Outperforming LRU with an Adaptive Replacement Cache Algorithm

Paper by - Nimrod Megiddo Dharmendra S. Modha
IBM Almaden Research Center

Presentation by - Anuj Rai & Mahip Soni

Introduction

- Caching is fundamental wherever computing is involved.
- Cache works on the principle of hit and miss.
- Cache is mostly evaluated by the criteria of Hit Ratio.
- A replacement policy tells which page to evict. Eg FCFS, LRU, ARC etc.
- Replacement policy has its overhead, which should be less than improvement it provides.

ARC

- The paper discusses about ARC cache management policy.
- ARC is supposed to have low overhead than LRU.
- ARC is self tuning.
- ARC uses the concept of frequency, and recency.
- ARC maintains two LRU pages lists: L1 and L2, L1 captures recency, while L2 captures frequency.

ARC

- Implementation contains four lists : MRU(T1), MRUG(B1), MFU(T2), MFUG(B2).
- T1, which contains the top or most-recent pages in L1, and B1, which contains the bottom or least-recent pages in L1
- T2, which contains the top or most-recent pages in L2, and B2, which contains the bottom or least-recent pages in L2

Algorithm

ARC(c) INITIALIZE $T_1 = B_1 = T_2 = B_2 = 0, p = 0$. x - requested page.

Case I. $x \in T_1 \cup T_2$ (a hit in ARC(c) and DBL($2c$)): Move x to the top of T_2.

Case II. $x \in B_1$ (a miss in ARC(c), a hit in DBL($2c$)):
Adapt $p = \min\{c, p + \max\{|B_2|/|B_1|, 1\}\}$. REPLACE($p$). Move x to the top of T_2 and place it in the cache.

Case III. $x \in B_2$ (a miss in ARC(c), a hit in DBL($2c$)):
Adapt $p = \max\{0, p - \max\{|B_1|/|B_2|, 1\}\}$. REPLACE($p$). Move x to the top of T_2 and place it in the cache.

Case IV. $x \in L_1 \cup L_2$ (a miss in DBL($2c$) and ARC(c)):

case (i) $|L_1| = c$:

If $|T_1| < c$ then delete the LRU page of B_1 and REPLACE(p).
else delete LRU page of T_1 and remove it from the cache.

case (ii) $|L_1| < c$ and $|L_1| + |L_2| \geq c$:

if $|L_1| + |L_2| = 2c$ then delete the LRU page of B_2.
REPLACE(p).

Put x at the top of T_1 and place it in the cache.

Subroutine REPLACE(p)

if $(|T_1| \geq 1)$ and $((x \in B_2$ and $|T_1| = p)$ or $(|T_1| > p))$ then move the LRU page of T_1 to the top of B_1 and remove it from the cache.

else move the LRU page in T_2 to the top of B_2 and remove it from the cache.

Source: Megiddo, N. and Modha, D.S., 2004. Outperforming LRU with an adaptive replacement cache algorithm.

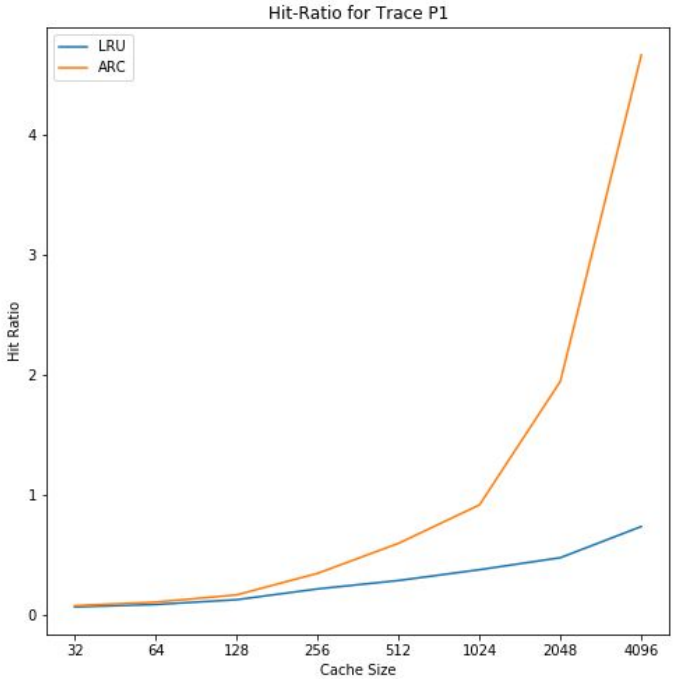
Implementation Approach

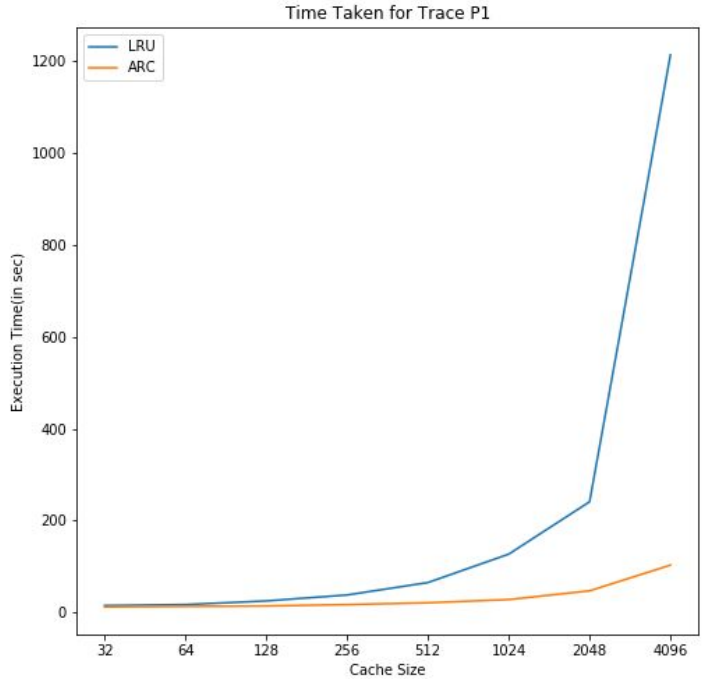
- ARC was implemented as described in paper
- Used Standard Template Library(STL) from C++ to implement necessary queue operations like insertion and deletion
- Program reads traces and performs lookup in the cache (implemented using vectors)
- Writes the output i.e. Hit count, Miss count, Time taken and Hit Ratio in the output file only

Experiments

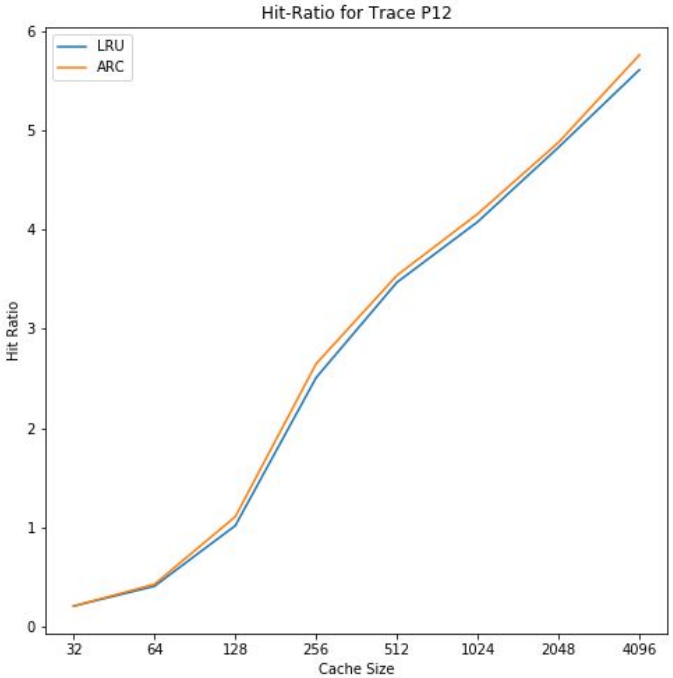
- P1 through P7 over several months from Windows NT workstations.
- All traces have page size of 512 bytes.
- We run both the algorithms on all these traces.
- We plot two graphs for each trace : Cache Size vs RunTime, CacheSize vs Hit Ratio.

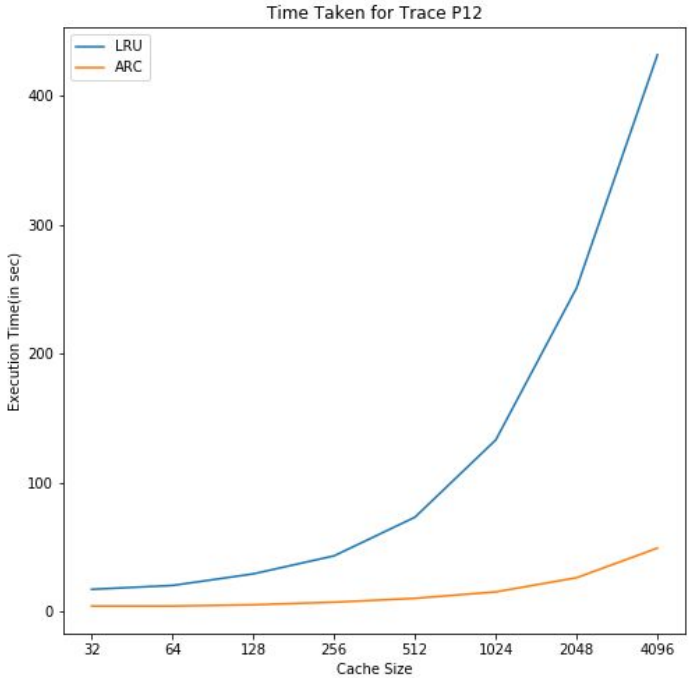
Trace P1


Hit-Ratio for Trace P1
LRU
ARC
Hit Ratio
0
1
2
3
4
32
64
128
256
512
1024
2048
4096
Cache Size

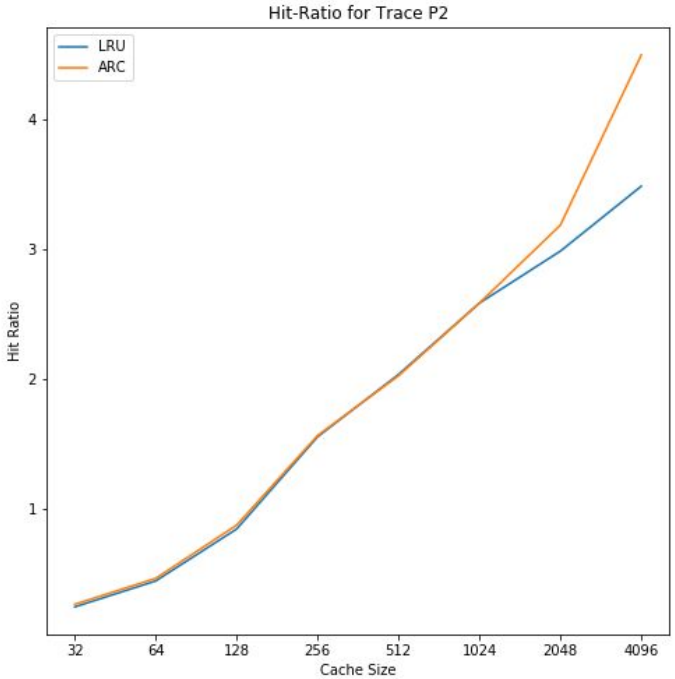

Time Taken for Trace P1
LRU
ARC
Execution Time(in sec)
0
200
400
600
800
1000
1200
32
64
128
256
512
1024
2048
4096
Cache Size

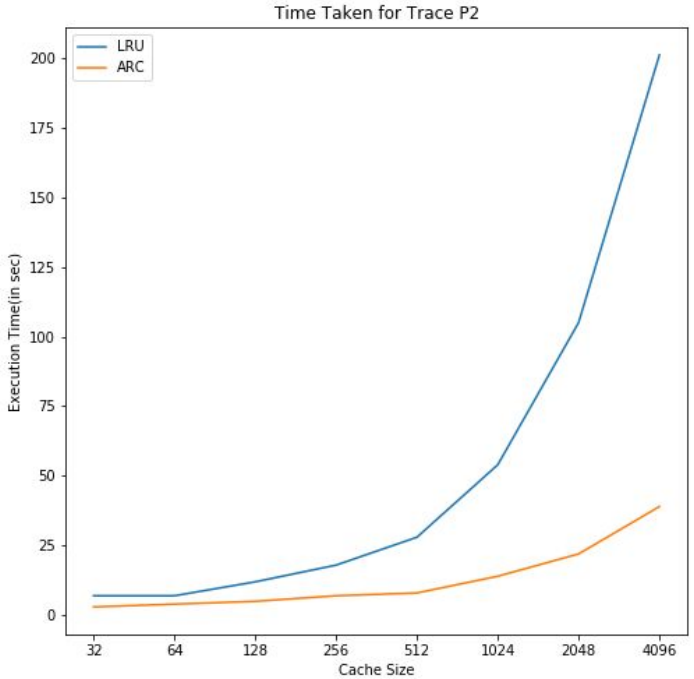
Trace P12


Hit-Ratio for Trace P12
LRU
ARC
Hit Ratio
Cache Size
0
1
2
3
4
5
6
32
64
128
256
512
1024
2048
4096

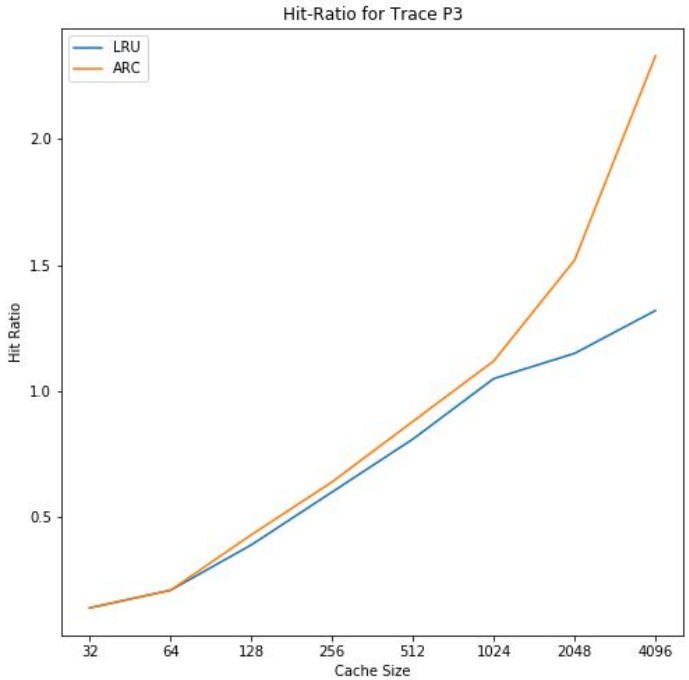

Time Taken for Trace P12
LRU
ARC
Execution Time(in sec)
Cache Size
0
100
200
300
400
32
64
128
256
512
1024
2048
4096

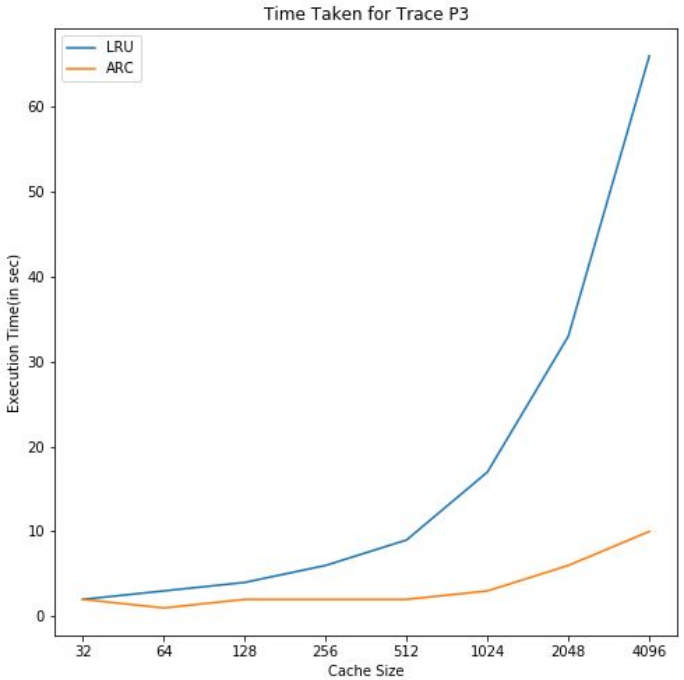
Trace P2


Hit-Ratio for Trace P2
LRU
ARC
Hit Ratio
1
2
3
4
32
64
128
256
512
1024
2048
4096
Cache Size

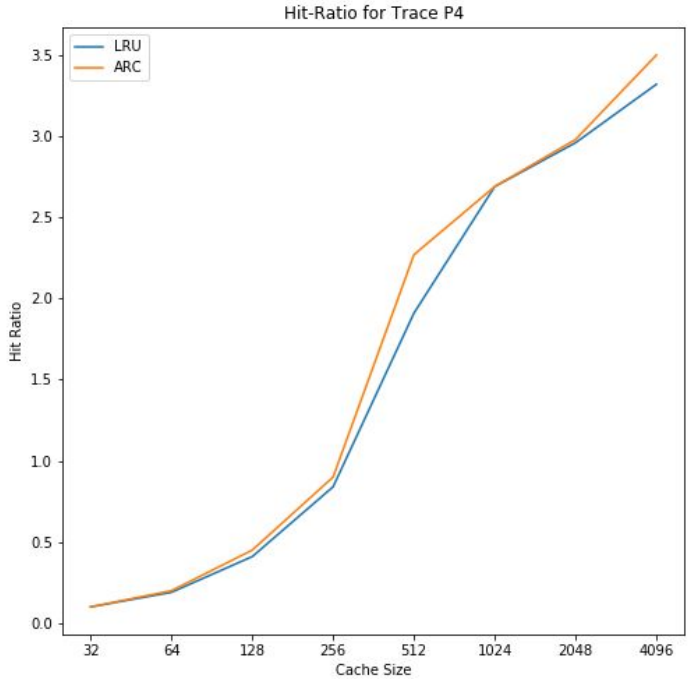

Time Taken for Trace P2
LRU
ARC
Execution Time(in sec)
0
25
50
75
100
125
150
175
200
32
64
128
256
512
1024
2048
4096
Cache Size

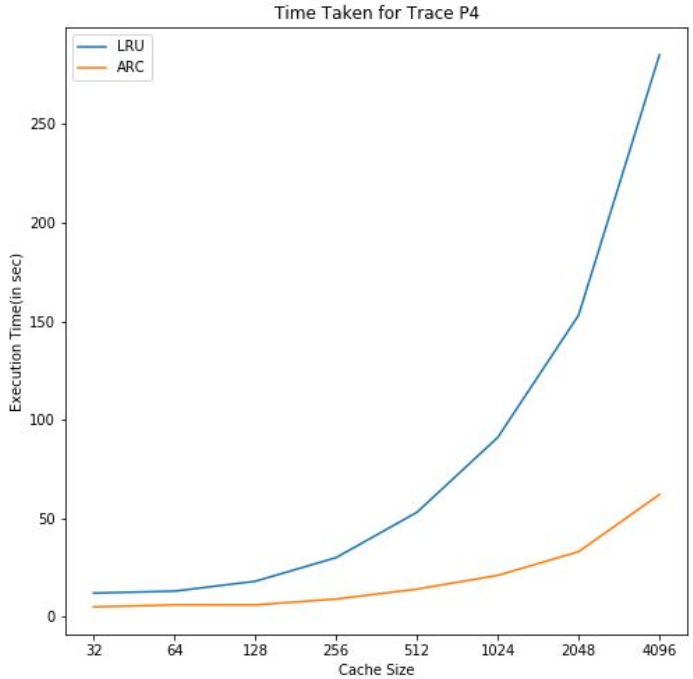
Trace P3


Hit-Ratio for Trace P3
LRU
ARC
Hit Ratio
2.0
1.5
1.0
0.5
32
64
128
256
512
1024
2048
4096
Cache Size

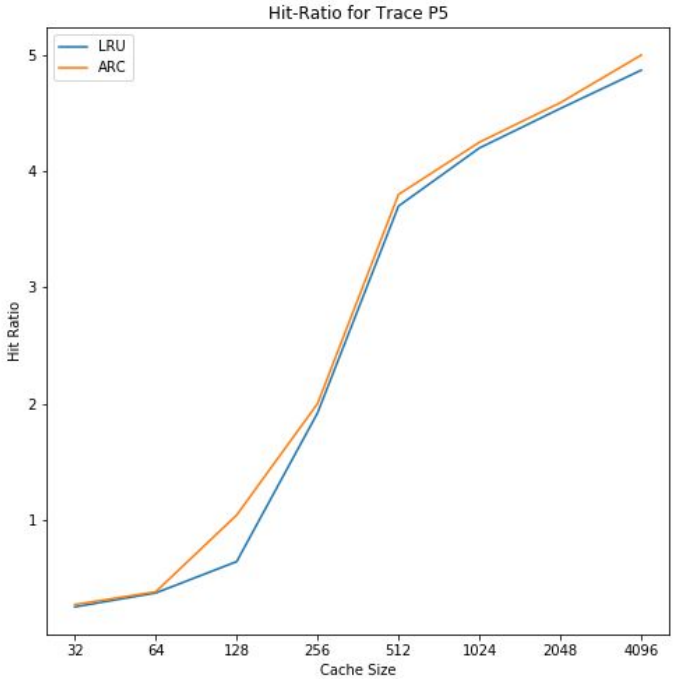

Time Taken for Trace P3
LRU
ARC
Execution Time(in sec)
60
50
40
30
20
10
0
32
64
128
256
512
1024
2048
4096
Cache Size

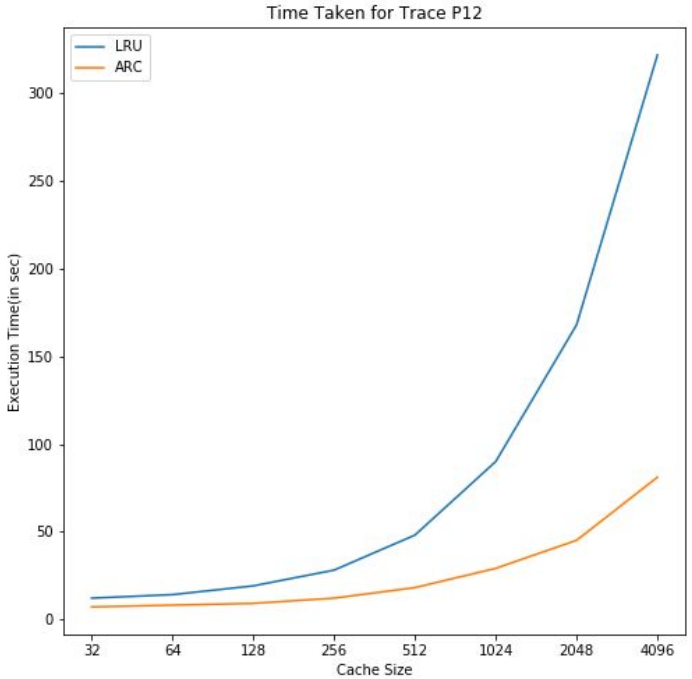
Trace P4


Hit-Ratio for Trace P4
LRU
ARC
Hit Ratio
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
32
64
128
256
512
1024
2048
4096
Cache Size

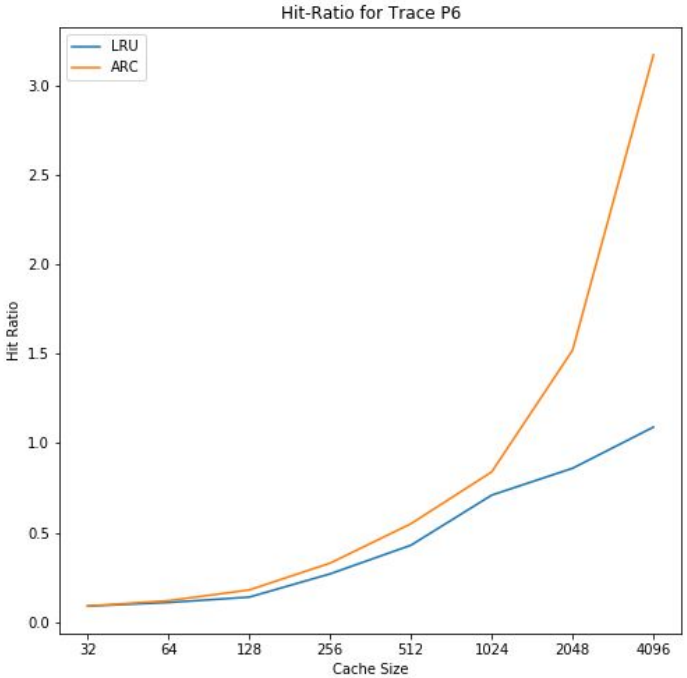

Time Taken for Trace P4
LRU
ARC
Execution Time(in sec)
0
50
100
150
200
250
32
64
128
256
512
1024
2048
4096
Cache Size

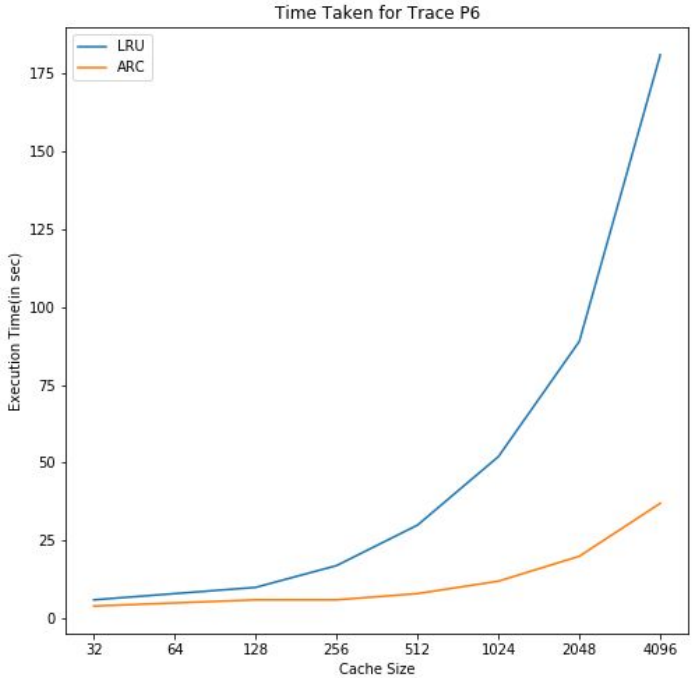
Trace P5


Hit-Ratio for Trace P5
LRU
ARC
Hit Ratio
1
2
3
4
5
32
64
128
256
512
1024
2048
4096
Cache Size

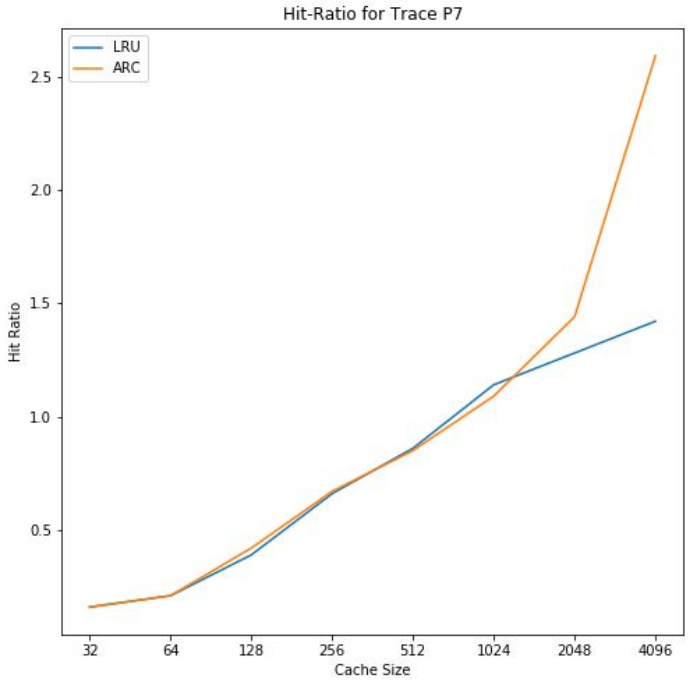

Time Taken for Trace P12
LRU
ARC
Execution Time(in sec)
0
50
100
150
200
250
300
32
64
128
256
512
1024
2048
4096
Cache Size

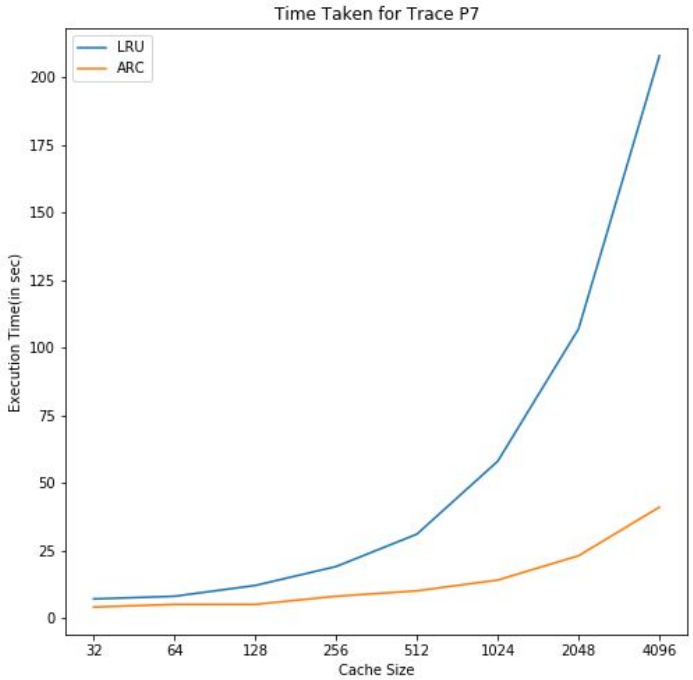
Trace P6


Hit-Ratio for Trace P6
LRU
ARC
Hit Ratio
0.0
0.5
1.0
1.5
2.0
2.5
3.0
32
64
128
256
512
1024
2048
4096
Cache Size


Time Taken for Trace P6
LRU
ARC
Execution Time(in sec)
0
25
50
75
100
125
150
175
32
64
128
256
512
1024
2048
4096
Cache Size

Trace P7


Hit-Ratio for Trace P7
LRU
ARC
Hit Ratio
2.5
2.0
1.5
1.0
0.5
32
64
128
256
512
1024
2048
4096
Cache Size


Time Taken for Trace P7
LRU
ARC
Execution Time(in sec)
200
175
150
125
100
75
50
25
0
32
64
128
256
512
1024
2048
4096
Cache Size

Observations

- Hit ratio for ARC algorithm is always greater than or equal to LRU
- Time taken by ARC is lesser than LRU and the difference increases as the size of cache increases
- ARC outperforms LRU and hence is more efficient than LRU.
- RunTime of LRU is linear to Cache size. (Exponential in graph because Cache size is scaled logarithmically)

Conclusions

- Self Tuning :
- Low Over Head :
- Scan Resistant :
- Responds online to changing access patterns.
- the self-tuning, low-overhead, scan-resistant ARC cache-replacement policy outperforms LRU
- Can work on varying real-life workloads because of its self tuning ability.

Thank You.