Design and Analysis of Efficient Application-Specific On-line Page Replacement Techniques*

Virgil Andronache Edwin H.-M. Sha
Dept of Computer Science and Engineering
University of Notre Dame
Notre Dame, Indiana 46556

Nelson L. Passos
Department of Computer Science
Midwestern State University
Wichita Falls, Texas 76308

Abstract

The trend in computer performance over the last 20 years has indicated that the memory performance constitutes a bottlneck for the overall system performance. As a result, smaller, faster memories have been introduced to hide the speed differential between the CPU and memory. From the beginning of this process, an essential question has been the determination of which part of main memory should reside in the faster memory at each instant. Several on-line and off-line algorithms have been introduced, the best known and most used of which are LRU and MIN respectively. This paper introduces a new approach to page replacement in that it allows the compiler to enter the decision process. In introducing compiler help to page replacement, information available at compile time is passed on to new hardware added to the regular memory, with the overall effect of markedly decreasing overall memory access time.

1 Introduction

Beginning around 1980 and continuing through the present, an increasing performance gap has existed between memory and CPU performance. As a result of this trend, the techniques used to conceal this performance gap have become an issue of increasing importance in computer design. Nowadays, two level memory caches that are larger than the entire memory of primitive computers are commonplace and three level caches have begun being implemented in commercial personal computers. With growing miss penalties - in terms of CPU cycles - for each cache level, memory management algorithms that make better use of both spatial and temporal locality are one of the principal ways in which the memory and CPU can be made to work "in synch."

Recently, a number of new on-line algorithms have been proposed for varying purposes and with varying degrees of overall efficiency. One important concept that has surfaced is the LRU/k [11], together with its more efficient 2Q implementation [10]. LRU/k is an extension to the LRU page replacement concept, in which the last k accesses are taken into consideration in preempting a page from memory. Like other existing algorithms, LRU/k sets priorities according to page histories. A notable problem with history based priorities is the original access to a page, as well as first accesses after a period of inactivity. By using compile-time information, our algorithm assigns appropriate priorities to pages on all accesses, including the first one.

In [7], the algorithm centers on applications with repetitive access patterns. However, it waits for a problem - increased access cost - to occur before it attempts to resolve it. Additionally, the decision to change to replacement pattern has to be made at run time, thus incurring run-time overhead. By using the compiler to detect such patterns, our algorithm allows for the avoidance of such problems.

Other effective approaches have included variable-space memory management techniques, such as the working set algorithm [5, 6, 9, 12]. However, these techniques require a sizeable amount of extra hardware to perform a prefetch-like operation in which they pre-determine what pages will be needed by the executing process in the near future.

Currently the most widely used on-line algorithms for information replacement in the cache are least recently used (LRU) and random replacement. Studies have shown that LRU performs slightly better overall in terms of the cache hit rate [2, 8]. Our studies have shown that in the case of loop structures LRU performance can be improved upon. It is in this context that this paper proposes two new algorithms.

These algorithms differ in their complexity and their use of compiler involvement in the memory management process. The first (Basic) algorithm requires compiler support and a minimal amount of hardware, while the second (Ex-

*This work is partially supported by NSF grants MIP95-01006 and NSF ACS 96-12028, MIP-9704276 and JPL 961097

tended) makes increased use of information derived from the code being executed and requires additional hardware for implementation.

Consider the case of the Infinite Extent Impulse Response filter (IIR) below, originally described in [3].

$$H(z_1, z_2) = \frac{1}{(1 - \sum_{n_1=0}^{2} \sum_{n_2=0}^{2} c(n_1, n_2) * {z_1}^{-n_1} * {z_2}^{-n_2})}$$

This filter can be translated to:
$y(n_1, n_2) = x(n_1, n_2) + \sum_{k_1=0}^{2} \sum_{k_2=0}^{2} c(k_1, k-2) * y(n-1-k_1, n_2-k_2)$ for $k_1, k_2 \neq 0$.

In simulations run on this filter in a five level memory system, the new page replacements algorithm obtained improvements of up to 20% in memory access time over traditional LRU.

In order to describe the new techniques, section 2 introduces the background concepts and the basic ideas used in this paper. The new on-line algorithms are introduced in section 3, while section 4 contains the test results, together with the configurations used. A summary section concludes the paper.

2 Fundamental concepts

This paper deals with memory hierarchy and specifically with ways of transferring data between different memory levels in such a way that the memory latency is minimized. It considers the general case of a k-level memory hierarchy. In this case there are a total of k levels of memory in the system with the first level being the primary cache and the k^{th} level being the main memory.

As mentioned in the introduction, the most widely used page replacement algorithm in existing systems is LRU, which can be easily extended to multi-level memory systems.

A key observation to be made about nested loop implementations, is that when multi-dimensional structures are used, regular execution does not make the best possible use of the locality of reference present. To take better advantage of this locality of reference, this paper uses the technique of partitioning. The fundamentals of this technique can be seen in the following example. Consider the code

```
for i = 1 to 100
for j = 1 to 100
A[i,j] = B[i,j] * 5 + B[i-1,j]
```

If array A is modelled as a two-dimensional entity, the loop execution proceeds as seen in Figure 1.

Between the time when A[1,0] is computed and the time A[2,0] is about to be computed, the page containing B[1,0] - needed for both the former and the latter computations - is unlikely to be found in the primary cache. The phenomenon

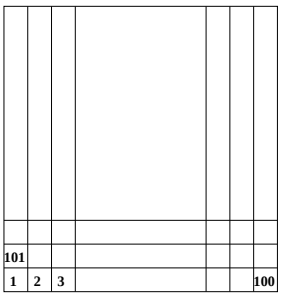

Figure 1. Regular execution sequence

cascades, such that less than optimal use of memory is made on practically every access to memory locations that have been used in a row prior to the one in current execution. The end result is a deterioration in the overall memory performance of the system. A counter measure to this phenomenon is to divide the iteration space as seen in Figure 2.

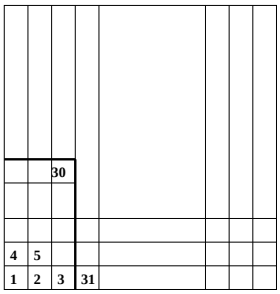

Figure 2. Partitioned execution sequence

In this case, the reduced number of iterations between the computation of A[1,0] and A[2,0] allows the page containing B[1,0] to remain in the primary cache long enough to avoid the extra penalty incurred in the previous case. To ensure that the pages needed stay close enough to the CPU, several modifications are made to the page replacement algorithm.

3 On-line Algorithms

In this section, two new on-line replacement algorithms are presented. Each algorithm will consist of two parts: a compiler part, in which decisions are made regarding the priorities associated with each access and a run-time part, in which the priorities are assigned to each accessed page. The first algorithm is very efficient while execution proceeds along a row. However, it incurs relatively significant penalties upon row switches. The second algorithm adds the observation that in nested loops there are two levels of

order. The first level, present along an individual row, states that elements used in an iteration are likely to be used again in the next. The second level of order concerns the transition from one row to the next. Since execution proceeds from left to right in each row, the pages that are going to be reused along the next row are going to be reused in roughly the same order that they were originally used. Thus, by the time the end of each row is reached, the pages that have been used in that row need to be in a last-in first-out order. It is this additional order that is implemented by the second algorithm.

The algorithms is to introduce compile-time information into the run-time priority assignment process. To illustrate compiler assisted priority, consider the IIR filter described in section 1. In terms of a programming language, this filter can be implemented with some index shifts as

```
for i = 1 to n
    for j = 1 to m

        y[i,j] = x[i,j] + c[2][2] * y[i-2][j-2] + c[2][1] *
y[i-2][j-1] + c[2][0] * y[i-2][j] + c[1][2] * y[i-1][j-2] +
  c[1][1] * y[i-1][j-1] + c[1][0] * y[i-1][j] + c[0][2] *
            y[i][j-2] + c[0][1] * y[i][j-1]
```

If we consider the region of matrix y that is accessed in iteration (2,2), we obtain the region in Figure 3.

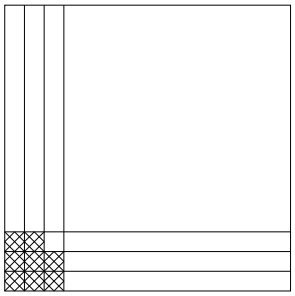

Figure 3. y array data referenced for square (2,2)

In Figure 3, when calculating the value that should be placed in the top right corner of each 3 x 3 square, a region corresponding to the shaded part of the square is accessed. For ease of understanding, we introduce the following definition.

Definition 3.1. *An **access region** is the totality of the memory locations accessed within one iteration.*

With this definition, the description of the first algorithm follows.

3.1 Basic On-line Algorithm

It can be seen that, in regular execution, advancing one iteration within the same row simply shifts the access region one column to the right. As a result, the memory locations accessed in the second column of the access region will be the first-column elements to be accessed in the next iteration. By the same reasoning applied in the optimal off-line algorithm, these elements should be assigned the highest priority. In the basic algorithms priorities are set once per iteration for each accessed page. Thus, since a page can be accessed multiple times in an iteration, a flag is needed to signal if the priority for a page has already been set in the current iteration.

Definition 3.2. *A **priority flag** is a one-bit field associated with each page, that indicates whether the priority has been set for that page in the current iteration. Priority flags are set on the first access to a page in each iteration and then cleared at the end of each iteration.*

In a similar fashion, priorities should decrease with each column moving off to the right from the second, while the priorities for elements accessed in the first column of the execution region should be set to zero. The implementation of the system needed for this assignment can be additionally simplified by allowing the compiler to relocate instructions in such a way that accesses to pages that will be used in the following iteration starting with the data in the second column of the access space are made closer to the beginning of each iteration.

The next step in the compiler involvement actually requires priority assignments to each page accessed in an iteration. In the case of nested loops with regular dependencies, analyzing the execution in each row reveals the simple access structure shown above. In this context, the priority of a location in execution space would not change while changing from an iteration to the next within the same row. As a result, a relation can be set up between the index of each access in an access space and the priority related to it. The algorithm is described below, in its two component parts: the compiler aid and the hardware execution.

The compiler section of the algorithm is based on grouping certain dependency vectors together based on certain properties of their components. For a better understanding of these properties, the following definitions are introduced.

Definition 3.3. *An **iso-y subset** is a subset of the dependency vector set that contains dependency vectors of equal y components and whose x components form an ordered, uninterrupted interval on the integers.*

Definition 3.4. *A **maximal iso-y subset** is an iso-y subset such that the addition of any dependency vector would result in the set not being an iso-y subset.*

Both the basic and extended algorithms assign priorities to each page once every iteration. Once the priority is set a boolean variable associated with each page is set to TRUE and checked on subsequent accesses. This variable is reset to FALSE at the beginning of each iteration. Each priority is directly related to the dependency vectors that result in accesses to the page. In particular, the priority depends on the maximal iso-y subsets that the dependency vectors are part of and their positions in that subset. As a result, a strict order needs to be established among dependency vectors. In our case, the ordering involves arranging the dependency vectors in a y-descending, x-descending sequence.

Definition 3.5. *A set of dependency vectors,* $\{d\} = (d_x, d_y)$ *is in y-descending, x-descending order if, given i and j integers such that* $i < j$*, then* d_i *and* d_j *satisfy one of these two conditions:*

1. $d_{i_y} < d_{j_y}$*, or*
2. $d_{i_y} = d_{j_y}$ *and* $d_{i_x} < d_{j_x}$

This sequencing method naturally creates the maximal iso-y subsets of the dependency vector set. For ease of priority assignment, the process below is used, starting with the following definition.

Definition 3.6. *The* ***index*** *of a dependency vector is the position that dependency vector within the reordered dependency vector set.*

Algorithm SPRT (sprite) Compiler

Input: A set of dependency vectors D = $\{d_1, d_2, ..., d_n\}$ in y-descending, x-descending order.

Output: A set of priorities for the memory accesses related to these dependencies.

1. Find all maximal iso-y subsets, $\epsilon_1, \epsilon_2, ..., \epsilon_m$
2. For each $\epsilon = \{\delta_1, \delta_2, ..., \delta_k\}$
   Priority(δ_1) $\leftarrow$ 1
   for j = 2 to k
   Priority(δ_j) $\leftarrow$ $\mid D \mid$ - Index(δ_j) + 2
   Priority(d_1) = 0
   end

This algorithm is run separately for each maximal iso-y subset, with each data array having separate iso-y subsets.

One thing to be noticed is the added constant 2 to the priority associated with most accesses. The constant is needed for the last maximal iso-y subset. Its absence would result in the first and last elements in that subset having the same priority and, as a result, a page that would be needed again might be moved down in the memory hierarchy.

It was also noted in our experiments that one place where LRU was consistently being outperformed by the optimal algorithm was the last access to a page. In the basic algorithm the solution adopted for this problem was to reverse the first two accesses in each maximal iso-y subset with two or more elements. This way, if the first two dependency vectors result in accesses to the same page, the page will be accessed in the next iteration and the priority is set high due to the second dependency vector. If the two accesses are made to different pages each page gets the priority set by the corresponding vector.

Algorithm SPRT Run-time

Input: A page together with the priority associated with the current access at compile time

Output: The priority for the page

if the current_page.set_priority = 0
set current_page.priority $\leftarrow$ compile time priority for current access
current_page.set_priority $\leftarrow$ 1
Return current_page.priority
end

With these considerations, the continuous section of execution that takes place in each row is serviced at a rate very close to optimal in terms of memory cost. For further improvements, the second algorithm needs to be explored.

3.2 Extended On-line Algorithm

This algorithm goes into more detail in assigning priorities to pages, to the overall effect of minimizing the penalty incurred at the end of each row fragment in execution space. An additional requirement for better performance comes from the observation that while locality is important, in a number of practical cases the information is only accessed once in each location. In this case, it is of a substantial advantage to be able to determine when the last location in a page is accessed. The advantages derived from this are evident in the second of the filters that the algorithms have been tested on. Thus, in the instances where no further accesses are expected to a page, a very low priority can be set. Finally, with the use of partitions, one more problem is generated: priorities being set every iteration, the priorities set in the last iteration in each row remain set unless explicitly changed by a latter access. The main implication is that pages with high priorities remain to reduce the effective usable space of low memory levels. Thus, an additional bit is needed to mark those pages used in the last iteration. Since those pages will most likely not be used until the end of the next row, at the end of the last iteration the priority of each page is set down to 1.

A side advantage of this approach to page replacement is obtained from the fact that the algorithm replaces pages in a non-preemptive manner. The end result of the process is that at the end of each row, the pages that have been used in that row will be found in memory in roughly Last In First Out order. Thus, the inherent working of the algorithms reduces the additional cost incurred over the optimal.

It should be noted that the compile-time process, (named XPRT compiler) is identical for the two algorithms. The difference in efficiency is gained at the hardware level, with additional checks for the last memory location in a page and the last iteration in each row of a partition.

Algorithm XPRT Run-time

Input: A page together with the priority associated with the current access at compile time and the iteration in the current row

Output: The priority for the page

if the current_page.set_priority = 0

if the access is the first in a maximal iso-y subset and it accesses the last location in a page, current_page.priority ← -127

if the current iteration is the last iteration in the row, current_page.last_iteration ← 1

set current_page.priority ← compile time for current access

current_page.set_priority ← 1

if the end of the last iteration in the current row has been reached

for all pages in memory

if page.last_iteration = 1

page.priority ← 0

Return current_page.priority

end

In conclusion, the mechanism for setting a priority is as follows. If a priority has not been set, set the priority according to the guidelines below.

1. If information from the page being accessed is not required in a different row than the current one and the access is not to the last location in the page, set the priority to 0.

2. If in the above situation the access is to the last location in the page, set the priority to the lowest allowable by the hardware (-127 in the case of an 8-bit priority field).

3. Otherwise, set the priority according to the number of the access in the iteration. This value is decided at compile time and is programmed into the memory controller.

It should be noted that the priority bits are reset at the end of each iteration in both the basic and extended algorithms.

4 Experimental Results

In this section the simulation results of several application related code sequences run under various memory configurations are presented. The results are given in the form of total memory access time under the following criteria:

1. The system consists of five memory sizes, each level having double the capacity of the previous one, with the exception of the last level, which is considered to have infinite capacity.

2. The cost of fetching a page is taken as one less than the number of the memory level that the page was found in, where the first level is the primary cache and the fifth level is the main memory.

3. The data is assumed to be stored in a row-wise manner.

With these assumptions, the results obtained in running the filter previously presented in this paper are seen in Table 1. [1]

LRU	Basic	Extended	Memory size	Page size
22589	17469	16643	4,8,16,32,∞	12
28528	23920	21437	4,8,16,32,∞	8
12706	12706	11880	8,16,32,64,∞	12
16877	16877	16205	8,16,32,64,∞	8

Table 1. Memory access costs for IIR filter

Another filter, presented in [13], was also tested. This filter exhibited a smaller amount of locality and that is evident in the results from Table 2.

LRU	Basic	Extended	Memory size	Page size
36647	33605	30013	4,8,16,32,∞	12
41663	38686	35533	4,8,16,32,∞	8
25246	25246	24593	8,16,32,64,∞	12
32436	32436	31913	8,16,32,64,∞	8

Table 2. Memory access costs for second filter

A third test was run on a filter designed according to the Fettweis method [4] to solve a circuit transmission problem. The results were as follows.

Finally, a fourth test was run on livermore loop 18 with the following results.

[1] The memory sizes are given in terms of the number of pages that can be stored in each memory level

LRU	Basic	Extended	Memory size	Page size
76799	76799	76799	4,8,16,32,∞	12
82943	82943	78336	4,8,16,32,∞	8
76799	76799	66476	8,16,32,64,∞	12
82943	82943	73728	8,16,32,64,∞	8

Table 3. Memory access costs for Fettweis filter

LRU	Basic	Extended	Memory size	Page size
147025	134225	131865	4,8,16,32,∞	12
182368	166240	147040	4,8,16,32,∞	8
109747	96307	93411	8,16,32,64,∞	12
143681	124817	115517	8,16,32,64∞	8

Table 4. Memory access costs for livermore loop 18

It can be seen from above that improvements on the order of 20% can be obtained on a regular basis with the new online algorithms.

5 Summary

The desire to keep the CPU operating as much of the time as possible leads us to continue to strive to find more efficient ways to utilize the memory in high speed caches. This paper has demonstrated that with minimal hardware alteration and compiler support, it is possible to greatly increase the effectiveness of page replacement algorithms. It has presented two algorithms which far exceed the performance of both the online standards for memory management.

The online algorithms, SPRT and XPRT were introduced and described. Both algorithms measurably outperform LRU, the commonly accepted most efficient algorithm for online page replacement. The key concept behind the online algorithms resides in the fact that by utilizing the locality of reference inherent in nested loops and information from the compiler, it is possible for an online algorithm to gain an approximation of the foresight that an off-line algorithm would have access to, and to approach the performance of an off-line optimal algorithm. These algorithms outperform LRU by up to 20% while requiring only a modest amount of modification to the current hardware model.

References

[1] L. A. Belady. A study of replacement algorithms for vitual storage computers. *IBM Systems Journal*, 5:78–101, 1966.

[2] M. Chrobak and J. Noga. LRU is better than FIFO. *ALGORTHMICA: Algorithmica*, 23, 1999.

[3] D.E. Dudgeon and R.M. Mersereau. *Multidimensional Digital Signal Processing*. Prentice Hall, Englewood Hills, NJ, 1984.

[4] A. Fettweis and G. Nitsche. Numerical integration of partial differential equations using principles of multidimensional wave digital filters. *Journal of VLSI Signal Processing*, 3:7–24, 1991.

[5] A. Fiat and A. R. Karlin. Randomized and multipointer paging with locality of reference. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 626–634, 1995.

[6] A. Fiat and M. Mendel. Truly online paging with locality of reference. In *Proceedings of the 38th IEEE Symposium on Founations of Computer Science*, pages 326–335, 1997.

[7] G. Glass and P. Cao. Adaptive page replacement based on memory reference behavior. *Performance Evaluation Review: A Quarterly Publication of the Special Interest Committee on Measurement and Evaluation*, 25(1):115–126, June 1997.

[8] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan and Kaufmann Publisher Inc., California, 2 edition, 1996.

[9] S. Irani, A. Karlin, and S. Phillips. Strongly competitive algorithms for paging with locality of reference. In *3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 228–236, 1992.

[10] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th VLDB Conference*, pages 439–450, 1994.

[11] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Special Interest Group on the Management of Data Record*, volume 22, pages 297–306, Washington DC, May 1993.

[12] K. Park, S. L. Min, and Y. Cho. The working set algorithm has competitive ratio less than two. In *Information Processiong Letters*, volume 63, pages 183–188. Elsevier Science, August 1997.

[13] N. L. Passos. *Improving Parallelism on Multi-Dimensional Applications: The Multi-Dimensional Retiming Framework.* PhD thesis, University of Notre Dame, June 1996.