Approximating the Optimal Replacement Algorithm

Ben Juurlink
Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
P.O. Box 5031, 2600 GA Delft
Netherlands
B.H.H.Juurlink@ewi.tudelft.nl

ABSTRACT

LRU is the de facto standard page replacement strategy. It is well-known, however, that there are many situations where LRU behaves far from optimal. We present a replacement policy that approximates the optimal algorithm OPT more closely by *predicting* the time each page will be referenced again and by evicting the page that has the largest predicted time of next reference. Experiments using several benchmarks from the SPEC 2000 benchmark suite show that our algorithm is superior to LRU, in some cases by as much as 25%-30% and in one case by more than 100%.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Virtual memory*

General Terms

Algorithms, performance

Keywords

Virtual memory, paging, replacement strategy, LRU, OPT.

1. INTRODUCTION

Usually, the address space of a program is much larger than the amount of physical memory. To solve this problem, a technique called virtual memory is employed. It divides the virtual address space into fixed size pages and uses the main memory as a cache for the secondary storage. A page fault or miss occurs when a reference is made to a page that is not in main memory. In that case the page replacement strategy determines which page is removed from the main memory so that the missing page can be brought in.

It is well-known that it takes millions of cycles to serve a page fault. Because of this, even a small reduction in the miss rate will pay for the cost of an advanced replacement algorithm [6]. The least recently used (LRU) strategy or some variant of it is the most commonly used scheme. It replaces the page that has not been used for the longest time. The optimal algorithm (OPT) evicts the page that will not be used for the longest time. This is an off-line algorithm, since it requires knowledge of future page references. In a sense, the LRU strategy approximates the optimal algorithm, because the *expected* time until the next access to a page is equal to the amount of time that has elapsed since the most recent access to that page.

It is known, however, that there are many request sequences for which LRU behaves far from optimal. The classic example is that of a circular sequence of page references that accesses one more page than the number of physical page frames. Assume, for example, that there are four page frames and that the reference string is

$$1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, \ldots$$

Then LRU yields a miss rate of 100% whereas OPT generates a page fault in only 2 out of 5 cases, corresponding to a miss rate of 40%.

It can be observed, however, that the request sequence above is very predictable since the number of pages accessed between two successive references to a page p is always 4. In this paper we present a page replacement algorithm called TNRP (Time of Next Reference Predictor) that exploits this information to predict the time each page will be referenced again and selects the victim page by considering its predicted time of next reference. In other words, it is assumed that each page is accessed at a regular interval and that the time that a page will be re-referenced can be predicted by adding its interval to its time of last reference.

The organization of this paper is as follows. In Section 2 related work is described. Section 3 presents our novel page replacement algorithm. Implementation issues are discussed in Section 4 and experimental results obtained using several SPEC 2000 benchmark are presented in Section 5. Finally, in Section 6, conclusions are drawn.

2. RELATED WORK

The paging problem has received considerable attention in the theoretical community. Sleator and Tarjan [13] proposed to use competitive analysis to determine how well an algorithm is performing. The competitive ratio of an algorithm is c if the algorithm incurs at most c times as many page faults as the optimal, off-line algorithm on every input.

CF'04, April 14–16, 2004, Ischia, Italy.

Sleator and Tarjan showed that no deterministic, online (i.e., without knowledge of future accesses) algorithm can achieve a competitive ratio better than n, where n is the number of physical page frames. They also showed that FIFO and LRU are n-competitive, and hence best possible in their model.

In practice, however, LRU usually outperforms FIFO. Furthermore, its competitive ratio is usually much better than n. The reason is that programs exhibit locality of reference. Borodin et al. [2], therefore, proposed to use an *access graph* to limit the set of request sequences that can be made. The access graph of a program has a vertex for every page the program can reference and there is an edge between two vertices if the two pages associated with the endpoints can be referenced successively. Thus a request sequence must correspond to a walk on the access graph. Borodin et al. presented an algorithm called FAR and proved that it incurs at most $O(\log n)$ times as many page faults as any online algorithm. This bound was later improved by Iarani et al. [7], who showed that FAR is in fact optimal to within a constant factor. However, FAR is not a truly online algorithm since the access graph must be known a priori.

A truly online algorithm has been presented by Fiat and Rosen [4]. Their heuristic dynamically builds a (directed or undirected) *weighted access graph* $G = (V, E, w)$ that has a vertex for every page the program references. An edge is created between two vertices the first time two pages are requested in succession, and each time the edge is traversed (i.e., the two pages associated with the endpoints are referenced successively), the edge weight is decreased by a constant factor. "Forgetfulness" is introduced by multiplying the weight of all edges by a constant factor every $10n$ page accesses, where n is the number of page frames. The page to be evicted on a page fault is the furthest page from the page accessed just before the page fault in the access graph. A detailed comparison between our algorithm and the algorithm of Fiat and Rosen is beyond the scope of this paper, but we would like to remark that the algorithm of Fiat and Rosen seems rather costly. Although the most time-consuming work (finding the page furthest from the current page in the access graph) is performed on a miss, on a hit the edge associated with the two pages accessed in succession has to be located (or created) to decrease its weight. Since this must be performed on every memory access, it appears that this cannot be done without significantly affecting the time to hit the translation lookaside buffer (TLB).

O'Neil, O'Neil, and Weikum [12] proposed a replacement algorithm called LRU-2. This strategy replaces the page whose penultimate (second to last) access is least recent. LRU-2 improves upon LRU because the second to last access is a better indication of the interarrival time between references than the most recent access. It can be seen, however, that LRU-2 does not remove the problems with circular reference strings. On such sequences, it behaves as the conventional LRU algorithm. Although a detailed comparison between LRU-2 and TNRP is beyond the scope of this paper, we will present some results comparing LRU-2 and TNRP.

Johnson and Shasha [9] observed that LRU-2 has two practical limitations: (1) it needs to maintain a priority queue which implies that each time an uncorrelated reference occurs $O(\log n)$ work needs to be performed, and (2) it has several parameters that need to be tuned. They, therefore, propose an algorithm called 2Q which is similar to LRU-2 but has constant time overhead per page access. Although their algorithm also has two parameters (*Kin* and *Kout*) and they remark that fixing these parameters is a tuning question, they show that fixed, a priori values work well in practice. We remark that our algorithm TNRP also has two tunable parameters but we also show that using fixed, a priori values works well across all workloads and cache sizes. Furthermore, although our algorithm in principle also requires a priority queue to quickly find the victim page, we argue that the time to read the disk is so large that we can spent some time to determine the victim page.

A truly self-tuning replacement strategy called ARC has recently been proposed by Megiddo and Modha [11]. It maintains two variable-sized lists, L_1 and L_2, that together contain twice as many pages as the number of page frames n. L_1 can be though of as containing recent pages while L_2 can be thought of as containing high-frequency pages. ARC keeps the p most recent pages in L_1 and the $n - p$ most recent pages in L_2 in main memory, where p is a parameter that is dynamically adapted in an online, self-tuning fashion. In this way, ARC dynamically balances between recent and high-frequency pages.

Our algorithm predicts the time a page will be referenced again based on its stride, i.e., the number of pages accessed between its last and second to last access where consecutive accesses to the same page are treated as one access. Stride prediction has been used before, most notably in the area of hardware prefetching (see, e.g., [1, 5, 8]). However, we use the stride to predict *when* a block or page will be referenced again and to determine which page is replaced on a page fault.

Wong and Baer [15] proposed two modified LRU policies. Although they focussed on improving L2 cache behavior, their strategies can also be used for the paging problem. They associate a temporal bit with each cache line that indicates that the data exhibits temporal locality. Instead of always replacing the LRU line, the victim is chosen by considering both its time of last access and whether the temporal bit is set or not.

3. THE ALGORITHM

In this section our novel replacement strategy is described. The algorithm will be referred to as TNRP (*Time of Next Reference Predictor*).

The algorithm works as follows. For each page p, we record its time of last reference $tlr(p)$ as well as the number of pages accessed between its last reference and its previous to last reference. The latter is called the stride of page p and will be denoted by $str(p)$. This information can be stored in the page table and cached in the translation lookaside buffer. The first time a page is referenced $str(p)$ is set to zero. In addition, a state is associated with each page. The state is set to *Transient* the first time a page is referenced. When "more or less" the same stride occurs two times in a row, the page enters the *Steady* state. We say "more or less" because page references are not as predictable as, for example, data references or branches. In particular, the page enters or remains in the *Steady* state when the current stride $str_{curr}(p)$ is such that

$$str(p) - SD \leq str_{curr}(p) \leq str(p) + SD,$$

where SD is a constant value called the *Stride Deviation*.

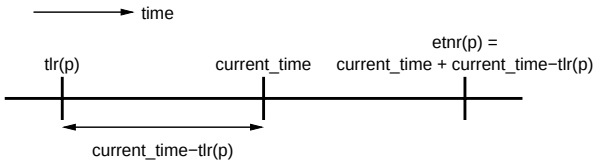


Figure 1: The expected time of next reference of a page based the time of last access.

When a page fault occurs, the page p whose *expected time of next reference* or $etnr(p)$ is the largest is selected for replacement. $etnr(p)$ is defined as follows. If the page has been accessed before and its state is *Steady*, then

$$etnr(p) = tlr(p) + str(p). \quad (1)$$

If the page has not been accessed before or the state is transient, the expected time of next reference is based on the time of last reference, as in LRU. As illustrated in Figure 1, based only on the time of last reference the expected time of next reference is

$$current\_time + current\_time - tlr(p), \quad (2)$$

where *current_time* denotes the current time. However, we give precedence to pages in the *Steady* state, because otherwise, such pages would be replaced in most cases. Consider, for example, the following reference string

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 5, 11, 12, 13, 14, 5, 16, 17, 18, 19, 5, \ldots$$

and assume that three pages can be kept in memory. Although page 5 is the only page that is reused and accessed with a regular stride of 5, it will never hit the cache if the expected time of next reference of the other pages is given by Equation (2). Consider, for example, "cycle" 16 in which page 16 is accessed. At the beginning of this cycle pages 13, 14, and 5 are in memory. Furthermore, $etnr(5) = 20$, and if the expected time of next reference of pages 13 and 14 is given by Equation (2), $etnr(14) = 18$ and $etnr(13) = 19$. So, page 5 will be replaced.

Therefore, if the page has not been accessed before or the state is transient, the expected time of next reference is given by

$$etnr(p) = current\_time + TF \cdot (current\_time - tlr(p)), \quad (3)$$

where TF is a constant factor larger than or equal to 1 called the *time of next reference factor*.

A further refinement is necessary to prevent pages that are dead (i.e., they will never be referenced again) from staying in memory. If a page is not referenced within the time it is expected to be referenced, i.e., when

$$current\_time > tlr(p) + str(p) + SD$$

the expected time of next reference is also given by Equation (3) rather than Equation (1). The next time page p is accessed it will return to the transient state.

4. IMPLEMENTATION

There are two implementation details that need to be discussed. First, since TNRP bases its predictions on the reference string where consecutive accesses to the same page are collapsed into one access, it needs to be discussed how this sequence is generated. Second, we need to discuss how information about pages currently not cached in physical memory is maintained.

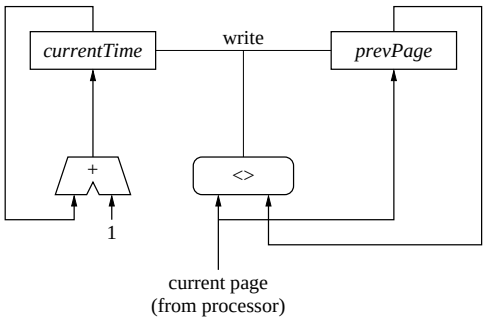


Figure 2: Hardware to compute the current time.

As remarked in Section 3, for each page p we record its time of last reference *tlr(p)* and its stride *str(p)* in the page table that contains the physical page addresses. Furthermore, these fields are cached in the TLB. The "time" is defined as the number of pages accessed since the process was started, where consecutive accesses to the same page are replaced by one access. Suppose, for example, that the following pages are accessed in succession:

```
1   4   1   1   6   1   1   4   4   6   6
```

This sequence reduces to the following reference string with time and stride as indicated

```
Time:   0   1   2   3   4   5   6
Page:   1   4   1   6   1   4   6
Stride: ?   ?   2   ?   2   4   3
```

Figure 2 depicts some simple hardware to compute the current time. The page number of the page accessed previously is stored in the latch *prevPage*. If the current page number does not match *prevPage*, the *write* signal is asserted so that the incremented time and the current page number are latched into *currentTime* and *prevPage*, respectively.

When the current page number p does not match the number of the previous page, its time of last reference *tlr(p)* and its stride *str(p)* need to be updated in the TLB. Updating *tlr(p)* simply means replacing it by *currentTime* and *str(p)* is computed by subtracting the old *tlr(p)* from *currentTime*. Because the computations are very simple, we assume these actions do not increase the time to hit the TLB.

The last question that needs to be addressed is how a victim page is found on a page fault. If it is requested to find the victim page quickly, one possibility is to maintain a priority queue of the paged in pages, where the priority is given by the predicted time of next reference. LRU-2 also needs a priority queue and this was mentioned as one of the limitations of LRU-2 in [11] and [9]. Using a priority queue, it takes $O(\log n)$ time to find the victim page, but this is not critical since it takes millions of cycles to serve a page fault. However, it also takes $O(\log n)$ time to update the priority queue in case the expected time of next reference to a page changes and this is critical since such updates occur frequently. We, therefore, do not maintain a priority queue but simply scan all the paged in pages when a page fault

Benchmark	Category	Number of data pages
164.gzip	Compression	861
175.vpr	FPGA placement and routing	50
176.gcc	C compiler	3435
177.mesa	3D Graphics library	2274
181.mcf	Combinatorial optimization	23744
183.equake	Seismic wave simulation	2482
255.vortex	Object-Oriented database	3551
256.bzip2	Compression	4399
300.twolf	Place and route simulator	46

Table 1: Benchmark descriptions

occurs to find the victim page. Since bringing in a page from disk takes millions of cycles, we estimate that this can be done without affecting performance significantly.

5. EXPERIMENTS

5.1 Methodology

We used trace-driven simulations to test the effectiveness of our replacement strategy. The traces were generated with the SimpleScalar toolset [3] using a simple in-order processor model. In all experiments, the page size was set to 4096, which is the page size used in the SimpleScalar simulators.

We only consider references to data pages. Although we performed some experiments on program traces, the working sets of most benchmarks were so small that the difference between the TNRP algorithm and LRU was hardly noticeable.

The set of benchmarks we employed is a subset of the SPEC 2000 benchmark suite. The benchmarks are briefly described in Table 1. There are three reasons why we have not simulated all benchmarks from the SPEC 2000 suite. First, the SimpleScalar toolset [3] only includes a C compiler so the benchmarks written in Fortran could not be used. Second, some of the benchmarks written in C did not compile correctly on our system or produced wrong results. The reason for this needs to be determined. Third, as explained in the following paragraph, we used reduced input datasets and such reduced inputs were not provided for all benchmarks.

In order to reduce the simulation time, we used the large reduced input datasets described in [10]. The execution characteristics of these reduced input datasets are similar to the execution profiles of the SPEC 2000 reference datasets. As argued in [10], using these reduced input datasets should be more representative than reducing the runtime by simulating, say, 100 million references from the middle of program execution, since this may cause the execution profile to be completely different from the execution profile obtained with the full input dataset. The last column in Table 1 shows the number of data pages referenced by each application using these reduced inputs.

5.2 Experimental Results

5.2.1 TNRP Parameters

Recall from Section 3 that our TNRP algorithm has two parameters: the *Stride Deviation* (*SD*) and the *Time of next reference Factor* (*TF*). The goal of this section is to determine appropriate values for them.

The *SD* parameter determines when a page enters or remains in the steady state and also controls if the expected time of next reference is given by Equation (3) or Equation (1). To determine an appropriate value for this parameter, we investigated how the stride varies over time using a training set of benchmarks (the first four benchmarks namely *164.gzip*, *175.vpr*, *176.gcc*, and *177.mesa*). Figure 3 depicts a histogram that shows how many times a stride difference (i.e., $|str_{\text{curr}} - str_{\text{prev}}|$, where str_{curr} is the current stride and str_{prev} the previous stride) of k has occurred, for $0 \leq k < 10$. The results are normalized w.r.t. the total number of time a stride has occurred. The last bar in each figure shows the number of times the stride difference was larger than or equal to 10. Note that these figures also give an impression of how predictable the time of next reference is.

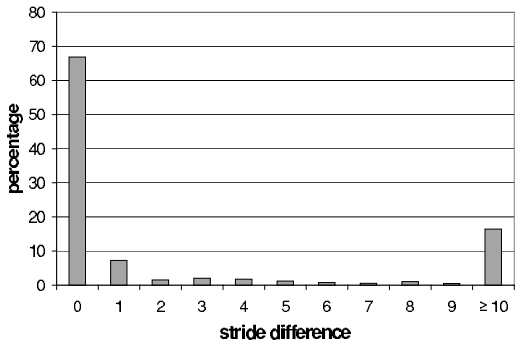
The precise behavior depends on the benchmark. For *164.gzip* and *176.gcc*, a stride difference of 0 occurs more often than any other stride difference, indicating that the stride is a good predictor of the time a page will be reused. A stride difference between 1 and 9 occurs seldom and in approximately 16% of the cases a stride difference larger than 9 was observed. This is not the case for *175.vpr*. For this benchmark a stride difference larger than 9 occurred more often than a stride difference of 0. Furthermore, a stride difference between 1 and 5 also occurred in a significant number of cases. The behavior of *177.mesa* is somewhere in between. For this benchmark stride differences of 0 and 1 occurred most often and in about 12% of the cases the stride difference was larger than 9.

We fixed the *SD* value at 5. Using this value captures more than 60% of all stride values for the *175.vpr* benchmark and more than 75% of all stride values that occurred during execution of the other three benchmarks from this training set.

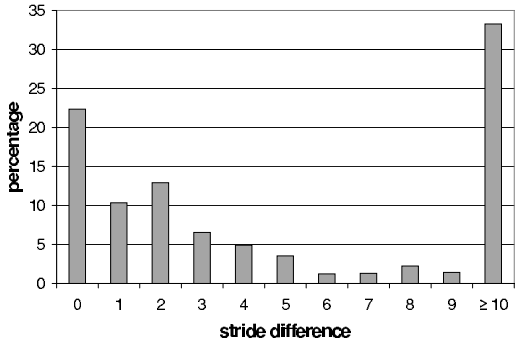
The *TF* factor controls the amount of priority given to pages in the steady state. To determine an appropriate value for this parameter, we measured the performance of TNRP with different *TF* values between 1 and 3. The *SD* value was fixed at 5. Figure 4 depicts the performance as a function of the *TF* value. Figure 4(a) depicts the performance when there are four physical page frames, and Figure 4(b) depicts the performance when there are eight page frames. The miss rate is normalized to the number of misses incurred when $TF = 1$.

It can be seen that the performance is rather independent of the exact *TF* value. When the number of page frames is four, the difference is at most 5%, and when there are eight page frames, the differences are even smaller (at most 0.8%). Nevertheless, some variance can be observed. For some benchmarks the miss rate monotonically decreases as the value of the *TF* parameter is increased. For other benchmarks, the miss rate reaches a minimum and then increases again. Generally, the best results are obtained using a *TF* value between 2 and 2.75. We used the minimum value in this range so that the algorithm is not too optimistic but returns to the default behavior (LRU replacement) when the time of next reference is not very predictable.

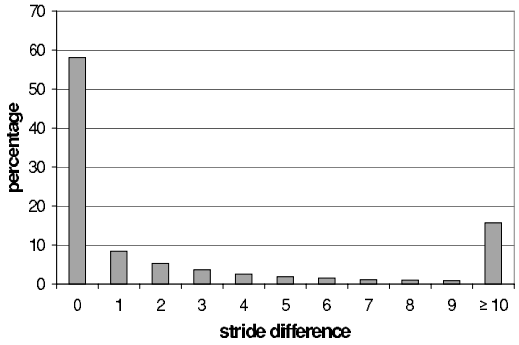
We remark that although algorithm TNRP uses two parameters, its performance is relatively independent of the exact values. In other words, we have found that using fixed,



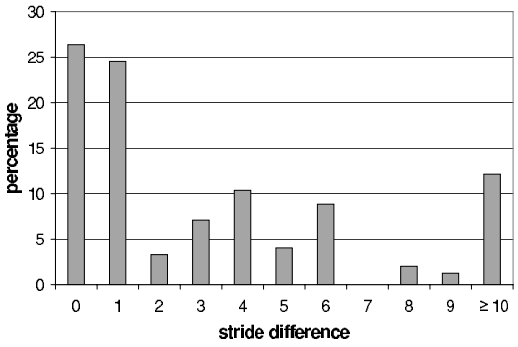
(a) 164.gzip



(b) 175.vpr

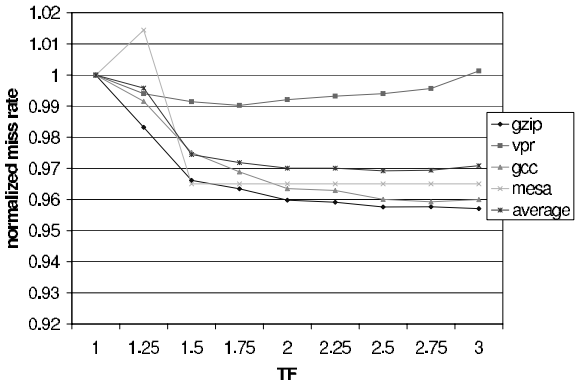


(c) 176.gcc

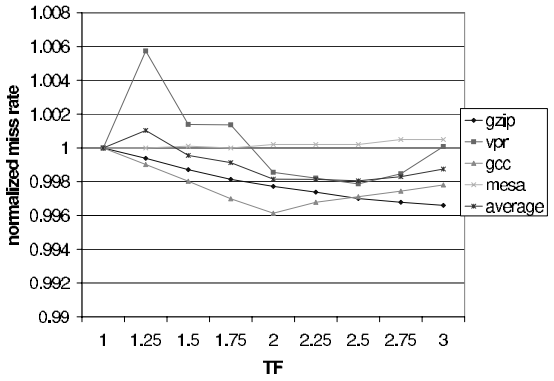


(d) 177.mesa

Figure 3: Distribution of stride difference



(a) Four page frames



(b) Eight page frames

Figure 4: Miss rate as a function of the *TF* value. The miss rate is normalized to the number of misses incurred when $TF = 1$.

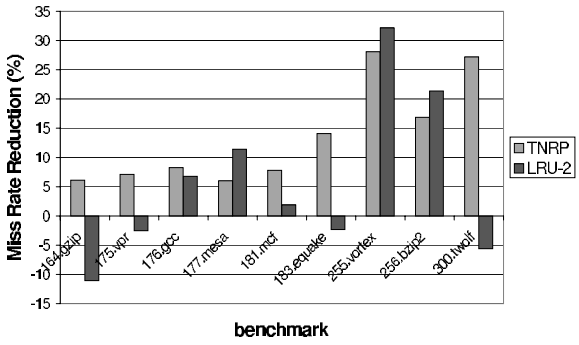
a priori choices works well for all workloads and number of page frames.

5.2.2 Miss Rate Reduction

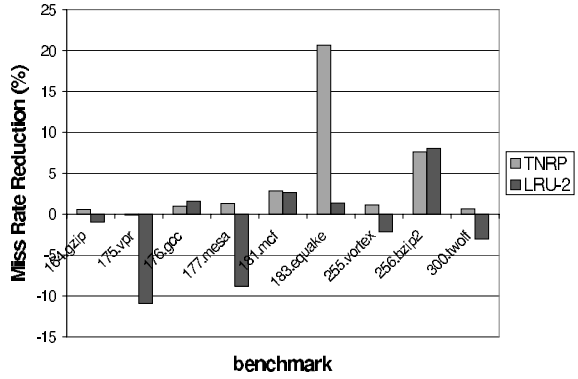
In this section the results of the TNRP algorithm are presented. The performance of TNRP is compared to the performance achieved by LRU as well as LRU-2.

The performance improvement of our page replacement algorithm over LRU is depicted in Figure 5. This figure shows the miss rate reduction, i.e., the number of misses TNRP incurs fewer than LRU as a percentage of the number of misses of TNRP. Figure 5(a) depicts the results for four page frames, Figure 5(b) for eight frames, and Figure 5(c) for 16 frames. As explained in the previous section, the TNRP parameters *SD* and *TF* are fixed at 5 and 2.0, respectively. The LRU-2 algorithm also has a parameter called *Correlated Reference Period* (*CRP*). This parameter captures the amount of time a page that has been referenced only once recently should be kept in cache and is related to the *TF* parameter. We used the value $0.25 \cdot n$ for the *CRP* parameter, where n is the number of page frames.

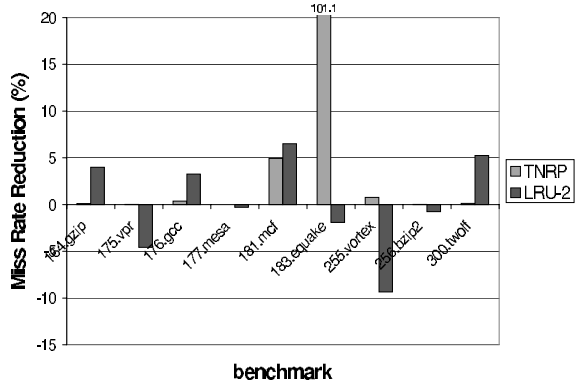
Figure 5 shows that in all but one case TNRP incurs fewer misses than LRU. The only exception is *175.vpr* with eight page frames, but in this case the additional number of misses



(a) Four frames



(b) Eight frames



(c) 16 frames

Figure 5: Miss rate reduction achieved by TNRP and LRU-2.

generated is less than 0.12%. A partial explanation for this can be given by looking at Figure 3(b). For this benchmark a stride difference larger than 9 occurred more often than any other stride difference, indicating that the stride may not be a good predictor of the time of next reference. Note that when the stride varies heavily, TNRP behaves almost as LRU. For all other benchmarks and number of page frames, however, TNRP performs better than LRU.

It can be seen that when the number of physical page frames is four, the improvement is most noticeable. In that case the improvement ranges from 6.0% for *177.mesa* to 28.1% for *255.vortex*, with an average of 13.5%. When the number of page frames is eight, the average improvement is smaller (4.0%), but for two benchmarks TNRP reduces the number of misses significantly (20.7% on *183.equake* and 7.6% on *256.bzip2*). When there are 16 page frames, the improvement is less than 1% for 7 of the 9 benchmarks. For *183.equake*, however, TNRP generates half as many misses as LRU. In absolute numbers, LRU incurred 3.23 million misses on this benchmark whereas TNRP generated only 1.61 million misses. The reasons for this need to be investigated further.

LRU-2 outperforms LRU for some benchmarks and number of page frames, but there are also many cases when LRU-2 actually performs worse than LRU. For example, when the number of page frames is four, LRU-2 performs slightly better than TNRP for three out of nine benchmarks, but for four benchmarks it performs worse than LRU. Also when the number of page frames is eight or 16, there are several benchmarks for which LRU-2 performs worse than LRU, whereas TNRP is beaten by LRU only in one case. LRU-2 was developed for online transaction processing systems, and it appears to perform less well for general-purpose programs.

Concluding, we find that the TNRP algorithm can noticeably reduce the number of misses and that the performance improvements are most significant when the number of physical page frames is small. This suggests that our algorithm works best in a multi-tasking environment, where the physical memory available needs to be shared among several processes.

5.2.3 Comparison With OPT

In order to investigate if the TNRP algorithm can be improved, we compare its performance to that of the optimal replacement strategy OPT. Our implementation of OPT is based on the algorithm described in [14].

Figure 6 depicts the competitive ratio of the TNRP algorithm. Recall from Section 2 that the competitive ratio is c if the algorithm incurs at most c times as many page faults as the optimal algorithm OPT. It can be seen that the competitive ratio achieved by TNRP is less than 2.05 in all cases. The average competitive ratio is 1.51. Furthermore, the competitive ratio tends to increase with the number of page frames. This indicates that there is scope for improvement, at least when comparing TNRP to the optimal off-line algorithm that knows about future references. However, for some benchmarks and number of page frames the competitive ratio of TNRP is remarkably good. For example, for *177.mesa* the competitive ratio of TNRP is 1.14 when the number of page frames is four and 1.03 when there are 16 page frames. It is also remarkable that the competitive ratio for *175.vpr* and *300.twolf* increases with the number of page

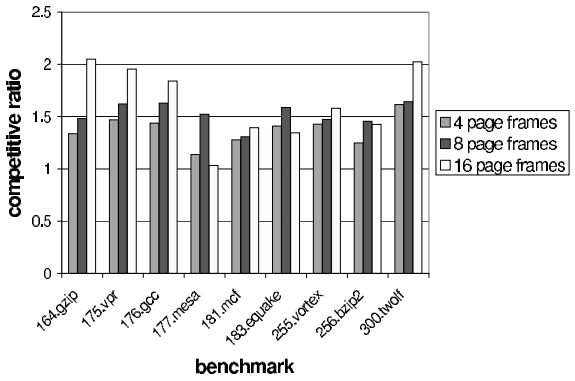


Figure 6: Competitive ratio achieved by the TNRP algorithm

frames, since the number of data pages referenced by these benchmarks is relatively small (see Table 1).

6. CONCLUSIONS

In this paper we have presented a novel page replacement algorithm called *Time of Next Reference Predictor* (TNRP). It approximates the optimal algorithm more closely than LRU by predicting the time a page will be referenced again rather than using the time of last reference.

We have evaluated the performance of the TNRP strategy using nine benchmarks from the SPEC 2000 benchmark suite. The results show that when the number of page frames is four, TNRP reduces the number of page faults significantly compared to LRU (up to 28.1% with an average of 13.5%). When the number of available page frames is larger, the improvements are less impressive. This suggests that our algorithm works best in a multi-tasking environment, where the physical memory available needs to be shared among several processes. Nonetheless, that it is possible to improve upon LRU using a simple prediction scheme is interesting on its own. Furthermore, because it takes millions of cycles to serve a page fault, even a small reduction in the miss rate should result in noticeable improvements. It has also been shown that TNRP outperforms a previously proposed algorithm, LRU-2, for most workloads and number of page frames.

We plan to continue this research in several directions. First, it may be possible to improve the algorithm. For example, the current algorithm changes the stride each time a page is referenced and returns to the initial/transient state each time the prediction turned out to be wrong. It may turn out to be advantageous to change a prediction only if it mispredicted twice, similar to a 2-bit branch predictor. This has not been investigated yet.

Second, we intend to investigate if the presented algorithm can also be used to improve the performance of second-level caches. As pointed out by Wong and Baer [15], a current trend in the design in L2 caches is increased capacity and associativity, and that increasing the associativity increases the probability that the LRU line is not the best line to replace.

7. REFERENCES

[1] Jean-Loup Baer and Tien-Fu Chen. An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty. In *Proc. Supercomputing'91*, 1991.

[2] A. Borodin, S. Irani, P. Raghavan, and B. Schieber. Competitive Paging With Locality of Reference. In *Proc. Symp. on Theory of Computing*, 1991.

[3] D. Burger and T.M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report 1342, Univ. of Wisconsin-Madison, Comp. Sci. Dept., 1997.

[4] Amos Fiat and Ziv Rosen. Experimental Studies of Access Graph Based Heuristics: Beating the LRU Standard? In *Proc. 8th ACM-SIAM Symposium on Discrete Algorithms (SODA'97)*, pages 63–72, 1997.

[5] J.W.C. Fu, J. H. Patel, and Bob L. Janssens. Stride Directed Prefetching in Scalar Processors. In *Proc. Int. Symp. on Microarchitecture*, 1992.

[6] J.L. Hennessy and D.A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, third edition, 2003.

[7] Sandy Irani, Anna R. Karlin, and Steven Phillips. Strongly Competitive Algorithms for Paging With Locality of Reference. *SIAM Journal on Computing*, 25(3), 1996.

[8] Yvon Jegou and Olivier Temam. Speculative Prefetching. In *Proc. Int. Conf. on Supercomputing*, pages 57–66, 1993.

[9] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proc. 20th Int. Conf. on Very Large Data Bases*, pages 439–450, 1994.

[10] AJ KleinOsowski, John Flynn, Nancy Meares, and David J. Lilja. Adapting the SPEC 2000 Benchmark Suite for Simulation-Based Computer Architecture Research. In *Proc. Workshop on Workload Characterization, Int. Conf. on Computer Design (ICCD)*, 2000.

[11] Nimrod Megiddo and D. S. Modha. ARC: A Self-tuning, Low Overhead Replacement Cache. In *Proc. 2nd USENIX Conf. on File and Storage Technologies*, 2003.

[12] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 297–306, 1993.

[13] D.D. Sleator and R.E. Tarjan. Amortized Efficiency of List Update and Paging Rules. *Communications of the ACM*, 28(2):202–208, 1985.

[14] R. A. Sugumar and S. G. Abraham. Efficient Simulation of Caches Under Optimal Replacement With Applications to Miss Characterization. In *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling Computer Systems*, 1993.

[15] Wayne A. Wong and Jean-Loup Baer. Modified LRU Policies for Improving Second-Level Cache Behavior. In *Proc. Int. Symp. on High-Performance Computer Architecture*, 2000.