Buffering of Index Structures

Tuba Yavuz-Kahveci and Tamer Kahveci and Ambuj Singh

Department of Computer Science University of California
Santa Barbara, CA 93106

ABSTRACT

Buffering of index structures is an important problem, because disk I/O dominates the cost of queries. In this paper, we compare existing algorithms for uniform, nonuniform static and nonuniform dynamic access patterns. We experimentally show that the LRU-2 method is better than the other methods. We also propose an efficient implementation of the LRU-2 algorithm. In the second part of the paper, we propose a new buffering algorithm for a distributed system where each machine has its own buffer. We show experimentally that this method performs better than other buffering techniques.

Keywords: buffering, index structure

1. INTRODUCTION

Buffering large index structures is one of the key factors that determine the performance of database management systems. This is because disk I/O still remains the bottleneck in answering queries. In order to speed up the query executions, disk I/O should be kept to a minimum. This can be achieved by effective use of the buffer space, keeping "important" pages of the index structure in the buffer. If the size of the buffer is larger than the total size of the target pages, then all of the target pages can be kept in the buffer. In this case, a disk page read occurs only for the first reference to that page. However, if the buffer size is less than the total size of the target pages, only a subset of the target pages can be stored in the buffer. In this case, if the referenced page is not in the buffer, one of the pages in the buffer must be replaced with the new page.

A good page replacement algorithm must adapt to different page access patterns. The most common probability distributions for page accesses are *uniform*, *non-uniform static*, and *non-uniform dynamic* access patterns. In a uniform page access pattern, all the pages are accessed with the same probability. Therefore, it is difficult to decide the page to be evicted for this kind of access patterns. In a non-uniform static access pattern, the probability of access can be different for different pages, but these probabilities are invariant with respect to time. A good page replacement algorithm should learn these probabilities and give higher priorities to the pages with higher access probabilities. In a non-uniform dynamic access pattern, the access probabilities for the pages can be different, and these probabilities can also change in time. A good page replacement method should learn the access probabilities of the pages and it should adapt to the changes quickly when these probabilities change.

The optimal page replacement algorithm called B_0[1] requires detailed knowledge of the future page accesses, which is generally impossible. Therefore, current page replacement algorithms estimate the future references by keeping history of the past references. The most popular page replacement algorithm, LRU (Least Recently Used) simply chooses the page which has not been accessed for the longest time as the replacement victim. Two improvements to the LRU method are proposed by Sacco.[2] These are ILRU (Inverse LRU) and OLRU (Optimal LRU). ILRU works similar to the LRU method, but it places the pages in the reverse order to guarantee a longer buffer life for higher levels of the index tree. The OLRU method estimates the number of pages that will be accessed at each level of the index and allocates different parts of the buffer to different levels of the index according to these estimated numbers. O'neil et al.[3] propose to use statistical information by keeping the last K accesses to all the pages. They call this method the LRU-K method.

In a multi-processor systems, each processor has its own buffer. Furthermore, the access pattern for different processors may have different characteristics. Therefore, keeping a global page access history decreases the performance

Further author information: (Send correspondence to T.K.)
T.Y.K.: E-mail: tuba@cs.ucsb.edu
T.K.: E-mail: tamer@cs.ucsb.edu
A.S.: E-mail: ambuj@cs.ucsb.edu

of the page replacement algorithms. A solution is to run a buffering algorithm for each processor independently. In this case each processor keeps its own access history. We propose a new technique called *Windowed LRU* for this purpose.

The rest of the paper is organized as follows. In Section 2, we present previous research on buffering. In Section 3, we present experimental results. We discuss implementation details for uni-processor and multi-processor systems in Section 4, and we conclude in Section 5.

2. PREVIOUS WORK

Buffering of pages has been addressed by many authors. Most of these works assume *independent reference model*. This model assumes that the access probabilities of the pages are independent of each other.

The optimal page replacement algorithm, called B_0[1] chooses the victim as the page which will be referenced the latest among the ones in the buffer. O'neil et al.[3] call this technique "optimal strategy with an oracle", because this method requires detailed knowledge of the future refererences. We call this technique the *optimal offline algorithm*, because generally the detailed access pattern can not be known in advance in real applications.

O'neil et al.[3] define an algorithm called A_0. They describe this technique as "optimal strategy with probabilistic information but without an oracle". The algorithm assumes that the access frequency of all the pages in the database are known in advance. Using the frequency information, the algorithm computes the expected access time for all the pages in the buffer and chooses the one which is expected to be accessed the latest as the victim. We call this technique the *optimal online algorithm*, because if the hot regions on the disk are known, the probability distribution for the pages can be computed. The expected access time for a page is computed as follows: Let $\mathcal{P} = \{p_1, p_t, ..., p_d\}$ be the set of pages on the disk and let $R = r_1 r_2 ... r_i ...$ be the reference string such that $r_i = p_j$ for some $1 \leq j \leq d$. The probability that the i^{th} referenced page is p_j is represented by $Pr(r_i = p_j) = \beta_j$. The probability that the k^{th} access will be the first access to the page p_j is $(1 - \beta_j)^{k-1}\beta_j$. The expected value for k can be computed as $lim_{k\to\infty} \sum_{i=1}^{k} (1 - \beta_j)^{k-1}\beta_j/k$ which is equal to $1/\beta_j$. This algorithm is *online* if the hot regions on the disk are known in advance. The access probability, p_j can be approximated as a function of its location on the index. Let the page p_j be located at the l^{th} level of the index, then the probability of access for p_j can be computed as:

$$\beta_j = P_{hot} \times V_{j,hot} + (1 - P_{hot}) \times (V_j - V_{j,hot})$$

where P_{hot} is the probability that a range query will be in the hot region, V_j is the volume of the page p_j, and $V_{j,hot}$ is the volume of the intersection of p_j with the hot region.

Other page replacement algorithms estimate the future references of the pages by keeping a history of the past references. One of the earliest page replacement algorithms is LRU. This method assumes that in most applications there is temporal locality in the access pattern. That is, if a page is accessed recently, then probably it will be referenced again soon. This algorithm chooses the page which has not been accessed for the longest time as the victim. The algorithm guarantees that a new referenced page will remain in the buffer until all other pages in the buffer are replaced or referenced again. If the accessed page is in the cold region, the performance of the technique decreases, because a hot page in the buffer can be selected as the victim before this cold page. Furthermore, this method is too sensitive to changes in the frequency of the access pattern.

Sacco[2] proposed two modified versions of the LRU method assuming a B+ tree index structure. These are ILRU and OLRU. Both of these methods are variations of the LRU method. They give higher priority to the pages in the higher levels of the index (the root page is of the highest priority and will always be in the buffer). To achieve this, ILRU places a page p at level i to the i^{th} place in the LRU stack. The root page always stays at position 0. If the number of pages in the buffer is less than i, then page p is put on top of the LRU stack. The page at the top of the stack is chosen for replacement. ILRU guarantees that a page will not be removed from the buffer until all of its children in the index are removed from the buffer. However, ILRU does not make any distinction between the pages at the same level.

The OLRU method assigns different LRU buffers to each level of the index structure. The estimate for the number of pages that will be accessed by a query at the i^{th} level of the index is represented by T_i. The value of T_i is calculated by Yao's function.[4] If size of the buffer B is greater than or equal to the sum of sizes of the LRU stack sizes requested, then each level is assigned an LRU stack with the required size. However, if the buffer is not large

enough, then separate LRU stacks are allocated for the first k levels where $\sum_{i=0}^{k} T_i < B$. Level $k+1$ gets an LRU stack of size $B - \sum_{i=0}^{k} T_i - 1$. For the pages at levels $k+2$ or higher, a coalesced region of a single page is used. When a page p at level i is accessed, if there is a private buffer for level i, that buffer is used; otherwise, p is kept in the coalesced region. There are several problems with the OLRU method: First, OLRU uses the LRU method for replacement for the pages at the same level of the index. This means that it distinguishes between the pages at the same level based on the recency of their access. Since LRU has no knowledge of access pattern, it may not be the suitable method for comparing priorities of two pages at the same level. Another deficiency of OLRU is that the function it uses for estimating the number of pages that will be accessed at different levels does not involve the size of the buffer. In the case of a large fanout and a large index tree height, only a few high levels will get a separate LRU stack and the other levels will share a single page for buffering.

Leutenegger et al.[5] consider the effect of pinning top levels of the index using R-tree as the index structure. The Pinning method keeps the pages in the higher levels of the index tree in the buffer. If a level does not fully fit into the buffer, pages to be buffered are selected randomly. Access to the lower levels translates directly into a disk access since the content of the buffer does not change. For the level that does not fully fit into the buffer, random selection of the pages may leave some frequently accessed pages out of the buffer.

Another derivative of the LRU method, called *LRU-K* is presented by O'neil et al..[3] The LRU-K method keeps timestamps for the last K accesses to all the pages. Initially all the timestamps for that page are set to $-\infty$. If the buffer is full when a new page is referenced, the page whose last K^{th} access is the earliest is selected as the victim. If there is at least one page whose K^{th} access time is $-\infty$, then among these pages, the page whose $(K-1)^{th}$ access is the earliest is selected as the victim, and so on. If $K = 1$, this method reduces to LRU. If $K > 1$, this method turns out to be a more stable version of *LRU*. This is because it uses statistical information from the last K accesses. For $K = 2$, the method performs very close to the A_0 method. According to the experimental results presented in the same paper, the gap between the optimal online algorithm A_0 and LRU-2 is very small and increasing the value of K does not increase the performance of the method much. For small values of K, the LRU-K algorithm adapts faster to the changes in the access patterns. For large values of K, the LRU-K algorithm becomes less sensitive to sudden changes in probability distribution. This is because the probability that the last K accesses contain values from different frequencies increases as K increases. According to experimental results, the method performs the best when $K = 2$ or $K = 3$.

3. EXPERIMENTAL RESULTS

We conducted several experiments in order to compare performance of LRU, LRU-2, ILRU, OLRU, Pinning, A0, and the Optimal algorithm. We ran the experiments on a Sun-Ultra Sparc machine running Solaris v2.6 operating system. We used two datasets in our experiments: a synthetic dataset and a stock market dataset. Both datasets consist of 100,000 data points. We reduced the number of dimensions to 2 by choosing the first 2 dimensions of each data. The synthetic dataset is distributed uniformly over the domain space. On the other hand, the stock market dataset is clustered. We used R^*-trees[6–8] as the index structure. We set the size of a page to 2K.

We posed range queries and retrieved the pages whose bounding boxes intersect the query rectangle. We considered three different access patterns. These are uniform, non-uniform static, and the non-uniform dynamic access patterns. Each of these access patterns accessed 30,000 pages. For the uniform access pattern, we set the likelihood of access for each page as equal. For the non-uniform static access pattern, we selected a *hot region* in the domain as follows: We randomly selected an interval for each dimension such that the length of that interval is one-fifth of the length of that dimension in the whole domain. We defined the box that tightly covers the set of points within the intersection of these intervals as the hot region. The rest of the domain is called the *cold region*. We generated 80% of the queries from the hot region and 20% of the queries from the cold region. The queries within the hot and cold regions are generated uniformly. For the non-uniform dynamic access pattern, we divide the whole domain space into 25 equal sized boxes by dividing each dimension into 5 equal intervals. We also divided the reference string into 4 equal size parts. We call each of these parts a *phase*. Each phase consists of 7,500 page references. For each phase, we randomly chose one of the rectangles within the domain as the hot region and the rest as the cold region. Similar to the non-uniform static access pattern, we constructed 80% of the page references to the hot region and 20% of the references to the cold region. The queries within hot and cold regions are generated uniformly.

The experimental results for the synthetic dataset are presented in Figures 1 to 3. These figures represent the percentage of the page references that cause disk I/O for different buffer sizes. Figure 1 corresponds to the uniform

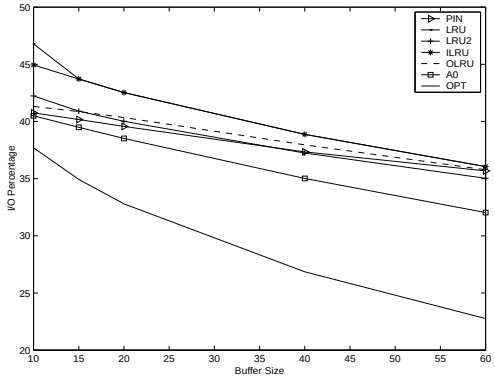

Figure 1. Percentage of I/O for uniform access pattern on synthetic data

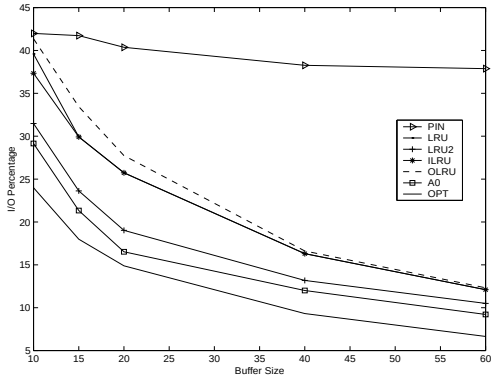

Figure 2. Percentage of I/O for non-uniform static access pattern on synthetic data

page access pattern, Figure 2 corresponds to the non-uniform static access pattern, and Figure 3 corresponds to the non-uniform dynamic access pattern. Similarly, the experimental results for the stock market dataset are presented in Figures 4 to 6. In all these figures, the optimal algorithm performs the best and A_0 performs the second best. In all these figures, ILRU performs almost the same as LRU. Some important notes about the experiment results for the synthetic dataset are as follows:

- The performance of LRU-2 gets closer to that of LRU as the buffer size increases. This can be explained as follows: As the buffer size increases, the expected time that a page is stored in the buffer increases. Therefore, the probability that a page (especially a hot one) will have a second reference before it is evicted increases.

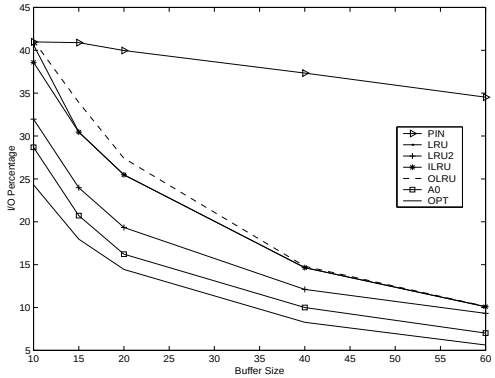

Figure 3. Percentage of I/O for non-uniform dynamic access pattern on synthetic data

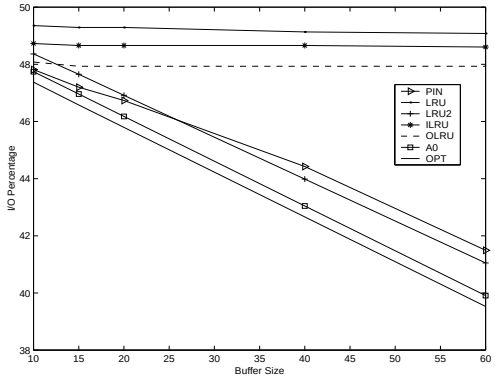

Figure 4. Percentage of I/O for uniform access pattern on stock market data

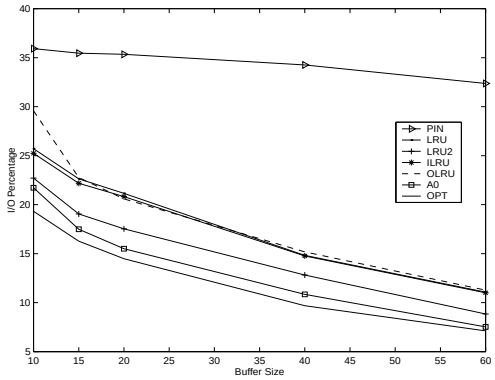

Figure 5. Percentage of I/O for non-uniform static access pattern on stock market data

- All the methods perform the worst for the uniform reference pattern. This is is because the frequencies of all the pages are similar. Therefore, the methods can not distinguish the pages.
- The gap between the optimal method and the A_0 method is the maximum in uniform access pattern. The optimal method knows all the future references. However, the A_0 method only knows the frequencies. Therefore the A_0 method gives the same priority to all the pages. A random page replacement algorithm is expected to perform as well as the A_0 method for uniform distribution.

Since stock market data is highly clustered the access frequencies of the pages are not uniform for uniform range queries. Some important notes about the stock dataset are as follows:

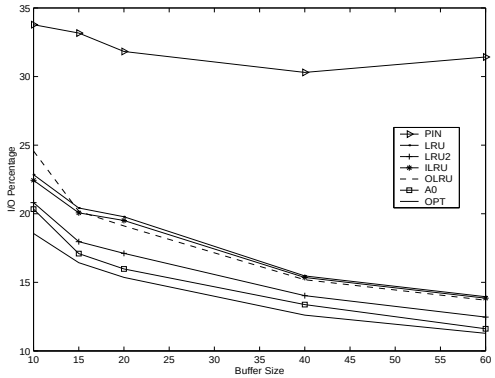

Figure 6. Percentage of I/O for non-uniform dynamic access pattern on stock market data

Let $\mathcal{P} = \{p_1, p_2, ..., p_d\}$ be the pages on the disk and $t_{p_i,1}, t_{p_i,2}, ..., t_{p_i,K}$ be the timestamps for page p_i. Let B be the buffer.

$t_{p_i,j} := -\infty$ for all $1 \leq i \leq d$ and $1 \leq j \leq K$
For $i := 1 : \infty$
  Access page $r_i = p_j$
  If $p_j \notin B$ then
    If B is full then
      $q :=$ EXTRACT_MIN(B)
      Write the page q on to the disk
    INSERT(B, p_j)
  $t_{p_j,k+1} := t_{p_j,k}$ for all $1 \leq k \leq K-1$
  $t_{p_j,1} := t_{now}$

Figure 7. LRU-K implementation for uni-processor systems

- The percentage of disk I/O for the LRU, ILRU, and OLRU methods does not decrease much as the buffer size increases in uniform range queries. This is because the data is clustered and if a range query is performed on a dense region, then this will cause a high number of page references. Therefore, the expected number of pages referenced between two consecutive references of the same page increases. This increases the probability that a page will be evicted from the buffer before it is referenced again.

- For non-uniform query patterns, since the hot region is very small compared to the whole domain, the probability that a hot region will be in a dense region is small. Therefore, unlike the uniform query pattern, the percentage of disk I/O for the LRU, ILRU and OLRU methods decrease as the buffer size increases.

4. IMPLEMENTATION OF LRU-K

In this section we present the implementation details for the LRU-K algorithm for uni-processor and multi-processor case.

4.1. Uni-processor Implementation

In uni-processor systems, we assume that there is only one client that performs queries. The algorithm works as follows: Each page contains $K-1$ timestamps to store the last $K-1$ access times to that page. These values are initially assigned to $-\infty$. If the new accessed page is already in the buffer, the last $K-1$ access times are shifted up and the last access time is set to the current time. If the new accessed page is not in the buffer and if there is available space in the buffer, then the page is stored in the next available place in the buffer, the last $K-1$ access times to that page are restored using the time information in that page, and the last access time is set to the current time. If the new accessed page is not in the buffer and if the buffer is full, then new page is replaced with the victim page. The victim is selected as follows: If the K^{th} access time of all the pages in the buffer are different from $-\infty$, then the page with the smallest K^{th} access time is selected as the victim. If the K^{th} access time of some of the pages are $-\infty$, then among these pages the one with the smallest $(K-1)^{th}$ access time is selected similarly. Since at least one access time of the pages in the buffer are guaranteed to be different than $-\infty$ and the access times of all the pages are different, the method guarantees to find a unique victim.

Given a page p, the reference vector t_p of p is defined as $t_p =< t_{p,1}, t_{p,2}, ..., t_{p,K} >$. The reference vector of the page p is less than the reference vector of the page q (i.e $t_p < t_q$) if and only if $t_{p,j} < t_{q,j}$ and $t_{p,i} \leq t_{q,i}$ for $j < i \leq K$. The victim page can be selected efficiently using a heap. The root of the heap stores the page with the smallest reference vector. The algorithm for the LRU-K method is given in Figure 7.

4.2. Multi-processor Implementation

In multi-processor systems, we consider the case when there is more than one processor each having its own buffer. The pages can be distributed to the processors or they can be stored on a single disk. Multi-processor systems contain more than one client accessing the pages. These clients can be distributed over the network. The page access patterns of these clients can be independent of each other. Some of the processors can uniformly access the pages while other access non-uniformly. Furthermore, the access patterns of the processors can change independent of

Let $\mathcal{P} = \{p_1, p_2, ..., p_d\}$ be the pages on the disk and $t_{p_i,1}, t_{p_i,2}, ..., t_{p_i,K}$ be the timestamps for page p_i. Let B be the buffer.

For $i := 1 : \infty$
  Access page $r_i = p_j$
  If $p_j \notin B$ then
    If $p_j \in W$ then
      Restore $t_{p_j,1}, t_{p_j,2}, ..., t_{p_j,K-1}$ from W
    else
      Set $t_{p_j,1}, t_{p_j,2}, ..., t_{p_j,K-1}$ to $-\infty$
    If B is full then
      $q :=$ EXTRACT_MIN(B)
      Insert $< t_{q,1}, t_{q,2}, ..., t_{q,K-1} >$ to W
    INSERT(B, p_j)
  $t_{p_j,k+1} := t_{p_j,k}$ for all $1 \leq k \leq K-1$
  $t_{p_j,1} := t_{now}$

Figure 8. LRU-K implementation for multi-processor systems

each other. Therefore a good buffering algorithm for the multi-processor systems must adapt to the access pattern of each processor separately, and it must not be affected by the page requests of other processors. A centralized buffering algorithm does not work well for multi-processor systems, because centralized buffering algorithms are not sensitive to the access patterns of individual processors. Therefore, each processor should run its own page replacement algorithm. The LRU, ILRU and OLRU methods can easily be adapted to the multi-processor systems, because there is no need to save any information about pages not in the buffer. However, this is not true for the LRU-K algorithm, since the last $K-1$ accesses need to be saved for any page. But this information is processor specific, and maintaining a single timestamp vector across all processors is not possible. One idea can be to keep $K-1$ timestamps separately for each of the processors. When a processor references a page, it only updates its timestamps. This scheme has a large space overhead: the amount of space needed for the timestamps increases linearly with the number of processors. This means that the space used to store the data itself within a page decreases as the number of processors increases. Therefore, the total number of pages required to store both the index and the database increases. As a result of this, the number of page references for a range query increases leading to higher page faults. In this section, we explain how the LRU-K algorithm can be adapted to the multiprocessor systems without suffering from the above problems. We call this technique the *WLRU-K* (Windowed LRU-K) algorithm.

In the WLRU-K method, each buffer stores a window W for the history of the last $K-1$ access times for a set of pages. Unlike the uni-processor implementation, the access times for the pages are not kept as a part of the page. If the newly accessed page is not in the buffer, first the page is searched in W. If it exists in W, then the WLRU-K method restores the last $K-1$ access times using the information in W and sets the last access time of that page to current time. If the newly accessed page does not exist in W, then the last $K-1$ access times are set to $-\infty$. The victim page is selected the same as the LRU-K method. That is, the page whose last K^{th} access time is earliest is selected as the victim. Whenever a page is evicted form the buffer, its timestamps for the last $K-1$ accesses are inserted into W. The window W is handled using the *FIFO* (First In First Out) method.

The size of the window is an important parameter to the WLRU-K method. Increasing the size of the window increases the number of pages whose history is available to the algorithm, but it decreases the amount of useful space in the buffer. In our experiments, we set the window size equal to the buffer size. The algorithm for the WLRU-K method is presented in Figure 8.

We ran experiments on a uni-processor system to compare the performance of the WLRU-2 method with other methods. The experimental results for the WLRU-2 method on the synthetic data are presented in Figure 9 to 11. According to these results, WLRU-2 method performs as well as the LRU-2 method for all three reference patterns. Two important observations follow from these results:

- The WLRU-K method is the best method for multi-processor systems, because the LRU-K method is not applicable to multi-processor systems and WLRU-K performs better than other methods.

- The WLRU-K method may also be better than LRU-K for the uni-processor systems. The LRU-K method stores the timestamps within the page. The timestamps of the page are updated whenever it is referenced. Therefore, reference to a page changes the contents of that page. This means that a page becomes dirty after a reference to that page in the LRU-K method. Therefore, when a page is evicted from the buffer, it must be written back to the disk. On the other hand, since the timestamps are stored in the buffer in the WLRU-K method, the pages are not modified at each access. Therefore, the number of disk writes for the WLRU-K method is less than that of that for the LRU-K method.

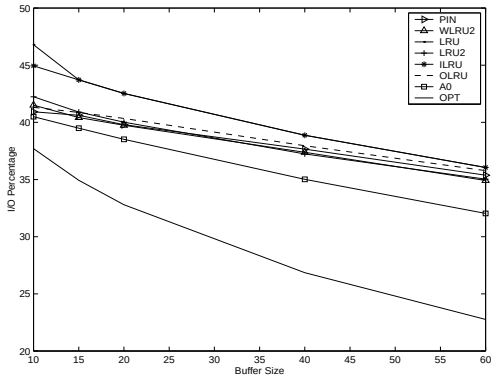

Figure 9. Percentage of I/O for uniform access pattern on synthetic data

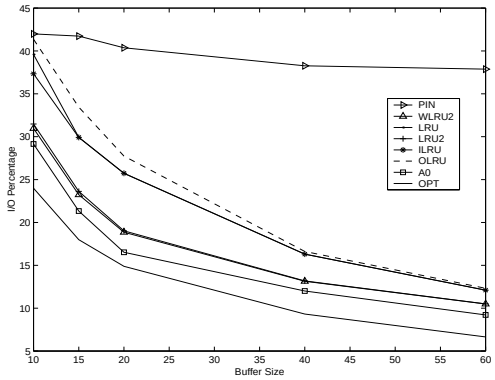

Figure 10. Percentage of I/O for non-uniform static access pattern on synthetic data

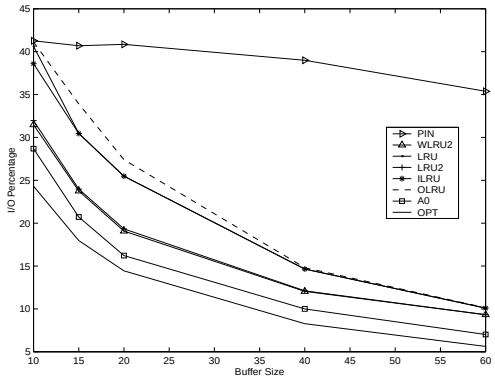

Figure 11. Percentage of I/O for non-uniform dynamic access pattern on synthetic data

5. DISCUSSION

In this paper we considered the buffering problem for uni-processor and multi-processor systems. We experimentally compared the existing buffering techniques for the uni-processor systems. We considered the Pinning, LRU, ILRU, OLRU, LRU-2, A_0 and the optimal algorithm in our experiments. We run our experiment on three different access patterns. These are uniform, non-uniform static and non-uniform dynamic access patterns. We experimentally showed that LRU-2 performs better than other techniques. According to our experimental results, the number of disk I/O's for the LRU-2 algorithm is $10 - 20\%$ less than that of other techniques for non-uniform access patterns. For uniform access pattern, the Pinning method performs as well as the LRU-2 method. The number of disk I/O's for the LRU-2 method (and also the Pinning method) is 5% less than that of LRU, ILRU, and OLRU methods.

In the second part of the paper, we showed that the LRU-K method can not be used directly for multi-processor systems. We proposed an approximation of the LRU-K method, called the WLRU-K method, which works well for multi-processor systems. We experimentally compared the WLRU-2 method with the LRU, ILRU and OLRU methods as well as the LRU-2 method. We experimentally showed that the WLRU-2 method performs as well as the LRU-2 method. Since the WLRU-K method does not store timestamps within the page, unlike the LRU-K method, the WLRU-K method does not need to write a page back the disk when it is evicted from the buffer. Therefore, the WLRU-K method can also be considered for the uni-processor systems.

Our future work on buffering technique will include examining other access patterns and video databases.

REFERENCES

1. A. V. Aho, P. J. Denning, and J. D. Ullman, "Principles of optimal page replacement," *J. ACM* **18**, pp. 80–93, 1971.
2. G. M. Sacco, "Index access with a finite buffer," in *VLDB*, pp. 301–309, (Brighton), 1987.
3. E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," in *SIGMOD*, pp. 297–306, 1993.
4. K.-Y. Whang, G. Wiederhold, and D. Sagalowicz, "Estimating block accesses in database organizations: A closer noniterative formula," *Commpunications of the ACM* , pp. 940–944, 1983.
5. S. T. Leutenegger and M. A. Lopez, "The effect of buffering on the perfomance of R-trees," in *Proceedings of the ICDE*, pp. 164–171, (Orlando, Florida), February 1998.
6. B. N., K. H.-P., S. R., and S. B, "The R*-tree: An efficient and robust access method for points and rectangles," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 322–331, (Atlantic City, NJ), 1990.
7. A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of the ACM SIGMOD Conference*, pp. 47–57, 1984.
8. V. Subrahmanian, *Principles of Multimedia Database Systems*, The Morgan Kaufmann Series, 1998.