

# **ECE7995 Caching and Prefetching Techniques in Computer Systems**

## **Lecture 8: Buffer Cache in Main Memory (II)**

# LRU has a Problem!

Guess when the page will be referenced again.

- LRU does not consider frequency of accesses
  - Is a page that has been accessed once in the past as likely to be accessed in the future as one that has been accessed  $N$  times?
- Consequence:
  - System spends resources to keep useless stuff around (low hit rate)
- Cause:
  - Cannot tell between frequent/infrequent refs on time of last reference

# Commonly Observed LRU Problems

## Case 1: Scan

- Scan (sequential read, never used again) of one large data region (larger than physical memory) flushes memory contents

Solution: Track frequency of accesses to page

Pure LFU (Least-frequently-used) replacement

- Problem: LFU can never forget pages from the far past

## Case 2: Looping

Problematic workload for LRU

- Repeated scans of large memory region (larger than physical memory)

Example:

- 5 blocks of physical memory, 6 blocks of address space repeatedly scanned (ABCDEFABCDEFABCDEF...)
  - What happens with LRU?



Solution?

## Commonly Observed LRU Problems (Cont'd)

### Case 3: Different Access Frequencies

- In a Database system, there are 20,000 records, 2000B each.
- Each record is associated with a B-tree index, 20B each.
- In each 4KB page, there are 4000 bytes useful to hold records or indices
- So we need 10000 record pages and 100 index pages.
- Multiple users randomly access records concurrently (Note that the associated index must be read before reading a record).
- So page reference stream is:  $I_1, R_1, I_2, R_2, I_3, R_3, \dots$
- Probability to reference page  $I_i$  is .005, to page  $R_i$  is .000005.
- We know that  $I_i$  is 100 times more likely to be accessed than  $R_i$ .
- Number of page frames is 100.
- However, in the 100 most recently accessed pages, most probably there are 50  $I_i$  and 50  $R_i$ . ( $R_i$  can be a little bit more)

## A Possible Solution --- Using Multiple Page Pools

Page sets with different access frequencies are received into different buffer pools corresponding to various predetermined frequencies.

- Advantage:
  - Pages of different frequencies don't compete with each other.
- Unsolved issues:
  - How to know access frequencies of a page?
  - What to do when the frequencies changes?
  - How to set (or tune) sizes of the pools?

Refer to this paper for clues:

Yuanyuan Zhou, James F. Philbin, and Kai Li, "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches", USENIX 2001."

# LRU-K

- Achieves the objective of using multiple page pools
- But does not explicitly set up multiple pools
- Does not rely on external hints about workload characteristics
- Adapts in real time to changing patterns of access

## Reference

- E. O'Neil, P. O'Neil, G. Weikum: The LRU-K Page Replacement Algorithm for Database Disk Buffering, SIGMOD 1993

# LRU-K Basic Concepts

- Observation:
  - The page access history used by LRU is too limited: simply the time of last reference.
  - While taking deep history into consideration, we need to give recent history a higher priority.
- Idea: Take into account history of last  $K$  references of a page to predict next reference of the page ( $K \geq 2$ , usually 2)
- Comparison with LRU:
  - $K$  is 1 (LRU-1), so too shortsighted;
- Comparison with LFU:
  - When  $K$  is large, both look way back to deep history
  - LRU- $K$  (when  $K$  is very large) shares the problems of LFU
    - ✓ Ignore the fact that recent references carry a higher weight than past references in predicting access events in the near future;
    - ✓ Not responsive to changes of access patterns
    - ✓ Impose high overhead

## Some Notations

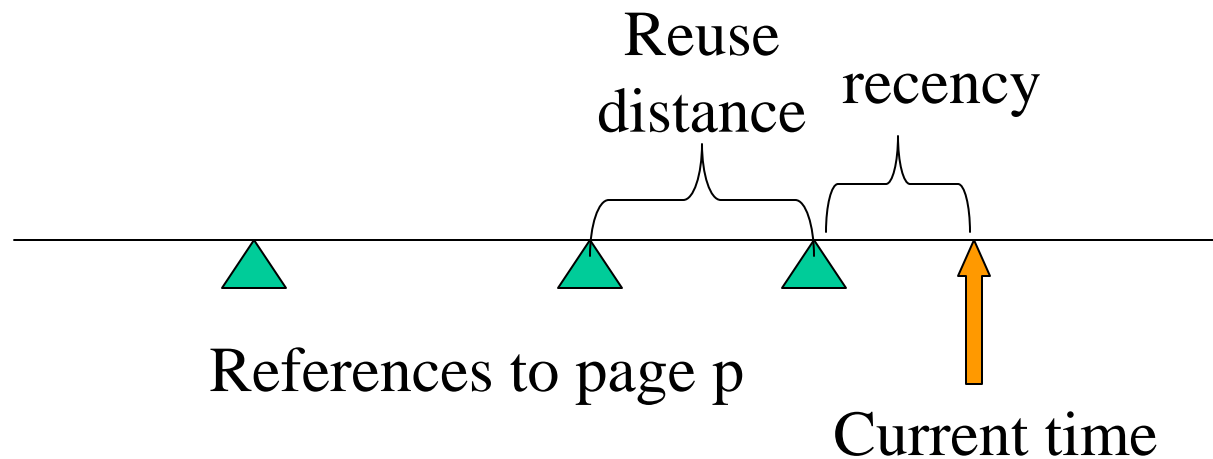
Pages  $N = \{1, 2, \dots, n\}$

Reference string  $r_1, r_2, \dots, r_t, \dots$

$r_t = p$ : page  $p$  is referenced at time  $t$  (all time intervals are measured in terms of counts of successive accesses in the stream)

Probability that next reference goes to  $p$ :  $\Pr(r_{t+1} = p) = b_p$

Time between successive references of  $p$ :  $I_p = 1/b_p$



# The LRU-K Algorithm

Backward K-distance  $b_t(p,K)$ :

- #refs from t back to the Kth most recent references to p

$b_t(p,K) = \infty$  if Kth ref doesn't exist

Algorithm:

- Drop page p with maximum backward K-distance  $b_t(p,K)$

Ambiguity can be resolved using subsidiary policy, e.g., LRU

LRU-2 is better than LRU-1

- Why? (Examine the three access scenarios where LRU doesn't work)

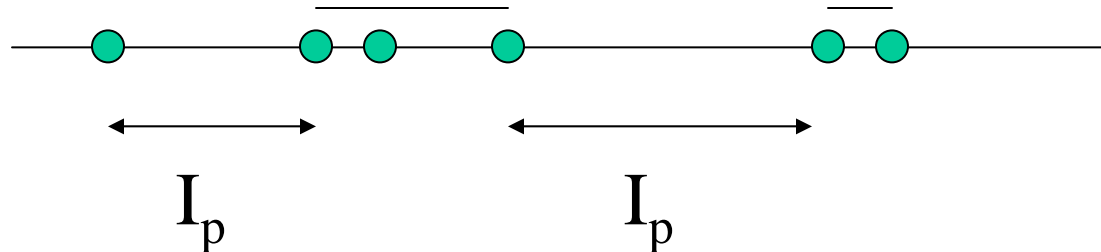
# Issues with LRU-K

## Early page replacement

- $b_t(p, K) = \infty$ , so drop  $p$
- But what if there is correlated reference of  $p$  that is coming.
- Correlated References (in the context of DB)
  - Intra-transaction, e.g., read tuple, followed by update
  - Transaction retry
  - Intra-process, i.e., a process references page via 2 transactions, e.g., update RIDs 1-10, commit, update 11-20, commit, ...

## A Solution

- Introduce a time-out period before computing  $bt(p,K)$  for replacement, say 5 sec
  - With or without timeouts, wrong decisions may result,
  - e.g., continuous one-time read/update transactions
    - ✓ Without timeout
      - 1) Algorithm sees  $p$  (read)
      - 2) Drops it ( $b_t(p,K) = \infty$ ) (wrong decision)
    - ✓ With timeout
      - 1) Algorithm sees  $p$  (read)
      - 2) Sees it again before timeout (update)
      - 3) Keeps it around (wrong decision again)
- Timeout must be carefully used
  - Inter-arrival time (a.k.a. reuse time) should be computed based on *uncorrelated* access  $\overline{CRP}$  (Correlated Reference Period)



- The good news is that this is really not an issue in a practical with sufficiently large buffer cache.

# Problem 2

## Reference Retained Information

- Algorithm needs to keep info for pages that may not be resident anymore, e.g., LRU-2
  - P is referenced and comes in for the first time
  - $b_t(p,2) = \infty$ , p is dropped
  - P is referenced again
  - If no info about p's previous references is retained, p may be dropped again

## Solution to Problem 2

### Retained Information Period (RIP)

- Period after which we drop information about page  $p$
- Upper bound: max backward  $K$ -distance of all pages we want to ensure to be memory resident
- In practice we usually maintain a system-wide upper bound, such as the maximum amount of history information retained.

# Pseudo Code of LRU-K

- HIST(p) denotes the history control block of page p; it contains the times of the K most recent references to page p, discounting correlated references: HIST(p,1) denotes the time of last reference, HIST(p,2) the time of the second to the last reference, etc.
- LAST(p) denotes the time of the most recent reference to page p, regardless of whether this is a correlated reference or not.

Procedure to be invoked upon reference to page p at time t:

```
if p is already in the buffer
then /* update history information of p */
  if t - LAST(p) > Correlated_Reference_Period
  then /* a new, uncorrelated reference */
    correl_period_of_refd_page := LAST(p) - HIST(p,1)
    for i := 2 to K do
      HIST(p,i) := HIST(p,i-1) +
                    correl_period_of_refd_page
    od
    HIST(p,1) := t
    LAST(p) := t
  else /* a correlated reference */
    LAST(p) := t
  fi
fi
```

```
else /* select replacement victim */
  min := t
  for all pages q in the buffer do
    if t - LAST(q) > Correlated_Reference_Period
      /* if eligible for replacement */
      and HIST(q,K) < min
      /* and max Backward K-distance so far */
    then
      victim := q
      min := HIST(q,K)
    fi
  od
  if victim is dirty then
    write victim back into the database fi
  /* now fetch the referenced page */
  fetch p into the buffer frame previously held by victim
  if HIST(p) does not exist
  then /* initialize history control block */
    allocate HIST(p)
    for i := 2 to K do HIST(p,i) := 0 od
  else
    for i := 2 to K do HIST(p,i) := HIST(p,i-1) od
  fi
  HIST(p,1) := t
  LAST(p) := t
fi
```

# Simulation Results

B	LRU-1	LRU-2	LRU-3	A <sub>0</sub>	B(1)/B(2)
60	0.14	0.291	0.300	0.300	2.3
80	0.18	0.382	0.400	0.400	2.6
100	0.22	0.459	0.495	0.500	3.0
120	0.26	0.496	0.501	0.501	3.3
140	0.29	0.502	0.502	0.502	3.2
160	0.32	0.503	0.503	0.503	2.8
180	0.34	0.504	0.504	0.504	2.5
200	0.37	0.505	0.505	0.505	2.3
250	0.42	0.508	0.508	0.508	2.2
300	0.45	0.510	0.510	0.510	2.0
350	0.48	0.513	0.513	0.513	1.9
400	0.49	0.515	0.515	0.515	1.9
450	0.50	0.517	0.518	0.518	1.8

Table 4.1. Simulation results of the two pool experiment, with disk page pools of  $N_1 = 100$  pages and  $N_2 = 10,000$  pages. The first column shows the buffer size  $B$ . The second through fifth columns show the hit ratios of LRU-1, LRU-2, LRU-3, and  $A_0$ . The last column shows the equi-effective buffer size ratio  $B(1)/B(2)$  of LRU-1 vs. LRU-2.

- LRU-K is much better
- There is no big difference between LRU-2 and LRU-3. So LRU-2 of lower cost is sufficiently good.

$A_0$ : the optimal MIN algorithm

## Simulation Results (Cont'd)

B	LRU-1	LRU-2	LFU	B(1)/B(2)
100	0.005	0.07	0.07	4.5
200	0.01	0.15	0.11	3.25
300	0.02	0.20	0.15	3.0
400	0.06	0.23	0.17	2.75
500	0.09	0.24	0.19	2.4
600	0.13	0.25	0.20	2.16
800	0.18	0.28	0.23	1.9
1000	0.22	0.29	0.25	1.6
1200	0.24	0.31	0.27	1.66
1400	0.26	0.33	0.30	1.5
1600	0.29	0.34	0.31	1.5
2000	0.31	0.36	0.33	1.3
3000	0.38	0.40	0.39	1.1
5000	0.46	0.47	0.44	1.05

Table 4.3. Simulation results on buffer cache hit ratios using an OLTP trace.

- OLTP is an on-line transaction processing benchmark, and the trace used here is from a production system of a bank.
- LRU-2 is superior than LRU.
- LFU is as good as LRU-2 in the experiment. But when access patterns (moving hot spots) changes, LFU is supposed to perform worse due to its inability to forget any previous references and adapt itself.

# A Summary of LRU-K

- Introduces frequency into recency-based policies such as LRU.
- Being adaptive to the changing access patterns (compared to LFU)
- Effectively distinguish pages of being used for once and those frequently used.
- But there are issues with it:
  - It cannot deal with pages whose reference frequency is not significantly different (such as the looping pattern).
  - Its cost is way high ( $O(N)$ ,  $N$  is the number of pages in the cache)