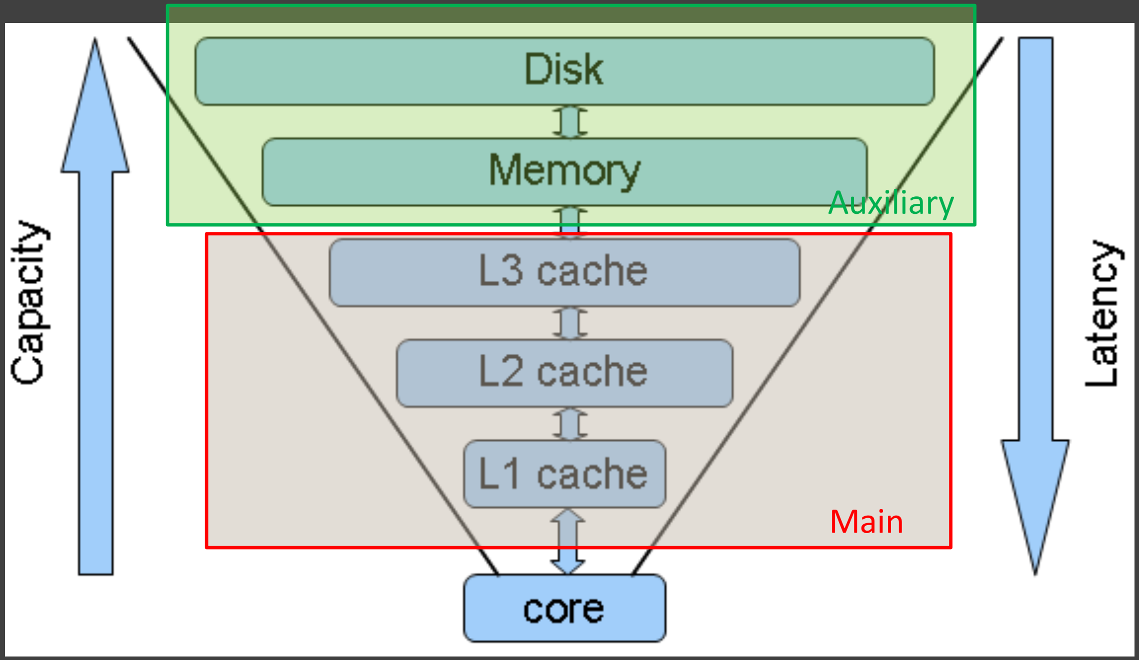
ARC: A SELF-TUNING, LOW OVERHEAD REPLACEMENT CACHE

AUTHORS: NIMORD MEGIDDO AND DHARMENDRA S. MOHDHA

PRESENTER: XIAOYUAN GUO

Cache



http://www.1024cores.net/home/parallel-computing/cache-oblivious-algorithms

Cache replacement

- Memories are stored in the size of pages.
- Consider demanding a page of memory that is not paged in the cache
- When the cache is full, “page out” happens, the page selected is chose by a cache replacement policy.

Policy evaluation metrics

- Hit rate: the fraction of pages that can be served from the main memory
- Miss rate: the fraction of pages that must be paged into the cache from the auxiliary memory
- Low space overhead?
- Good replace policy = $\uparrow$ hit rate + $\downarrow$ miss rate + $\downarrow$ space overhead

Objectives

Design a replacement policy ARC(Adaptive Replacement Cache):

- High hit ratio + low complexity
- Dynamically evolving workloads

Prior replacement policies

- Offline Optimal(MIN): replaces the page that has the greatest forward distance
  - Require knowledge of future
  - Provides an upper-bound
- Recency(LRU):the Least Recently Used
  - Most widely used policy
- Frequency(LFU):the Least Frequently Used
  - Optimal under independent reference model

Prior replacement policies

- LRU-2: replace page with the least recent penultimate reference
  - Better hit ratio
  - Need to maintain a priority queue
  - Corrected in 2Q policy
  - Must still decide how long a page that has only been accessed once should be kept in the cache

Prior replacement policies

- LIRS(Low Inter-Reference Recency Set)
- FBR(Frequency-Based Replacement)
- LRFU(Least Recently/Frequently Used):
  - Sumsumes LRU and LFU
  - All require a tuning parameter
- ALRFU(Automatic LRFU)
  - Adaptive version of LRFU
  - Still requires a tuning parameter

A class of replacement polices

c = cache size (# of pages)

π = cache replacement policy

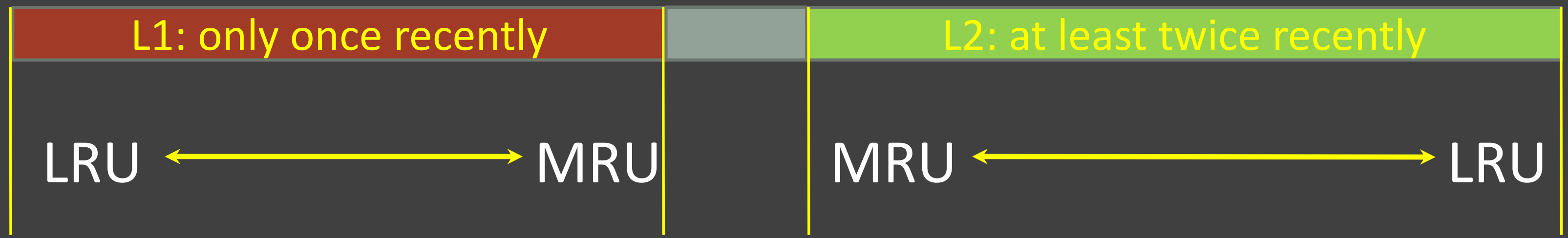
$\pi(c)$ => policy manages page c

$DBL(2c)$: mange and remember twice # of pages present in the cache

$\Pi(c)$: a new class of cache replacement policies

$DBL(2c)$

Given a cache with 2c pages: how could $DBL(2c)$ manage it?



$$0 \leq |L1| + |L2| \leq 2c \,, 0 \leq |L1| \leq c \,, 0 \leq |L2| \leq 2c$$

L1 L2 Unused

Note: LRU(least recently used); MRU(Most recently used)

$DBL(2c)$

Input: $x_1, x_2, x_3, x_4, x_5, \ldots, x_t, \ldots$

Case 1: $\ldots x_t \ldots$ | $\ldots x_t \ldots$

x_t is in L1 or L2. Cache hit! x_t = MRU in L1 or L2

Case 2: …… | ……

x_t is neither in L1 nor in L2. Miss cache!

Subcase A: $\cancel{x_k}$ … … x_t | ……

|L1| = c , delete x_k (LRU) in L1, x_t = MRU in L1

Subcase B1: …… x_t | …… $\cancel{x_k}$

|L1| < c & $|L1| + |L2| = 2c$, delete x_k (LRU) in L2, x_t = MRU in L1

Subcase B2: …… x_t | | ……

|L1| < c & $|L1| + |L2| < 2c$, insert x_t (MRU) in L1, x_t= MRU in L1

L1 L2 Unused

Note: LRU(least recently used); MRU(Most recently used)

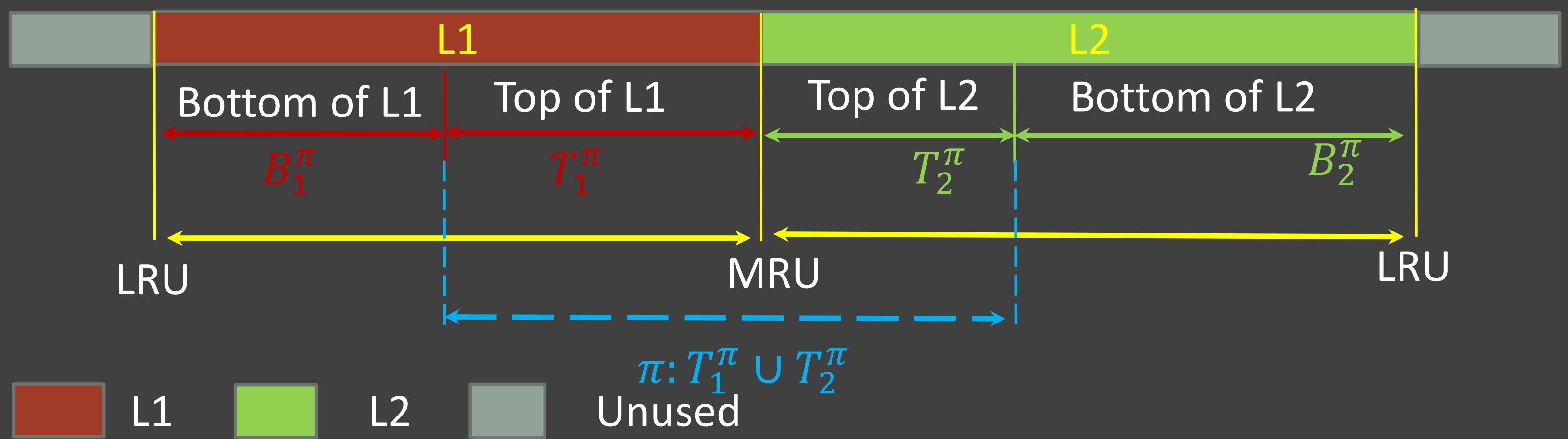
$\Pi(c)$

$\Pi(c)$: a class of demand paging cache replacement policies

Track all $2c$ items managed by $DSB(2c)$, actually keep c pages

L1, L2 : lists associated with $DSB(2c)$

$\pi(c) \in \Pi(c)$: $L_1 = T_1^\pi \cup B_1^\pi$, $L_2 = T_2^\pi \cup B_2^\pi$

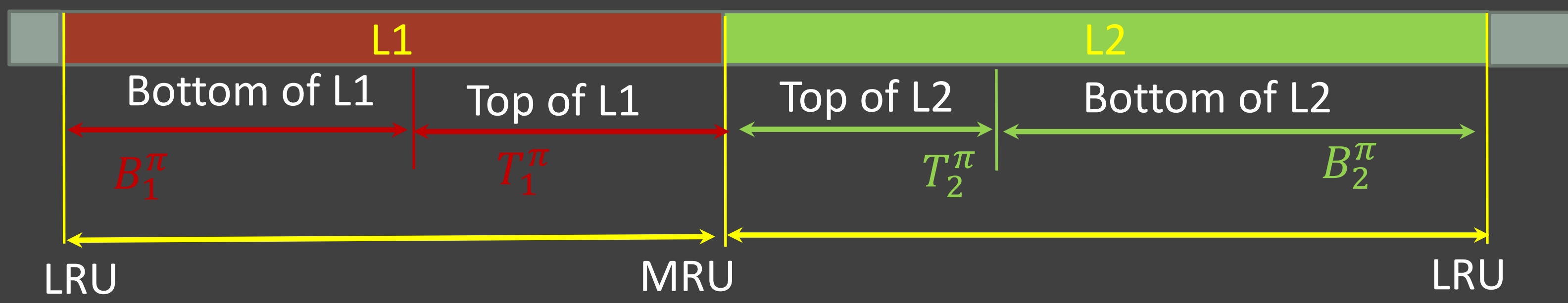


Note: LRU(least recently used); MRU(Most recently used)

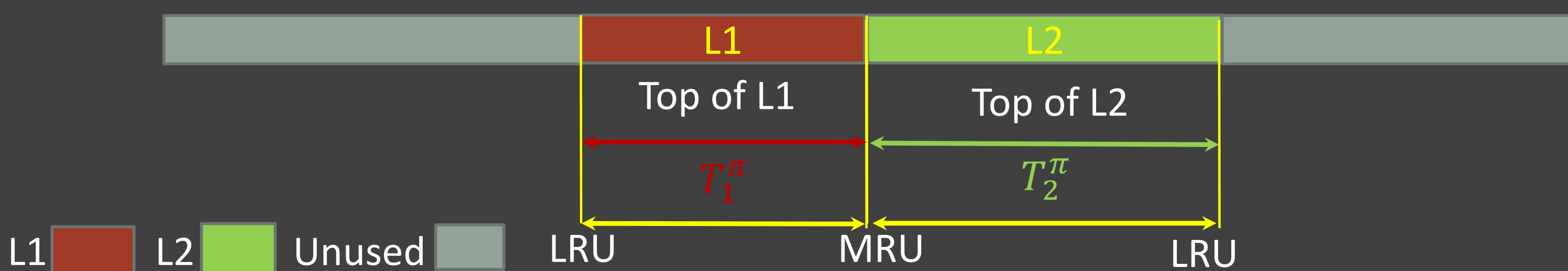
$\Pi(c)$ conditions

A.1 $T_1^\pi \cap B_1^\pi = \emptyset, T_2^\pi \cap B_2^\pi = \emptyset,$

$L_1 = T_1^\pi \cup B_1^\pi, L_2 = T_2^\pi \cup B_2^\pi$



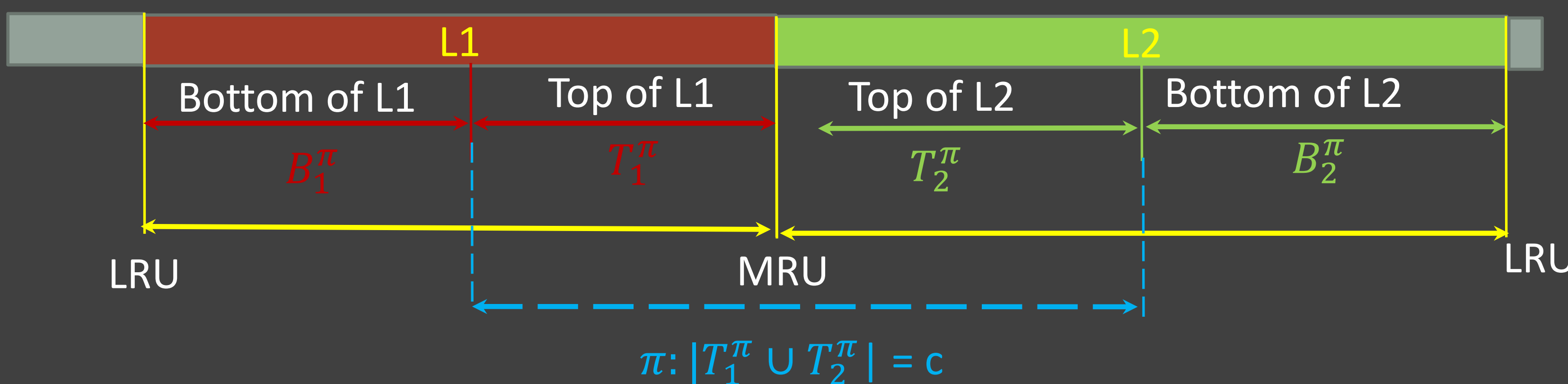
A.2 if $|L_1 \cup L_2| < c$, $B_1^\pi = \emptyset$, $B_2^\pi = \emptyset$



Note: LRU(least recently used); MRU(Most recently used)

$\Pi(c)$ conditions

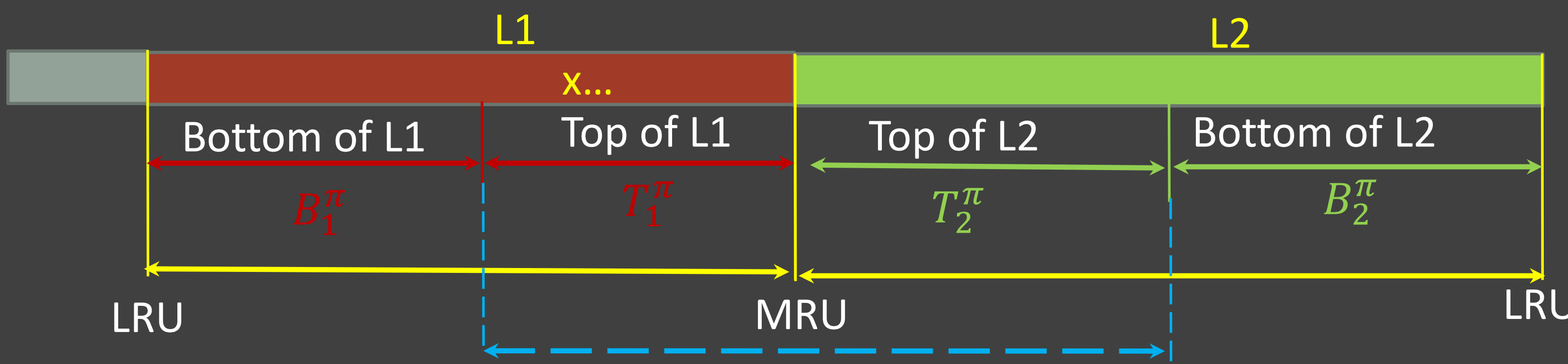
A.3 if $|L_1 \cup L_2| \geq c, |T_1^\pi \cup T_2^\pi| = c$



A.4 $(T_1^\pi = \emptyset)$ or $(B_1^\pi = \emptyset)$ or the LRU page in T_1^π is more recent than the MRU page in B_1^π.

$\Pi(c)$ conditions

A.5 For all traces and at each time, $T_1^\pi \cup T_2^\pi$ will contain exactly those pages that would be maintained in cache by the policy $\pi(c)$



π: $|T_1^\pi \cup T_2^\pi|$ = c

Remark:

- If a page x in L1 is kept, then all pages in L1 that are more recent than it must all be kept in the cache.
- If $|T_1^\pi \cup T_2^\pi|$ = c (cache is full), when a cache miss occurs, 1) replace LRU in T_1^π or 2) replace LRU in T_2^π.

L1 L2 Unused

Note: LRU(least recently used); MRU(Most recently used)

ARC-Fixed replacement cache

$FRC_p(c) \in \Pi(c)$:$0 \leq p \leq c$

$T_{1,p} \equiv T_1^{FRC_p(c)}, T_{2,p} \equiv T_2^{FRC_p(c)},$

$B_{1,p} \equiv B_1^{FRC_p(c)}, B_{2,p} \equiv B_2^{FRC_p(c)}$

$FRC_p(c)$ attempts to keep exactly p pages in $T_{1,p}$ and exactly c $- p$ in $T_{2,p}$, that is:

$$|T_{1,p}| = p$$

$$|T_{2,p}| = c - p$$

ARC-Fixed replacement cache

p is the target size for the list $T_{1,p}$

B.1 If $|T_{1,p}| > p$, replace the LRU page in $T_{1,p}$.

B.2 If $|T_{1,p}| < p$, replace the LRU page in $T_{2,p}$.

B.3 If $|T_{1,p}| = p$,and the missed page is in $B_{1,p}$(resp. $B_{2,p}$), replace the LRU page in $T_{2,p}$(resp. $T_{1,p}$).

ARC-Policy

$ARC_p(c) \in \Pi(c)$:$0 \leq p \leq c$ (adaptation parameter)

Given a fixed p, $ARC_p(c)$ -> $FRC_p(c)$

Differently, p in ARC is not a fixed one over the entire workload

$T_1^{ARC}, B_1^{ARC}, T_2^{ARC}, B_2^{ARC}$ denote a dynamic partition of L1 and L2

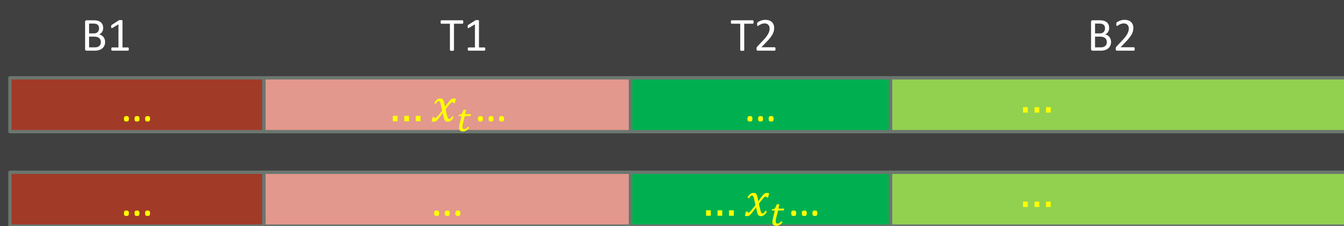
$T_1 \equiv T_1^{ARC}, B_1 \equiv B_1^{ARC}, T_2 \equiv T_2^{ARC}, B_2 \equiv B_2^{ARC}$

ARC-Policy

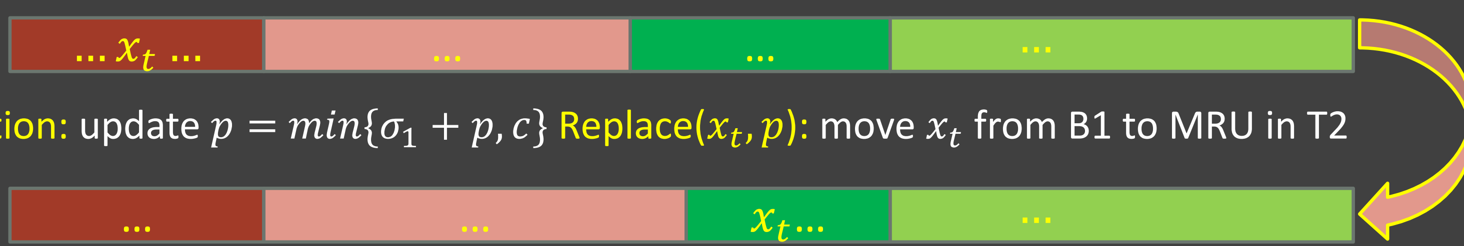
Input: $x_1, x_2, x_3, x_4, x_5, \dots, x_t, \dots$

Initialization: $p = 0$

Case 1: x_t is in T1 or T2, cache hit! move x_t to MRU in T2



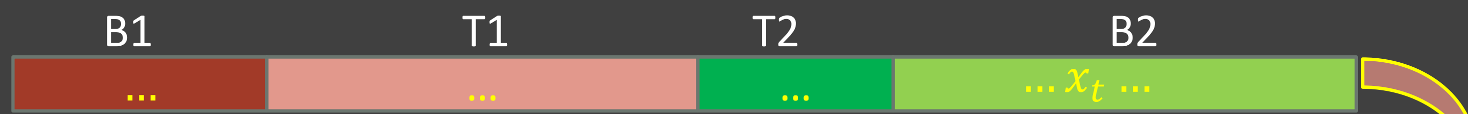
Case 2: x_t is in B1, cache miss for ARC(c)



Adaptation: update $p = min\{\sigma_1 + p, c\}$ Replace(x_t, p): move x_t from B1 to MRU in T2

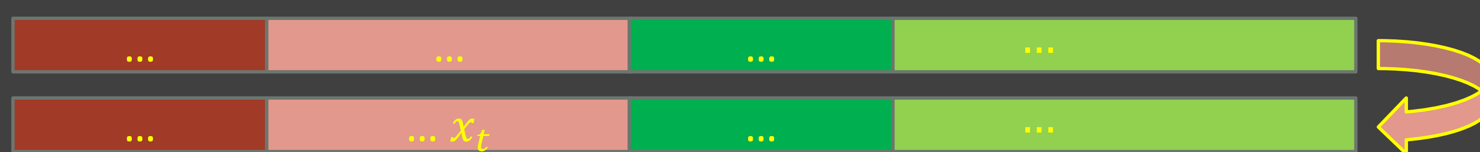
ARC-Policy

Case 3: x_t is in B2, cache miss for ARP(c)



Adaptation: update $p = max\{p - \sigma_2, 0\}$ Replace(x_t, p): move x_t from B2 to MRU in T2

Case 4: x_t is not in T1, T2, B1, B2, cache miss for both ARP(c) and DSB(2c)



Subcase A: |L1| = c
If (|T1|<c) Delete LRU page in B1,
replace(x_t, p)
Else B1 is empty, delete LRU page in T1

Subcase B: |L1| < c
If (|T1|+ |T2| + |B1| + |B2|>=c)
delete LRU in B2 if (|T1|+ |T2| + |B1| + |B2|==2c)
replace(x_t, p)

Finally, fetch x_t to the cache and move it to MRU in T1

B1 T1 T2 B2

Experimental results

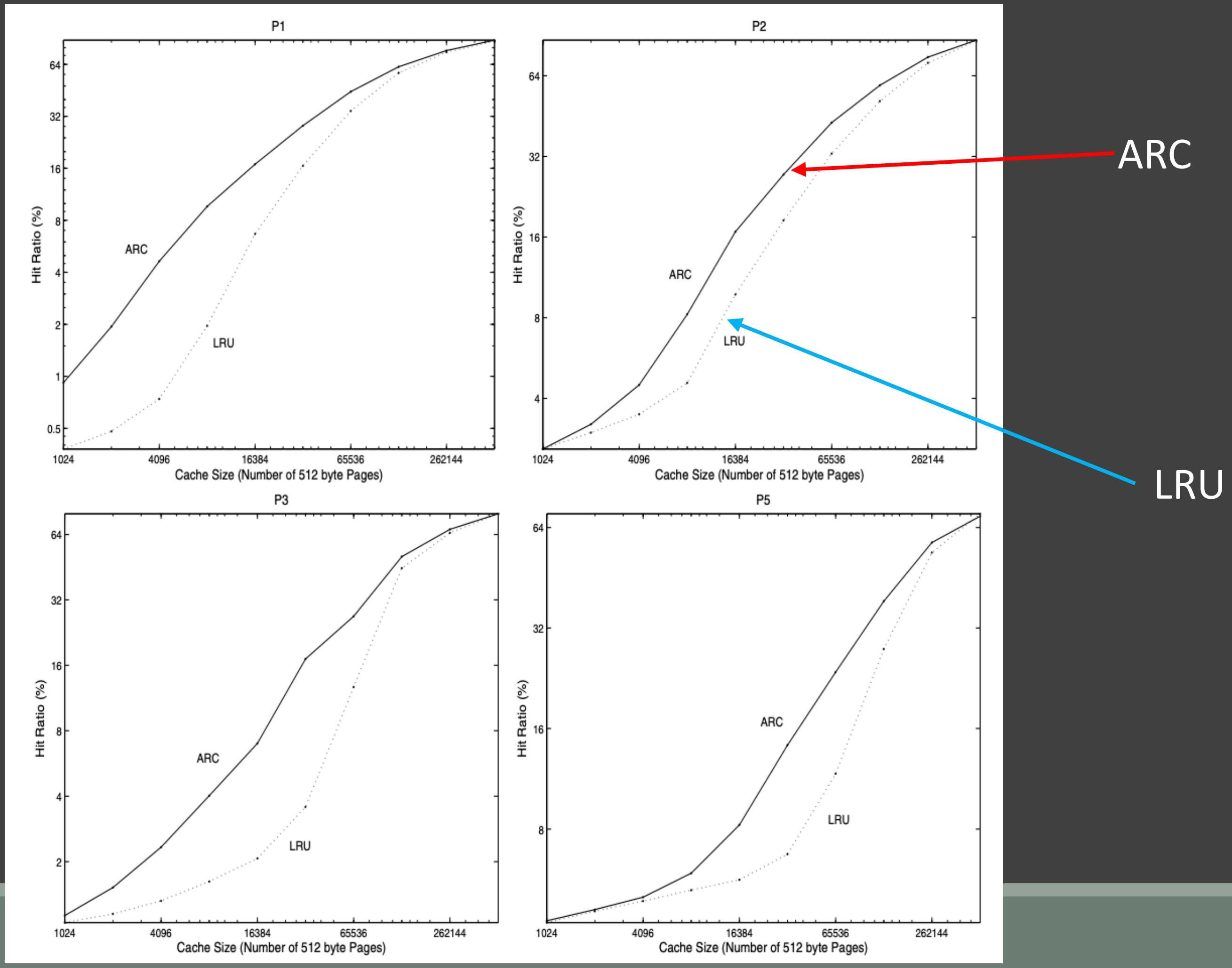
Trace Name	Number of Requests	Unique Pages
OLTP	914145	186880

OLTP

c	LRU	ARC	FBR	LFU	LIRS	MQ	LRU-2	2Q	LRFU	MIN
	ONLINE						OFFLINE			
1000	32.83	38.93	36.96	27.98	34.80	37.86	39.30	40.48	40.52	53.61
2000	42.47	46.08	43.98	35.21	42.51	44.10	45.82	46.53	46.11	60.40
5000	53.65	55.25	53.53	44.76	47.14	54.39	54.78	55.70	56.73	68.27
10000	60.70	61.87	62.32	52.15	60.35	61.08	62.42	62.58	63.54	73.02
15000	64.63	65.40	65.66	56.22	63.99	64.81	65.22	65.82	67.06	75.13

TABLE IV. A comparison of ARC hit ratios with those of various cache algorithms on the OLTP trace. All hit ratios are reported as percentages. It can be seen that ARC outperforms LRU, LFU, FBR, LIRS, and MQ and performs as well as LRU-2, 2Q, and LRFU even when these algorithms use the best offline parameters.

Experimental results



Experimental results

- Tested over 23 traces
- Always outperforms LRU
- Performs as well as more sophisticated policies even when they are specifically tuned for the workload