AMSQM: Adaptive Multiple Super-Page Queue Management

Moshe Itshak
Computer Science Department
Bar-Ilan University
Israel
izmo@cs.biu.ac.il

Yair Wiseman
Computer Science Department
The Open University
Israel
wiseman@cs.huji.ac.il

Abstract

Super-Pages have been wandering around for more than a decade. There are some particular operating systems that support Super-Paging and there are some recent research papers that show interesting ideas how to intelligently integrate them; however, nowadays Operating System's page replacement mechanism still uses the old Clock algorithm which gives the same priority to small and large pages. In this paper we show a technique that enhances the page replacement mechanism to an algorithm based on more parameters and is suitable for a Super-Paging environment.

1. Introduction

Super-Pages are an enhancement for the well-known paging concept. Super-Pages are larger pages that are pointed to by the TLB [1]. The internal memory of modern computers has been significantly increased during the last decades. However, the TLB coverage (i.e. the size of the memory that can be pointed to directly by the TLB) has been increased by a much lower factor during the same period [2]. Therefore, several new architectures like Itanium, MIPS R4x00, Alpha, SPARC and HP PA RISC support multiple page size of the frames pointed to by the TLB. In that way the memory size pointed to directly by the TLB is higher and the overhead of the page table access time is reduced. There are also some particular operating systems that support Super-Paging e.g. [3,4].

Multimedia applications typically have large portions of memory that are clustered in few areas. Such applications can benefit Super-Paging enormously [5]. Also, nowadays computers usually have large memories [6,7]; hence, larger pages can be used; however using larger pages can apparently cause a higher page fault rate. This is a well-known flaw of the Super-Paging mechanism; however the algorithm suggested in this paper does not suffer from this flaw and even utilizes the usual behavior of the paging mechanism to reduce the page fault rate. The algorithm actually makes use of the locality principle to pre-fetch base-pages that are a part of heavy used Super-pages and the results show that this pre-fetching makes the memory hit percents better.

We also aim at developing a good technique that finds the best page to be taken out when the page fault mechanism requires this in a Super-Paging environment based on all the available parameters. Here again the locality principle that the Super-paging environment induces helps us to select the victim page better, because if page's neighbors have been accessed, it can imply that the page itself might be accessed as well and it may not be a good choice to swap the page out as the common base-page algorithms would have done.

The dilemma of which page should be taken out also occurs in higher levels as well i.e. What should be in the cache and what should be pointed to by the TLB. Our algorithm can be also a good alternative for Clock in these decisions.

2. Page Replacement Algorithms

Over the years many replacement algorithms have been published e.g. [8,9,10,11,12,13,14]; however over the last decades, CLOCK [15] has been dominated page replacement algorithms.

2.1. CLOCK

The CLOCK algorithm looks at the memory pages as a circular linked list and moves around the pages like a clock hand. Each page is associated with a reference bit. This bit is set to 1 when the page is referenced. When a page fault occurs, the page which is pointed to by the hand is checked. If its reference bit is unset it will be swap out; otherwise its reference bit is unset, and the hand moves to the subsequent page. Research and experiences have shown that CLOCK is a close approximation of LRU, thus suffers from the same problems of LRU. Nevertheless, CLOCK is still

The authors would like to thank SUN Microsystems for their donation.

dominating the vast majority of OS including UNIX, Linux and Windows [16].

Some variant of CLOCK have been suggested over the years. GCLOCK [16] was published at 1992 as an expansion to CLOCK. This algorithm contains a counter to each page (instead of a reference bit), which is increased in each reference. The clock's hand checks the pages and decrements their counter value, until it finds a page with a zero value. This page is swapped out. Unlike CLOCK, GCLOCK is taking into account the frequency, thus achieves better performance.

CLOCK-Pro [17] counts for each page the number of other distinct pages accesses since its last access. This number is called "reuse distance" and a page with a larger "reuse distance" will be considered as a colder page and will be swap out before a page with a smaller "reuse distance".

2.2. ARC

We focused in the above section at CLOCK, because CLOCK dominates the Operating Systems market; however some other methods seem to suffer from two acute problems:

(i) The need for parameters tuning (e.g 2Q [9] and LRFU [10]) and/or

(ii) Non-constant complexity (e.g. LRU-K [8], LRFU [10] ,CLOCK and GCLOCK [16]).

CLOCK also has a Non-constant complexity, so we prefer to adapt more modern algorithm to the Super-Paging environment. Recently, N. Megiddo and S. Modha proposed a new "online" tunable algorithm called ARC (Stands for Adaptive Replacement Cache) [18,19,20]. The unique capability of this algorithm is its ability to adapt itself "online" according to the systems properties e.g. from the Stack Depth Distribution (SDD) model to the Independence Reference Model (IRM) and vice versa.

The main concept of ARC is having two lists of active pages (one for the frequently used pages and one for the most recent pages) and to endow the list that is performing the best with a larger memory space. The two lists that ARC maintains are variably-sized lists called L_1 and L_2. L_1 contains the pages that have been accessed only once and L_2 contains the pages that have been accessed twice or more. The algorithm always holds that $0 \leq L_1 + L_2 \leq 2C$, where C is the number of pages in the memory. L_1 consists of two buffers - T_1 which consists of the most recent pages in the memory and B_1 which consists of the history of the most recent pages that were in the memory. Similarly L_2 is partitioned into T_2 and B_2. In addition p which always holds $p \leq c$, is the automatic adaptive parameter of the algorithm which sets the target size for T_1.

The algorithm in a simplify version is for any page request:

- If the requested page is in T_1 or in T_2:
  - Move the page to the MRU of T_2.
- If the requested page is in B_1:
  - If $|B_1| \geq |B_2|$
    - $\delta_1 = 1$
  - Else
    - $\delta_1 = |B_2| / |B_1|$
  - $P = Min(P + \delta_1, C)$
  - Move the page from B_1 to be the LRU of T_2 (swap out page according to P).
- If the requested page is in B_2:
  - If $|B_2| \geq |B_1|$
    - $\delta_2 = 1$
  - Else
    - $\delta_2 = |B_1| / |B_2|$
  - $P = Max(P - \delta_2, 0)$
  - Move the page from B_2 to be the LRU of T_2 (swap out page according to P).
- If the requested page is not in $T_1 \cup T_2 \cup B_1 \cup B_2$:
  - Move the new page to be the MRU of T_1 (swap out page according to P).

As we mentioned above, CLOCK can move its clock hand over many pages, until a page with an unset bit is found. Unlike CLOCK, ARC has a constant complexity - O(1). In addition, ARC is tunable i.e. ARC can adapt itself according to the characteristics of the data that the processes use. These are the reasons why we chose to adapt ARC to the Super-Paging mechanism.

3. AMSQM

We used ARC to develop a new algorithm - Adaptive Multiple Super-Pages Queues Management (AMSQM) which is an expansion of the ARC algorithm that supports Super-Paging. AMSQM algorithm has two levels - the high level manages the different Super-Page queues (sizes and allocations); whereas the low level is the internal management of each Super-Page's queue. In addition, there is a special buffer for each Super-Page size that collects fractions of bigger Super-Pages. The purpose of these buffers is giving the demoted Super-Pages a chance to get a better priority if they are hot pages.

The suggested algorithm uses a reservation-based scheme, in which region is reserved for a super-page at the page fault time and the promotion is done when the number of the super-page's populated base pages gets to a promotion threshold. Since we would like a partially populated super-page to have the opportunity of being promoted, the decision for preempting reservation of a super-page candidate or swapping out its base-pages is taken based on the super-page "recency" in the page lists and not based on the number of currently resident base-pages that the super-page consists of.

Hardware maintains only a single reference bit; thus it is difficult to decide whether all (or at least most) of the base-pages that the super-page consists of are actually in use. Sometimes, only a small percentage of the base pages should be in the memory. Therefore, AMSQM manages several queues for each super-page size, preventing from cold super-pages to be retained in the cache occupying the space of some potential hotter smaller super-pages or base pages.

Finally, in order to wisely balance the different queues length, the algorithm counts the number of times that each page has been referenced and checks the relative "recency" of each super-page's queue.

Now we will write down the AMSQM algorithm in pseudo-code:

Let us define:

C- The memory size.

c_i - Physical size of the super (and base) pages buffers. $\Sigma c_i \leq C$.

s_i - Target size of each buffer.

Qi- Queue (FCFS) that saves demoted Super-Pages (or base-pages), which are a fraction of bigger Super-pages.

T^i_1 - The most recent pages in the memory of every Super (or base) page, which were accessed only once.

B^i_1 - The most recent pages in the history of every Super (or base) page, which were accessed only once.

T^i_2 - The most recent pages in the memory of every Super (or base) page, which were accessed more than once.

B^i_2 - The most recent pages in the history of every Super (or base page), which were accessed more than once.

P_i - Tunable parameter - the recommended size of T^i_1.

$size_i$ - Super-Page size in base pages.

$bound_i=\beta\cdot size_i/size_1$

count(x) - The number of times that Super-Page x was referenced.

$rank_i$ - Determines which queue removes an entry. $rank_i=\alpha\cdot dif_i+(1-\alpha)\cdot rec_i$, where dif_i is the difference between s_i and c_i; i. e. max(0, $size_i\cdot(c_i-s_i)$) and rec_i is the relative recency of the LRU of Super-Page i among the LRU of the other Super-Pages.

threshold - threshold for promoting a partially occupied (candidate) super-page to a fully occupied super-page.

$SP(x_j)$ - The superpage which the base page x_j belongs to. $x_j = SP(x_j)$ iff x_j does not belong to any super-page (a solitary base page).

$\omega(x)$ - The number of occupied base pages in super-page x.

α,β,γ - Parameters that should be set according to the data characteristic; where $0\leq\alpha\leq1$, $\beta\geq1$ and $0\leq\gamma\leq½$.

The algorithm AMSQM is:

```
**AMSQM**(c, Stream of base pages requests: $x_1,x_2,..,x_n$)
  • $c_1=c_2=...=c_k=0$
  • For each $x_j$
      o Call **HandleSuperPage**($x_j$,| $SP(x_j)$|)
      o If $\omega(SP(x_j))\geq threshold\cdot size_{|SP(x_j)|}$
          ▪ Promote $SP(x_j)$
  • if the access type is "write" ,recursively demote
    $SP(x_j)$ to clean base/super pages and move them
    to the suitable Q lists.
**HandleSuperPage**($x_j$,i)
  • If $SP(x_j)$ is in $T^i_1$,
      o If $x_j$ is valid
          ▪ Move $SP(x_j)$ to be the MRU of $T^i_2$
      o Else
          ▪ Fetch $x_j$ to the cache.
          ▪ Move $SP(x_j)$ to be the MRU of $T^i_1$
      o If (count($SP(x_j)$)= $bound_i$)
          ▪ count($SP(x_j)$)=$\gamma\cdot bound_i$
      o Else
          ▪ count($SP(x_j)$)= count($SP(x_j)$)+1
  • If $SP(x_j)$ is in $T^i_2$ or $Q_i$
      o If $x_j$ is invalid
          ▪ Fetch $x_j$ to the cache.
      o Move $SP(x_j)$ to be the MRU of $T^i_2$
      o count($SP(x_j)$)= count($SP(x_j)$)+1
  • If $SP(x_j)$ is in $B^i_1$
      o If the size of $B^i_1$ is at least the size of $B^i_2$
          ▪ $\delta=1$
      o Else
          ▪ $\delta=|B^i_2|/|B^i_1|$
      o $P_i=\min(P_i+\delta, c_i)$
      o Call **Release** ($x_j$,i)
      o Fetch $x_j$ to the cache.
      o Move $SP(x_j)$ to be the MRU of $T^i_2$
      o count($SP(x_j)$)= count($SP(x_j)$)+1
  • If $SP(x_j)$ is in $B^i_2$
      o If the size of $B^i_2$ is at least the size of $B^i_1$
          ▪ $\delta=1$
      o Else
          ▪ $\delta=|B^i_1|/|B^i_2|$
      o $P_i=\max(P_i-\delta,0)$
      o If count($SP(x_j)$)$\leq 2\cdot\gamma\cdot bound_i$
          ▪ Call **Release** ($x_j$,i)
          ▪ Fetch $x_j$ to the cache.
          ▪ Move $SP(x_j)$ to be the MRU of $T^i_2$
      o If count($SP(x_j)$)$>2\cdot\gamma\cdot bound_i$
          ▪ If $0\leq C-\sum c_i<size_i$
              • Call **IncreaseBuffer** ($x_j$,i)
              • If we couldn't allocate a continuous of $size_i$
                  • Call **Release** ($x_j$,i)
          ▪ Else
                  • Call **Allocate** ($x_j$,i)
          ▪ Count($x_j$)=$\gamma\cdot bound_i$
```

```
            ▪ $s_i = s_i + 1$
    • If $SP(x_j)$ is not in $T^i_1$, $T^i_2$, $B^i_1$ or $B^i_2$
        o If (i>1) and ($SP(x_j)$ has ever been in lists $B^i_1$ or $B^i_2$)
            ▪ Call Demote ($x_j$,i)
        o Else
            ▪ If $0 \le C - \sum c_i < size_i$
                • If ($|Q_i|+|T^i_1|+|B^i_1|=c_i$)
                    o If ($|Q_i|+|T^i_1|<c_i$)
                        ▪ Remove the LRU of $B^i_1$
                        ▪ Call **Release** ($x_j$,i)
                    o Else
                        ▪ Remove the LRU among $Q_i$ and $T^i_1$.
                • Else
                    o If ($|Q_i|+|T^i_1|+|B^i_1|+|T^i_2|+|B^i_2|>c_i$)
                        ▪ If ($|Q_i|+|T^i_1|+|B^i_1|+|T^i_2|+|B^i_2|=2\cdot c_i$)
                            ▪ Remove the LRU of $B^i_2$
                        ▪ Call **Release** ($x_j$,i)
                • Fetch $x_j$ to the cache.
                • Move $SP(x_j)$ to be the MRU of $T^i_1$
            ▪ Else
                • Call **Allocate** ($x_j$,i)
**IncreaseBuffer** ($x_j$,i)
    • Do until $size_i$ base-pages are released:
        o r=max $rank_i$
        o Remove LRU among $T^r_1$, $T^r_2$ and $Q_r$
        o $c_r = c_r - 1$
        o If ($c_r < s_r$)
            ▪ $s_r = s_r - 1$
    • Call **Allocate** ($x_j$,i)
**Release** ($x_j$,i)
    • If (($|T^i_1|>P_i$) or ($|T^i_1|=P_i$ and $x_j$ is in $B^i_2$)
        o Take the LRU page between the LRU of $T^i_1$ and the LRU of $Q_i$ and put it as the MRU of $B^i_1$.
    • Else
        o Take the LRU page between the LRU of $T^i_2$ and the LRU of $Q_i$ and put it as the MRU of $B^i_2$.
**Allocate** ($x_n$,i)
    • If there is a contiguous empty space of $size_i$ in the cache
        o Fetch $x_j$ to the cache.
        o Move $SP(x_j)$ to be the MRU of $T^i_2$
        o $c_i = c_i + 1$
**Demote** ($x_n$,i)
    • Cancel $SP(x_j)$
    • If(i>1)
        o $Dsize = size_{i-1}$
    • Else
        o $Dsize$=1
    • $free$=The biggest available continuous empty space of maximum $Dsize$.
    • if ($free$>0)
        o Create superpage $x'_j$ of size $free$ which must contain $x_j$
        o Move $x'_j$ to the MRU of $Q_{free}$.
    • Else
        • Call **Release**($x_j$,1).
        • Fetch $x_j$ to the cache.
        • move $x_j$ to the MRU of $Q_1$.
```

4. Results

We implemented the standard CLOCK algorithm, the ARC algorithm and the AMSQM algorithm. We used Valgrind [21] to capture the pages that were used by some of the SPEC – cpu2000 [22]. The SPEC manual explicitly notes that attempting to run the suite with less than 256Mbytes of memory will cause a measuring of the paging system speed instead of the CPU speed. This suits us well, because our aim is precisely to measure the paging system speed; hence, we simulated a machine with just 128MB of RAM, although it is obviously a very small memory.

The sizes of the Super-pages that we used were 8 KB, 16 KB, 32 KB, 64 KB, 128 KB and 256 KB. We assumed a tagged TLB of 32 entries for instructions and 64 entries for data.

According to experiments that we do not have enough space to detail in this paper, we found that AMSQM gives the best results if its parameters are set to the following values:

- $\alpha=0.5$
- $\beta=4$
- $\gamma=0.25$
- threshold=0.5

Both AMSQM and ARC outperform CLOCK by all the parameters in our simulation, so we found no point in presenting the results of CLOCK; therefore, the results presented here are only the ratio between strict ARC and AMSQM.

Let us define:

n - Number of memory requests by the benchmark.

p - Number of pages that the benchmark accesses.

tm_{ARC} - Number of TLB misses when ARC is the replacement algorithm.

tm_{AMSQM} - Number of TLB misses when AMSQM is the replacement algorithm.

pf_{ARC} - Number of the benchmark's page faults when ARC is the replacement algorithm.

pf_{AMSQM} - Number of the benchmark's page faults when AMSQM is the replacement algorithm.

tm_ratio=1-((tm$_{AMSQM}$-p)/(tm$_{ARC}$-p))
pf_ratio=1-((pf$_{AMSQM}$-p)/(pf$_{ARC}$-p))

The TLB misses are shown as the ratio between the TLB misses that AMSQM produces and the TLB misses that ARC produces.When a page is accessed at the first time, any algorithm will have to induce a TLB miss and obviously there is no way to eliminate this TLB miss, so we calculated only the TLB misses of the pages just from the second time they are accessed. The page faults are shown also as the ratio between the page faults that AMSQM produces and the page faults that ARC produces counting for each page only the second and further accesses.

tm_ratio and pf_ratio are the values that represent the calculation of the TLB miss ratio and the page fault ratio respectively.

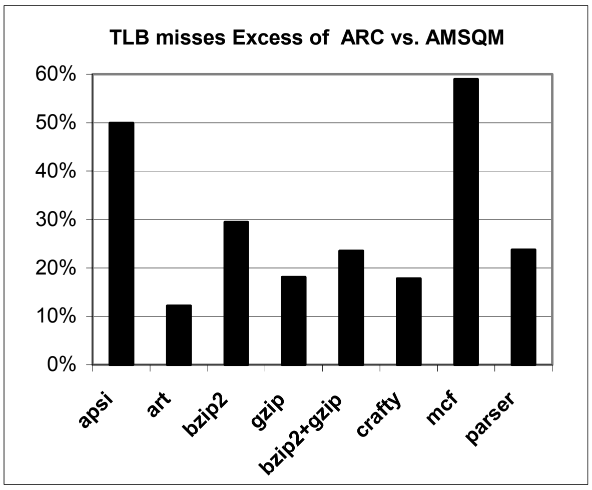

Figure 1. The TLB miss reduction of AMSQM

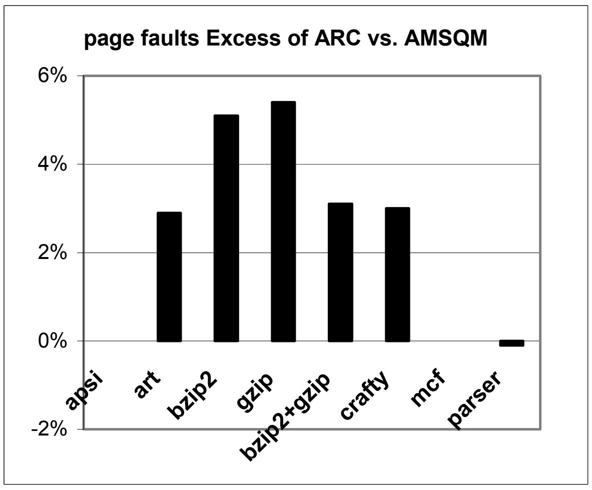

Figure 2. The page fault reduction of AMSQM

Figure 1 shows the tm_ratio of several selected SPEC2000 benchmarks whereas Figure 2 shows the pf_ratio of the same SPEC2000 benchmarks. It can be clearly seen in Figure 1 that AMSQM achieves a higher TLB ratio, because of the super-pages usage.

Furthermore, AMSQM memory hit ratio is also higher than ARC memory hit ratio in most of the benchmarks as can be noticed in Figure 2. The improvement of the memory hit ratio is because AMSQM takes advantage of the locality principle as is mentioned above in the introduction section. The other SPEC benchmarks show similar results, so we do not include these benchmarks in this paper.

We also tested the running of both of the algorithms using the subroutine "clock()" in "time.h" of GNU C compiler. We found the results quite similar, so we do not include these results in this paper as well.

5. Conclusions and Future Work

The new adaptive super-page replacement algorithm AMSQM has been presented. We have shown that AMSQM usually achieves a higher TLB coverage than ARC and also a better page fault ratio in most of the benchmarks we have used.

This paper shows another important aspect of the Super-Paging environment. We believe operating systems have an improper attitude toward the Super-Page replacement algorithm selection. They usually just copy the old algorithms of the traditional paging mechanism with no attention to the new Super-Paging environment. This brings about an improvement of the hardware support for a smaller TLB miss ratio, but the software support for a smaller TLB miss ratio is considerably poorer.

We show a way to adapt one of the most recent algorithms to the Super-Paging environment with the aim of obtaining a better TLB hit ratio.

In the future we would like to find ways to set the AMSQM parameters (α,β,γ) dynamically. In our experiments we have found that the values we used for these parameters are the best for most of benchmarks; however, there is a minority of benchmarks that have a preference of other values and there are also few benchmarks that will have a preference of adaptively modified values. Therefore, we believe that adaptively modified values can improve the performance of several benchmarks.

In addition, we would like to find a pattern for super-pages reoccurrence. Such a pattern can improve the efficiency of the super-page promotion decisions. The traditional threshold parameter seems to be not enough for taking the most beneficial decision.

We would like also to integrate the suggested patch into the Linux kernel. The current results are encouraging and they support our belief that the new replacement algorithm can significantly enhance the memory management mechanism in the two above mentioned manners: better TLB hit ratio and fewer page faults.

6. References

[1] Y. A. Khalidi, M. Talluri, M. N. Nelson and D. Williams. Virtual memory support for multiple page sizes. In Proceedings of the Fourth IEEE Workshop on Workstation Operating Systems, Napa, California, October 1993.

[2] J. Navarro. Transparent operating system support for superpages, Ph.D. Thesis, Department of Computer Science, Rice University, April 2004.

[3] N. Ganapathy and C. Schimmel. General purpose operating system support for multiple page sizes. Proceedings of the USENIX Annual Technical Conference, New Orleans, 1998.

[4] Subramanian, C. Mather, K. Peterson, and B. Raghunath. Implementation of multiple pagesize support in HP-UX. Proceedings of the USENIX Annual Technical Conference, New Orleans, 1998.

[5] Abouaissa H., Delpeyroux E., Wack M. and Deschizeaux P., "Modelling and integration of resource communication in multimedia applications with high constraints using hierarchical Petri nets", Proceedings of IEEE International Conference on Systems, Man, and Cybernetics (SMC-99), pp. 220-225, vol. 5, Tokyo, Japan, 1999.

[6] R. F. Wallace, R. D. Norman and E. Harari, "Computer memory cards using flash EEPROM integrated circuit chips and memory-controller systems", US Patent no. 7106609, 2006.

[7] Geppert L., "The New Indelible Memories", IEEE SPECTRUM, Vol. 40, Part 3, pp. 48-54, 2003.

[8] E. O'Neil, P. O'Neil and G. Weikum, The LRU-K Page Replacement Algorithm for Database Disk Buffering, Proceedings of SIGMOD `93, Washington, DC, May 1993.

[9] T. Johnson and D. Shasha, 2Q: a low overhead high performance buffer management replacement algorithm, Proceedings of the Twentieth International Conference on Very Large Databases, VLDB' 94, Santiago, Chile, pp. 439-450, September 1994.

[10] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies, IEEE Trans. Computers, vol. 50, no. 12, pp. 1352-1360, 2001.

[11] J. Kim, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, and C. Kim, A Low-Overhead, High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References, 4th Symposium on Operating System Design and Implementation, San Diego, California, pp. 119-134, October 23-25, 2000.

[12] S. Jiang and X. Zhang, LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance, In Proceeding of 2002 ACM SIGMETRICS, Marina Del Rey, California, pp. 31-42, June 15-19, 2002.

[13] Y. Smaragdakis, S. Kaplan, and P. Wilson, The EELRU adaptive replacement algorithm, Performance Evaluation (Elsevier), Vol. 53 , No. 2, pp. 93-123, July 2003.

[14] Y. Zhou, Z. Chen and K. Li. "Second-Level Buffer Cache Management", IEEE Transaction on Parallel and Distributed Systems (TPDS), Vol. 15, No. 7, pp. 505-519, 2004.

[15] F. J. Corbato, .A Paging Experiment with the Multics System.,MIT Project MAC Report, MAC-M-384, May 1968.

[16] M. B. Friedman, Windows NT Page Replacement Policies, Proceedings of 25th International Computer Measurement Group Conference, pp. 234-244, December 1999.

[17] S. Jiang, F. Chen and X. Zhang, CLOCK-Pro: an Effective Improvement of the CLOCK Replacement, Proceedings of 2005 USENIX Annual Technical Conference, pp. 323-336, Anaheim, California, April 2005.

[18] N. Megiddo and D. S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," Proc. of the 2nd USENIX Conference on File and Storage Technologies (FAST'2003), San Francisco, pp. 115-130, March 31 - April 2, 2003.

[19] N. Megiddo and D. S. Modha, "One Up on LRU" ;login: - The Magazine of the USENIX Association, vol. 28, no. 4, pp. 7-11, August 2003.

[20] N. Megiddo and D. S. Modha, "Outperforming LRU with an Adaptive Replacement Cache Algorithm," IEEE Computer, pp. 4-11, April 2004.

[21] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation", Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), San Diego, California, USA, June 2007.

[22] SPEC (2000) CPU-2000. Standard Performance Evaluation Corporation, Warrenton, Virginia, http://www.spec.org/.