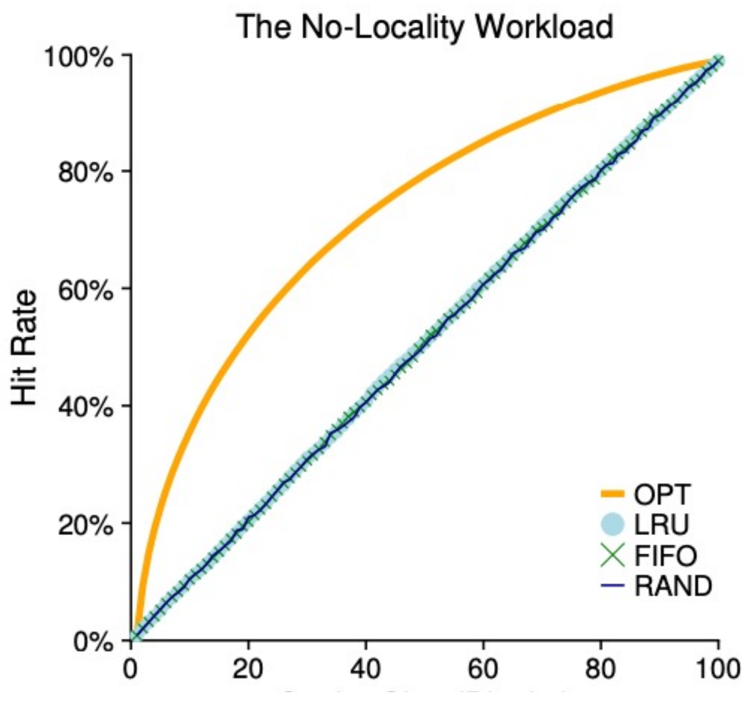
Disadvantage – slow, accounting to do on every page access

Page	0	1	2	0	1	3	0	3	1	2	1
Hit/Miss	M	M	M	H	H	M	H	H	H	M	H
Evict						2				0	
Least Recent	-	-	0	1	2	0	1	1	0	2	3
	-	0	1	2	0	1	3	0	3	3	2
Most Recent	0	1	2	0	1	3	0	3	1	1	1

What if All Accesses Were Random?

Without locality, none of the policies (except the cheating OPT) provide cost-effective benefit

If main memory is 50% of total pages, only 50% hit rate



Percent of total pages that fit in main memory

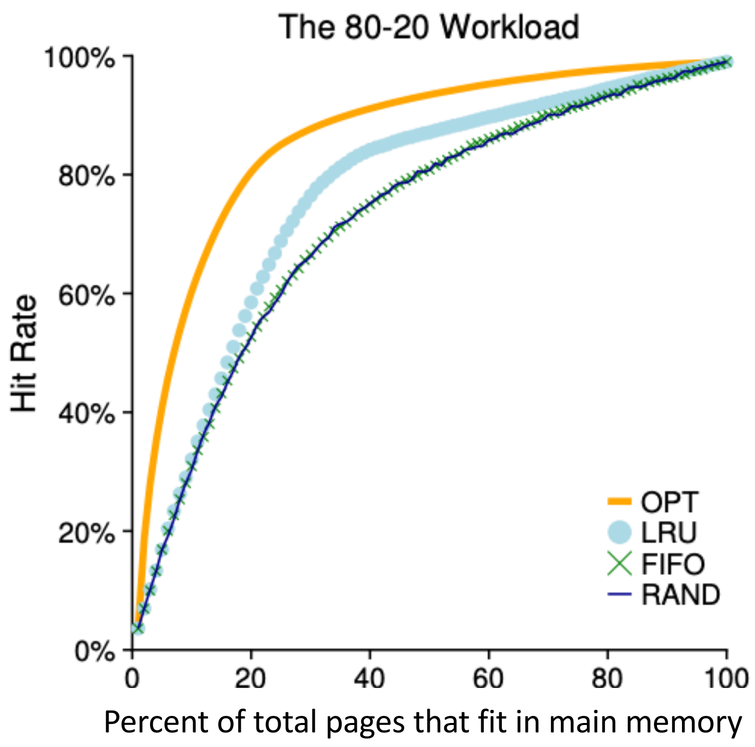
80-20 Workload

More typical, the **80-20 workload** (80% of accesses are to 20% of pages)

- A few pages get most accesses
- Most pages get few accesses

Results from locality (either spatial or temporal)

LRU performs closest to optimal, FIFO and RAND perform the same

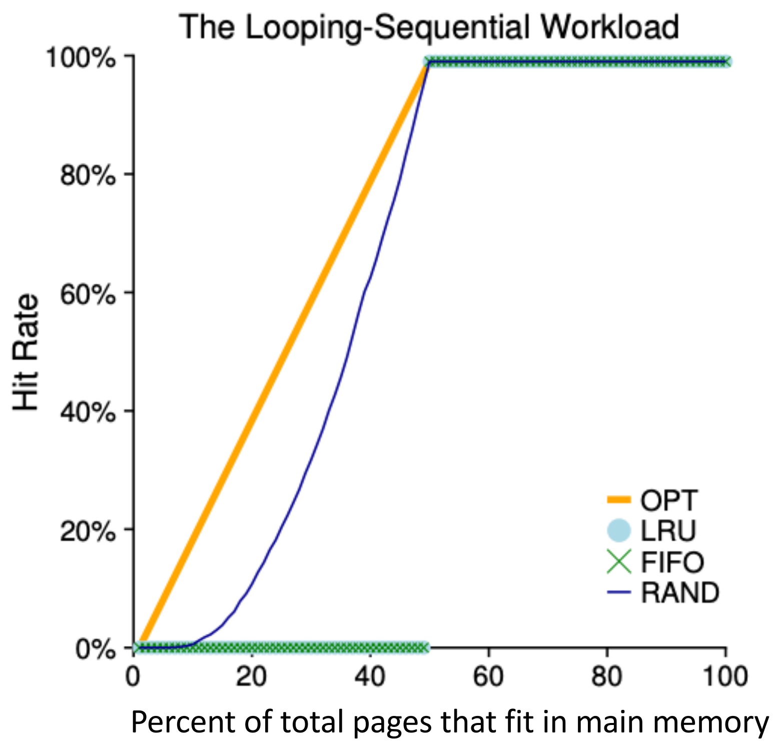


Looping-Sequential Workload

Assume loop repeatedly reads pages 1 to 49 in increasing order

When cache size is below 50, LRU and FIFO have same corner case that causes every access to be a miss

Random avoids corner cases



Clock Algorithm

LRU performs the closest in optimal for typical workloads, but it is really costly on every memory access

Clock Algorithm - is a way to approximate LRU cheaply

Add extra **use bit** to page table entry, on every memory access MMU set use bit of page to 1

When looking for page to evict

- visit pages in round-robin order
- if page use bit is 0, chose that page to evict
- else set use bit to 0 and continue search

Dirty Bit Optimization

If a page has only been read from (never written to) there is no reason to write it back to the swap space when it is evicted

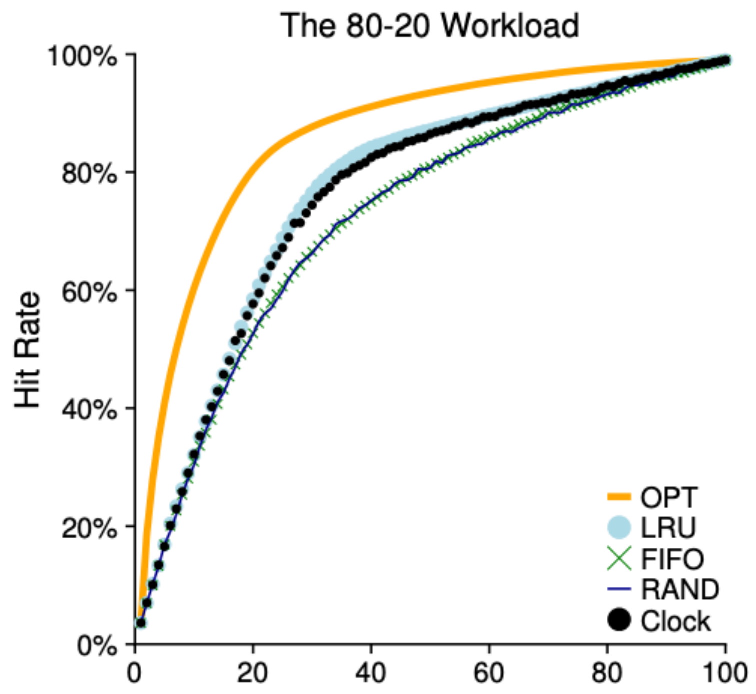
Add a **dirty bit** to the page table, initialize to 0, on every write to the page table the MMU (hardware) sets the bit to 1

Can also make clock algorithm more efficient

- First try to find a page with dirty bit and use bit both 0, because it is lower cost to replace

Clock Policy with 80-20 Workload

Clock is a very close approximation of LRU



Percent of total pages that fit in main memory

Thrashing

Example: System has two processes that both sequentially read from N pages in a continuous loop. The System only has enough main memory to store N pages.

Thrashing is when process is spending more time paging than executing

- Starts at one processes and snowballs into several processes thrashing
- Multiprogramming and multitasking make it worse, not better
- Sudden and extreme drop in system performance

Need to reduce the amount of multiprogramming and multitasking