Dynamic Buffer Cache Management Scheme based on Simple and Aggressive Prefetching*

H. Seok Jeon Sam H. Noh
Department of Computer Engineering
Hong-Ik University
Mapo-Gu Sangsoo Dong 72-1 Seoul, Korea 121-791
tel) +82-2-320-1470 fax)+82-2-320-1105
hsjeon@cs.hongik.ac.kr
samhnoh@cs.hongik.ac.kr, http://www.cs.hongik.ac.kr/~noh

Abstract

Many replacement and prefetching policies have recently been proposed for buffer cache management. However, many real operating systems, including GNU/Linux, generally use the simple Least Recently Used (LRU) replacement policy with prefetching being employed in special situations such as when sequentiality is detected. In this paper, we propose the SA-W^2R scheme that integrates buffer management and prefetching, where prefetching is done constantly in aggressive fashion. The scheme is simple to implement making it a feasible solution in real systems. In its basic form, for buffer replacement, it uses the LRU policy. However, its modular design allows for any replacement policy to be incorporated into the scheme. For prefetching, it uses the LRU-One Block Lookahead (LRU-OBL) approach, eliminating any extra burden that is generally necessary in other prefetching approaches. Implementation studies based on the GNU/Linux kernel version 2.2.14 show that the SA-W^2R performs better than the current version of GNU/Linux with a maximum increases of 23 % for the workloads considered.

1 Introduction

Considerable research for optimizing the use of buffer caches in both replacement and prefetching aspects have been undertaken. This paper proposes yet another scheme, but which has sailent features such as simple prefetching and modular integration of replacement and prefetching. These features lead to a scheme that is easily implementable, and which leads to performance improvements compared to previously known schemes.

*This work was supported by the Korea Science and Engineering Foundation grant 98-0102-09-01-3.

In the following, we first discuss the state-of-the-art in buffer cache replacement and prefetching, and then point out their limitations, which is the motivation behind the development of the proposed scheme.

1.1 Previous Research in Buffer Cache Management

Many replacement policies for improving the performance of buffer cache management have been proposed. Policies such as the Least Recently Used (LRU), Least Frequently Used (LFU), LRU-K [11], 2Q [6], Frequency-Based Replacement (FBR) [14], and Least Recently/Frequently Used (LRFU) [8] are examples.

All these replacement policies were developed independent of prefetching. Recent developments in buffer management policies has lead to the investigation of incorporating prefetching to the replacement policies. It has been shown that for some workloads incorporating prefetching can result in considerable improvement in the performance of buffer management [15, 16].

Research in incorporating prefetching can be categorized into three groups. The first group of policies maintain a history of past behavior of the applications [5, 7, 9]. This speculative approach, however, may result in performance degradation due to inaccurate prefetching and history maintenance overhead.

The second category of policies obtain hints from applications themselves prior to execution or during execution [2, 3, 4, 10, 13, 17]. Currently, these approaches appear promising as it has been shown that hints may be obtainable for specific applications with minor overhead, though their feasibility in real systems still needs to be tested.

The final approach does not require any information neither from the application nor from observations of past behavior. The LRU-OBL (One Block Lookahead) scheme [15, 16] (and its variant of prefetching multiple blocks at once) is the only scheme known to date using this approach. This scheme simply prefetches the logical next block of the currently referenced block if it is not resident in cache. It has been shown that through this simple scheme, improvements of up to 80% in the hit rate was possible for some workloads [16]. To the best of our knowledge, this approach is the only prefetch technique used in real systems due to its simplicity and effectiveness.

1.2 Buffer Cache Management in GNU/Linux

GNU/Linux adopts the LRU scheme for buffer cache management. In GNU/Linux, the *bread()* function handles the block requests. If the requested block exists in the hash table, that is, the buffer cache, it returns the block pointer. Otherwise, it makes a request for a disk I/O.

The *breada()* function is provided as a primitive for prefetching in GNU/Linux. This function is simply a variant of the LRU-OBL scheme in that it reads blocks adjacent to the requested block. However, in the GNU/Linux kernel version 2.2.14, the *breada()* function is seldom used. To the best of our knowledge, prefetching is issued only at two places. One is for reading directories in the ISO 9660 file system and the other is for the kernel thread that synchronizes the spare disk with the active disk array in RAID reconstruction.

1.3 The Remainder of the Paper

The rest of the paper is organized as follows. In the next section, we provide the motivation behind this work. In Section 3, the SA-W^2R scheme, which is the buffer cache management scheme proposed in this paper, is presented. Simulation and implementation experimental studies are presented in Section 4 and 5, respectively. Finally, Section 6 concludes with a summary and directions for further research.

2 Motivation

In this section, we discuss the motivation behind the development of the SA-W^2R scheme that is presented in the next section. To this end, we first describe the Weighing-Waiting Room (W^2R) scheme. This scheme provides the framework for an efficient integration of buffer replacement and prefetching that is simple and effective. However, its limitation restricts it from being deployed in real systems hence, providing the basis for the development of the SA-W^2R scheme.

2.1 The W^2R Scheme

The Weighing-Waiting Room (W^2R) scheme partitions the buffer cache into two rooms, that is, the Weighing Room and the Waiting Room as shown in Figure 1. The name is derived from the fact that we use the weight analogy in describing the management of the buffer cache. In general, the block to be replaced by the incoming block is the block that is considered to be the least likely to be re-referenced. This likelihood can be represented as a weight. Each block is given a weight, and the heavier block is considered more likely to be re-referenced. Then, in general, the lightest block is replaced by the incoming block as it is considered to be the least likely to be referenced again.

The Weighing Room, in the W^2R scheme, is where the weights of the blocks are contested, and rank is formed among the blocks. Only blocks that have been referenced have weights associated with them. In buffer replacement policies such as the LRU or 2Q, the whole buffer is simply the Weighing Room as only blocks that have been referenced are brought into the buffer.

The Waiting Room is where the prefetched blocks reside. Prefetching is done exactly like the LRU-OBL scheme, where the logical next block of the currently accessed block is prefetched when it is not residing in cache. They remain in this part of the buffer and wait until they obtain permission to be weighed with the other blocks. This permission is obtained when and only when the block has actually been referenced. Until this time, the prefetched blocks are weightless. Again, blocks obtain weight only when referenced as their weight cannot be determined until they have been referenced.

Note some of the features of this scheme. First, replacement (through the Weighing Room) and prefetching (through the Waiting Room) are integrated into the whole scheme, yet modularity is ensured. New replace-

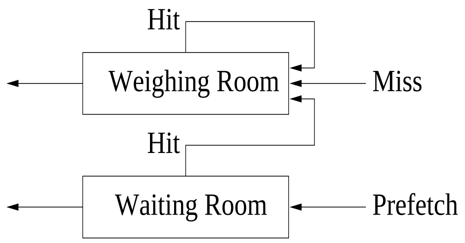


Figure 1: Structure of the W^2R scheme.

ment policies are constantly being developed. As new replacement policies that have practical significance are developed, these policies can directly be incorporated into the W^2R scheme, simply by replacing the implementation in the Weighing Room. This is unlike the LRU-OBL scheme that provide no means of doing this.

Second, by partitioning the buffer into a Weighing Room and a Waiting Room, prefetched blocks, which have not yet proven their worth, cannot replace a block in the Weighing Room, which has proven its value by being referenced. Figure 2 quantifies a deficiency in the LRU-OBL approach. This figure shows that in some situations over 60% of prefetched blocks may never be used when using the LRU-OBL scheme. Hence, in LRU-OBL, a prefetched block can hold on to valuable real estate without ever being referenced. This problem is alleviated in the W^2R scheme as blocks that are not referenced after being prefetched are not promoted to the Weighing Room. Hence, a prefetched block that is never referenced cannot replace a block that has some weight (that is, in the Weighing Room). That is to say, the wrong block may be prefetched into the buffer, but it will not replace a block that has proven its worth by having been referenced.

Third, note that a prefetched block enters the Weighing Room only after it is referenced. Hence, the prefetched block does not directly replace a block that is referenced prior to the prefetched block. The block being promoted to the Weighing Room is promoted right when it is needed, and never before.

2.2 Motivation for SA-W^2R

Although the benefits of the W^2R scheme seem to be attractive, as it stands, there is a serious problem that needs to be resolved in order for it to be deployable in real systems. The obvious and difficult problem is how to partition a fixed-size buffer cache into the two rooms. By introducing the Waiting Room, we are in effect, reducing the size of the Weighing Room compared to conventional buffer management policies. Hence, we

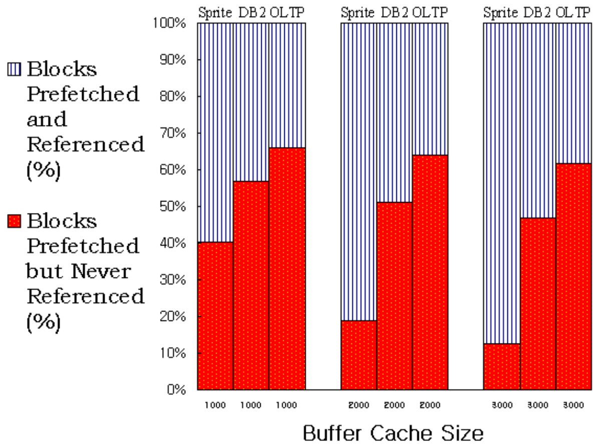


Figure 2: Percentage of prefetched blocks, using the LRU-OBL scheme, that are never referenced for the Sprite, DB2, and OLTP traces.

want to keep the Waiting Room small. However, once a block is prefetched we would like to hold it in the Waiting Room just enough so that it is eventually referenced. That is, if the Waiting Room is too small a prefetched block may be evicted too early to be of any help to the system. A judicious selection of the room size is necessary for efficient management of the buffer. This problem is addressed in the SA-W^2R scheme.

3 Self-Adjusting W^2R

The optimal partition ratio between the Weighing Room and the Waiting Room will vary according to the system environment and workload. To reiterate the point mentioned above, we want to keep the Waiting Room small so that the deterioration in performance due to the reduced Weighing Room size is limited, but we want to keep it big enough so that the prefetched blocks are used, instead of being evicted from the Waiting Room. Hence, the partitioning should accommodate the workload characteristics such that the performance benefits obtained by increasing the Waiting Room outweighs the loss incurred by the reduced Weighing Room. For all practical purposes, this should be determined on-line and must be done with minimal overhead. The Self-Adjusting W^2R (SA-W^2R) scheme attempts to do both.

The SA-W^2R adjusts the partitioning between the Weighing and Waiting Rooms via a two-step process, namely, interval-based and fault-based adjustment steps as shown in Figure 3. We discuss the two steps in

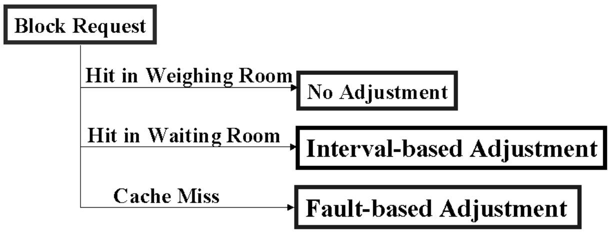


Figure 3: The SA-W^2R Scheme.

the following subsections.

3.1 Interval-based Adjustment

The Waiting Room, as the name implies, is where the prefetched blocks wait to be referenced. In other words, the role of the Waiting Room is to maintain the prefetched blocks until they are actually referenced. Recall that prefetched blocks are weightless, hence they are managed in a FIFO queue; the newly prefetched block is put at the rear (position 1) of the queue, which pushes the block at the head (position n) of the queue out of the FIFO queue, where n is the size of the Waiting Room. Let us define the position in the FIFO queue at which a block is actually referenced to be the *reference interval* of that block. If the block is evicted from the Waiting Room without being referenced, then the reference interval of that block is ∞. This reference interval will vary from workload to workload, and in adjusting the room sizes the adjustment should be such that the majority of the blocks have reference intervals less or equal to n for as small an n as possible. Hence, given a certain n value, if the reference intervals of blocks are getting smaller and smaller, then n should also be adjusted to become smaller, so that the loss in the Weighing Room can be minimized. On the other hand, if the reference intervals of blocks are getting larger and getting close to or surpasses n, then n should be increased, so as to increase the benefits of the Waiting Room.

The SA-W^2R scheme tunes the Waiting Room size by maintaining the reference interval values of the last k blocks that are referenced in the Waiting Room. (To minimize bookkeeping overhead, we set k to 3.) Using this information, we observe the trend of the reference intervals as shown in Figure 4. If the reference interval of the last k blocks show an increasing trend, then the Waiting Room is increased to accommodate the reference interval increase. Similarly, if the reference intervals show a decreasing trend, the Waiting Room is decreased. If the reference intervals of the last k blocks do not show any regularity, the Waiting Room size remains unchanged.

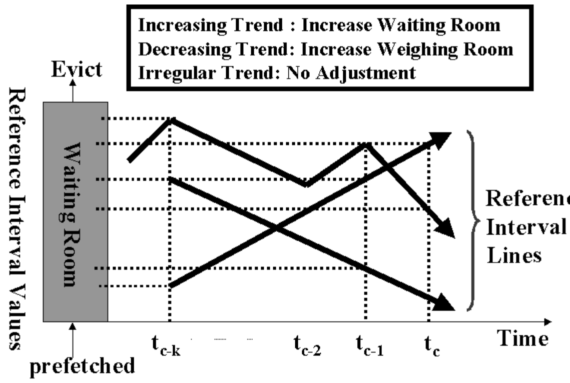


Figure 4: Interval-based Adjustment.

3.2 Fault-based Adjustment

Interval-based adjustments cannot be ideal as it adjusts the Waiting Room size based only on observations made in the Waiting Room. Adjustments of the two rooms can also be made based on hints provided by misses that occur upon a block reference. Consider the following situation. Block i is referenced, but it is not in the buffer, hence a miss occurs. Since we are prefetching the logical next block, we can deduce considerable information based on the availability of blocks $i-1$ and $i+1$ in the buffer cache. For example, if we find that block $i-1$ is in the Weighing Room and that block i is in disk, then we know that block i was evicted from the buffer cache as block i would have been prefetched with block $i-1$.

The SA-W^2R scheme exploits these fault-based information for adjusting the room sizes. It considers the situation when a request for block i is a miss. When a miss occurs on block i, nine possible situations can arise depending on the location of blocks i, $i-1$, and $i+1$ as shown in Table 1. Since a miss occurred on block i, it is in disk. At this point, blocks $i-1$ and $i+1$ can be in the Weighing Room, the Waiting Room, or the disk. Table 1 also shows the adjustments that are made for each of the nine cases.

Let us now consider how these adjustment recommendations came about. Let us start with cases 1 and 3, which have in common block $i-1$ in the Weighing Room. (Case 2 also falls into this category, but we will consider this case separately later.) The fact that block $i-1$ is in the Weighing Room tells us that block i had once been in the Waiting Room before being evicted. It may also have been in the Weighing Room before being evicted, but the fact that block $i+1$ is not in the Waiting Room makes it likely that block i was in the Waiting Room before being evicted. (Note that we are

Table 1: Nine situations in relation to the locations of blocks i, $i-1$, $i+1$ when a miss for block i occurs.

Cases	Weighing Room	Waiting Room	Disk	Adjustment Made
case 1	i-1, i+1		i	Increase Waiting Room
case 2	i-1	i+1	i	Increase Weighing Room
case 3	i-1		i, i+1	Increase Waiting Room
case 4	i+1	i-1	i	No Adjustment
case 5		i-1, i+1	i	Increase Weighing Room
case 6		i-1	i, i+1	No Adjustment
case 7	i+1		i-1, i	No Adjustment
case 8		i+1	i-1, i	Increase Weighing Room
case 9			i-1, i, i+1	No Adjustment

referring to likely scenarios that could have occurred and are not guaranteeing such scenarios.) Hence, we conjecture that block i was evicted before being referenced because the Waiting Room was too small. So, we increase the Waiting Room size.

Now consider cases 5 and 8. This is the opposite of the previous situation. The fact that block $i+1$ is in the Waiting Room tells us that block i was in the Weighing Room. This means that the Weighing Room had to evict a block that was to be referenced soon in the future, meaning that the Weighing Room was too small. Hence, adjustments to increase the Weighing Room size is made.

Let us now consider case 2. Case 2 satisfies both of the two previous scenarios, that is, block $i-1$ is found in the Weighing Room and block $i+1$ is found in the Waiting Room. Note, however, that the implications are different. While having block $i-1$ in the Weighing Room only suggests that block i could have been evicted from the Waiting Room, having block $i+1$ in the Waiting Room tells us that block i must have been in the Weighing Room when it was evicted. Hence, the Weighing Room is increased in this situation.

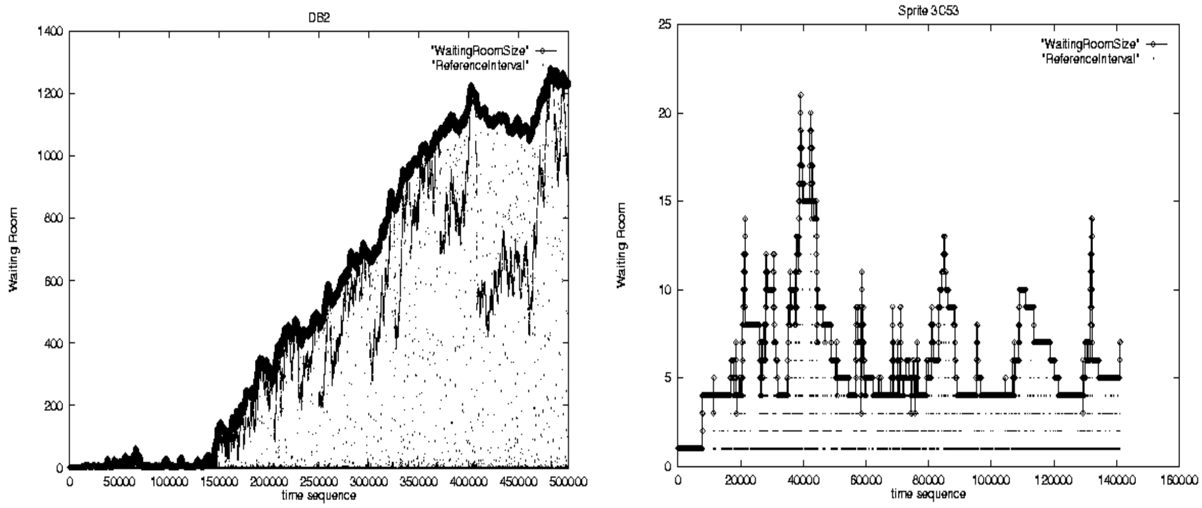
For cases 4, 6, 7, and 9, no solid relation can be deduced. For example, take case 4. The fact that block $i+1$ is in the Weighing Room suggests that block $i+2$ could be in the Waiting Room, but nothing in relation to block i can be deduced. Likewise, the fact that block $i-1$ is in the Waiting Room suggests that block $i-2$ may still be in the Weighing Room, but again, nothing in relation to block i may be deduced. Hence, for these cases no adjustments are made.

Based on these situations and their adjustments, the SA-W^2R scheme adjusts the partitioning of the buffer cache between the Weighing and Waiting Rooms.

3.3 Adaptability of the SA-W^2R Scheme

Figure 5 shows how the SA-W^2R adapts to the changing workload of the system when the cache size is 3000 for the DB2 and Sprite 3C53 traces that will be explained in the next section. The dark line that looks like the upper boundary of the figure shows the Waiting Room size at each time point. Each dot within the "boundary" represents the access point of a block, that is, the reference interval within the Waiting Room at each time point. For the DB2 trace, the streaky lines going up within the boundary is showing that the reference interval is increasing, while the streaky lines going down shows that the reference interval is decreasing. The figure shows that SA-W^2R is adjusting the Waiting Room as needed.

Figure 5(b) is actually more interesting. Note that the size of the Waiting Room is much smaller compared to the DB2 trace. This is because there is much more sequentiality in this trace. When references are sequential there is no need to increase the Waiting Room. In fact, if the workload is totally sequential, a Waiting Room size of one is sufficient. Hence, for this trace, the SA-W^2R scheme is keeping the Waiting Room size small. The seemingly horizontal lines in the figure show that a majority of the blocks are being referenced at a particular point in the Waiting Room, that is, the reference interval is constant. The lowest horizontal line represents total sequentiality, while horizontal lines above this line show that the reference interval grows, but remains constant over time. The Waiting Room size is adjusted to reflect this change.



(a) DB2

(b) Sprite 3C53

Figure 5: Adaptability of the SA-W^2R scheme for DB2 and Sprite 3C53 traces when the cache size is 3000.

4 Simulation Experiments

In this section, we discuss the trace driven simulation experiments conducted to evaluate the SA-W^2R scheme. A description of the traces that were used is given in the next subsection. In the subsequent subsection, we report and discuss the results of these experiments.

4.1 The Simulator and Traces

The simulator developed to evaluate the schemes is programmed in C++. The basic component in this simulator is the buffer cache module which takes the traces as input. The buffer cache module checks if the block number is in the buffer. If it is a hit, appropriate action, which is dependent on the policy used, is taken. Otherwise, a block fetch request action to the disk is emulated. The block size, size of the buffer, and the policy used for managing the buffer are controllable parameters.

A wide range of traces are used to drive the simulator to show the robustness of the SA-W^2R scheme. Specifically, database traces, the Sprite traces, traces of real application programs, and synthetic traces that show Zipfian distribution are used. Detailed descriptions of these traces are given below.

Database traces: Two traces, namely, DB2 and OLTP, obtained from database systems were used. These traces are identical to the traces used in the papers by Johnson and Shasha [6] and by O'Neil and others [11]. The DB2 trace is obtained from running a DB2 commercial application and contains 500,000 block requests to 75,514 distinct blocks. Obtained from the On-Line Transaction Processing System, the OLTP trace contains records of block requests to a CODASYL database for a window of one hour. It contains a total of 914,145 requests to 186,880 distinct blocks.

Sprite traces: Sprite traces are traces obtained from 4 file servers and 40 clients running the Sprite distributed file system [12]. This file system environment had roughly 30 users who were consistent users with an additional 40 some users who used the system occasionally. Traces were obtained for eight separate periods of 24 or 48 hours. These traces are considered to represent scientific workloads as most of the users were operating system researchers, computer architecture researchers, VLSI circuit designers, parallel processing researchers, etc. Of these traces, the traces that are used in our experiments are traces taken on the 2nd and 3rd periods. The trace that we denote as Sprite 2C39, which is client 39 of the 2nd period, consists of a total of 141,233 block accesses to 19,990 distinct blocks, while the trace that we refer to as Sprite 3C53, which is client 53 of the 3rd period, consists of a total of 239,748 block accesses to 49,277 distinct blocks. Though these two traces were taken from the same system, they themselves are quite different in their characteristics. According to Baker and others [1], the traces in the 2nd period represent general scientific access patterns, while for the 3rd period the workload consisted largely of accesses to very large files making it different from the general workload of the 2nd period.

Application traces: These set of traces, obtained from executing real application programs, are those used in a previous study [3]. The length of these traces are very short compared to the database and Sprite traces ranging from roughly four thousand to thirty-five thousand block accesses. Specific details regarding the characteristics of these applications are given below.

cpp: Cpp is the GNU C-compatible compiler preprocessor. The kernel source was used as input with the size of header files and C-source files of about 1MB and 10MB, respectively.

link: Link is the Unix link-editor. This application is used to build the FreeBSD kernel from about 2.5MB of object files.

ld: Ld is the trace of a linking editor. It has random accesses for both reads and writes. There is no reuse of data, but since the size of a read request is not always 8K, there are occasional reuses at the block level. (A block is 8K bytes.)

XDataSlice: XDataSlice is obtained from a 3D volume rendering software working on a $220 \times 220 \times 220$ data volume, rendering 22 slices with stride 10, along the X axis, then Y axis, then Z axis. This trace accesses blocks in a file with regular strides. There is no reuse of data when rendering along one axis, but moderate reuse is done between rendering along different axes.

Zipfian traces: The Zipfian traces are synthetic traces correspondent with a Zipfian distribution of reference frequencies. The Zipfian distribution of reference frequencies is where the probability for referencing a page with block number less than or equal to i is $(\frac{i}{N})^{\frac{log a}{log b}}$ with constants a and b between 0 and 1. The notion of constants a, b, and N is that fraction a of the references accesses fraction b of the N blocks. We generated two types of distributions referred to as ZipfianA and ZipfianB. The ZipfianA trace contains 500,000 references to 75,514 distinct blocks with $a = 0.8$, $b = 0.2$ and $a = 0.7$, $b = 0.3$. ZipfianB contains 914,145 requests to 186,880 distinct blocks with constants a and b the same as that of the ZipfianA trace. The Zipfian distribution and their respective constants were taken as they are known to be a good representation of database reference patterns [11]. The number of requests and distinct blocks were selected to be the same as the DB2 and OLTP traces, respectively.

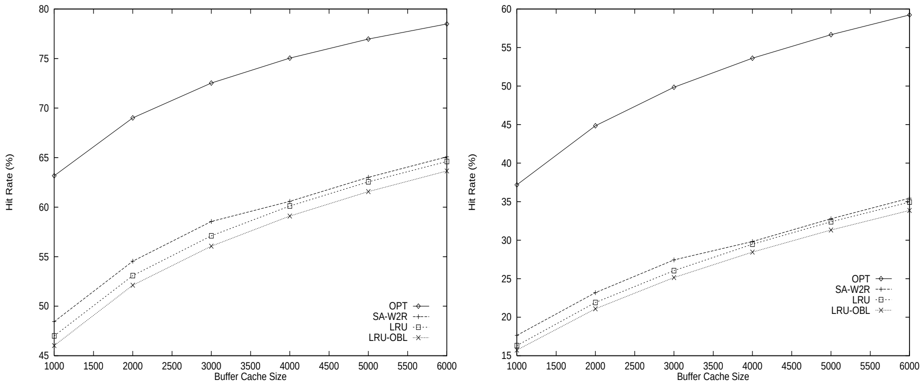
4.2 Results

Figure 6 shows the hit rates for the synthetic workload that shows Zipfian distribution. These workloads are interesting because no sequentiality is found, and we believe this represents one extreme end of reference characteristics. The results using the ZipfianA trace, shown in Figure 6, show that the LRU-OBL scheme is certainly not the choice. (The results are similar for the ZipfianB traces.) It performs even worse than the traditional LRU replacement policy. SA-W^2R shows consistently better performance than both the LRU and LRU-OBL.

Now, also consider how the schemes deal with purely sequential references, which is the other extreme end of reference characteristics. The LRU replacement policy will incur misses on every reference to a new block resulting in zero hit rate, while for both the LRU-OBL and the SA-W^2R schemes, the hit rate will approach 100% as every logical next block will be prefetched. The results of the two extreme reference characteristics show that the SA-W^2R is a versatile scheme.

Figures 7, 8, and 9 show the hit rates for the SA-W^2R scheme compared with other schemes for real workloads. Regarding the figures, first note that the scales are all different. Also, for all these figures that do not show the hit rates for LRU and/or OPT (the optimal replacement) policies, they are not shown because their margin of difference from the LRU-OBL and SA-W^2R is so large that it makes the lines indecipherable. Hence, we omitted the LRU and/or OPT lines for these cases.

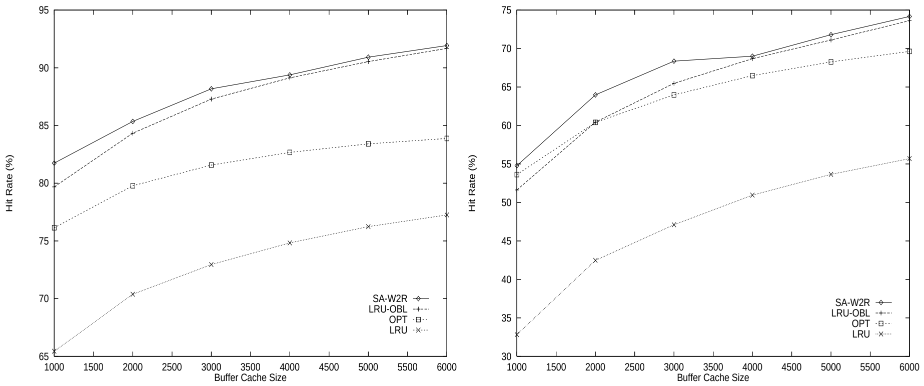
Overall, the performance of the SA-W^2R scheme performs better than all the others. An interesting observation of these results is that the LRU-OBL scheme is superior to the OPT policy. Except for the extreme case where there is only minimum or no sequentiality, the LRU-OBL policy is a good general scheme that could be used for general buffer cache management, and not just limited to sequential reference patterns. Of course, one has to consider how the hit rate performance measure translates to other measures such as response time in real systems. Extra queueing delays incurred by prefetching may limit the performance benefits observable by the user. However, with the advent of RAID



(a) ZipfianA 80_20 distribution

(b) ZipfianA 70_30 distribution

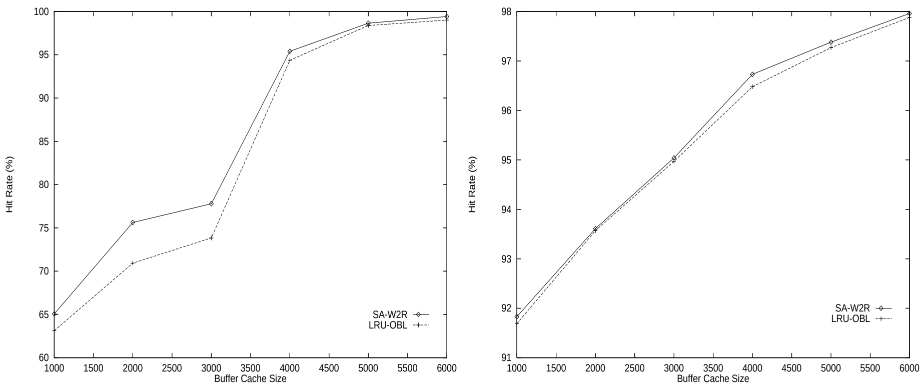
Figure 6: The hit rates of the SA-W^2R scheme for the ZipfianA traces.



(a) DB2

(b) OLTP

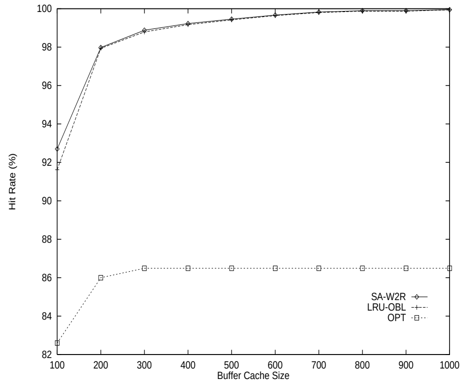
Figure 7: The hit rates of the SA-W^2R scheme for the DB2 and OLTP traces.



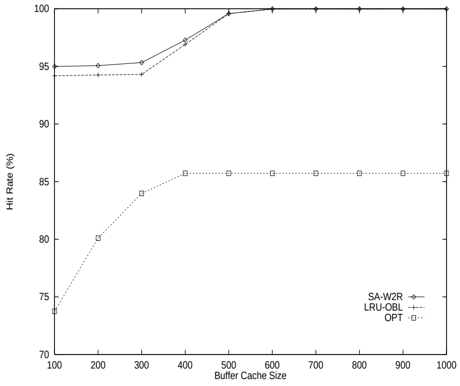
(a) Sprite 2C39

(b) Sprite 3C53

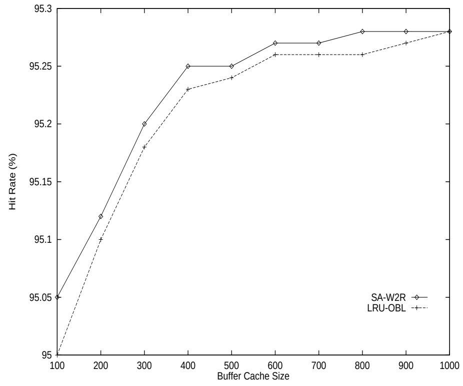
Figure 8: The hit rates of the SA-W^2R scheme for the Sprite traces.



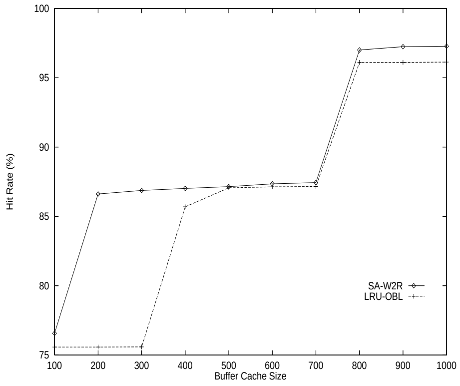
(a) cpp



(b) link



(c) ld



(d) XDataSlice

Figure 9: The hit rates of SA-W^2R scheme for real application traces.

systems and better caching techniques, delays due to this type of queueing should not have aggravated influence on the performance. Hence, hit rates should be a good reflection of the actual performance seen by the user for these schemes.

The SA-W^2R is an even better scheme than the LRU-OBL performing consistently better than the LRU-OBL for all situations including the extreme cases mentioned previously. The maximum performance difference comes from the XDataSlice application where the SA-W^2R has a hit rate that is over 11 percentage points higher than the LRU-OBL scheme. Overall, the performance improvement range somewhere around the 1 to 4 percentage point increase compared to the LRU-OBL.

5 Implementation Experiments

The SA-W^2R scheme was implemented on the GNU/Linux kernel version 2.2.14 on a Pentimum III 430 MHz PC with 128 MB of memory. (*At the time of this implementation, the kernel version 2.2.14 was the latest stable version.*) For performance comparison purposes, we also implemented the LRU-OBL scheme. The applications used to evaluate the performance are as follows.

gcc: Compile the GNU/Linux kernel version 2.2.14.

cp: Copy the whole GNU/Linux source code from /usr/src/linux directory to another directory.

tar: Create/extract the tar file for the GNU/Linux source code.

gzip: Compress/uncompress the tar file of the GNU/Linux source code.

grep: Grep the string "linux" from the /usr/src/linux directory.

sort: Sort 1,000,000 random data items.

5.1 Individual Application Performance

Table 2 shows the execution time of each application. The results shown in the table are averages of three executions of each application. Before each execution of each application, the system was rebooted to eliminate the effect of caching from the previous execution.

The results show that the SA-W^2R scheme shows the best performance compared to the original GNU/Linux and LRU-OBL schemes. Specifically, the performance improvements due to the proposed scheme range between 5 to 23 percent compared to the original GNU/Linux scheme.

Note also that the LRU-OBL scheme always performs considerably better than the original GNU/Linux scheme, though worse than the SA-W^2R. This is because regular reference patterns such as sequential references is a dominant characteristic of many of these applications. Hence, it may be argued that this set of applications, specifically, this characteristic, unfairly favors the LRU-OBL and SA-W^2R schemes. To show that the SA-W^2R scheme is a robust solution, we executed with each application a 'random' process that randomly references blocks such that it disrupts regular reference sequences such as sequential block references. Those results are shown in Table 3.

The results in Table 3 show that when regular reference behavior is disrupted LRU-OBL may perform worse than the original GNU/Linux management scheme. It also shows, however, that the SA-W^2R scheme consistently performs better though its improvement is now somewhat smaller. This shows that the SA-W^2R is quite robust in its management of the buffer.

5.2 Concurrent Execution of Multiple Applications

Using the same methodology for the experiments, we measured the performance of applications when multiple applications were executed concurrently. Table 4 shows the applications that were executed concurrently (Groups 1 to 3) and their respective execution times using the different buffer management schemes.

As the concurrently executing applications influence the buffer and CPU resource allocation, the execution times of the applications increase considerably. Again, in all of the situations, the SA-W^2R scheme shows the best performance. The improvements range from negligible (approximately 1% for gzip (compress) of Group 2) to an approximately 20% reduction (for the gzip (uncompress) of Group 3) in execution time.

To measure the overhead of the bookkeeping and management of information for adjusting the room sizes, we added a CPU bound process that continuously does simple add operations to each of the groups of applications. The results of these experiments are shown in Table 5. Note that the effect of the CPU bound process on the execution of each application depends on the characteristics of the applications that are executing. For applications of Group 1, the increase in the execution time is small, while for those of Group 2, the increase is substantial. (This can be observed by comparing the execution times of Tables 4 and 5.) Note though, that the increase in execution time of the CPU bound process (for Groups 1 and 3) are quite small, implying that the overhead for maintaining relevant information for dynamic room partitioning is quite small.

6 Conclusion and Future Works

In this paper, we proposed the SA-W^2R scheme that is in line with the LRU-OBL scheme, that is, a simple and practical scheme that integrates prefetching and replacement policies. It is simple and practical, and yet it is modular in that any replacement policy deemed appropriate may be incorporated into the scheme. Simplicity results in a scheme that is easy to implement, hence practical.

An extensive implementation study was done, and experimental results show that the SA-W^2R shows better performance compared to the original GNU/Linux and LRU-OBL implementations.

Issues such as quantifying the benefits of the modularity of the Weighing Room or the effect of hint-based prefetching on the SA-W^2R scheme are being considered. Performance of prefetching is also strongly influenced by the performance of the disk system as well, and, thus, their interaction must be studied more closely.

Table 2: Average execution time for applications using different buffer management schemes (in seconds).

Applications	Original Linux	LRU-OBL	SA-W^2R
gcc	244	241	230
cp	59.40	53.31	48.71
tar (create)	61.02	59.87	55.31
tar (extract)	41.70	39.93	36.26
gzip (compress)	72.87	64.73	56.11
gzip (uncompress)	21.45	19.99	17.23
sort	47.32	45.14	42.57
grep	46.92	38.28	37.01

Table 3: Average execution time for applications with a 'random' process disrupting regular reference behavior using different buffer management schemes (in seconds).

Applications	Original Linux	LRU-OBL	SA-W^2R
gcc	394.15	393	387.24
cp	302.32	304.64	297.57
tar (create)	312	304.89	298.99
tar (extract)	46.55	49.22	43.58
gzip (compress)	76.93	69.17	66.12
gzip (uncompress)	22.47	22.34	20.84
sort	54.55	55.29	54.25
grep	294.08	277.59	272.65

Table 4: Average execution time for groups of applications executed concurrently (in seconds).

	Applications	Original Linux	LRU-OBL	SA-W^2R
Group 1	cp	325.73	318.44	313.23
	tar (create)	329.57	323.79	319.53
Group 2	gzip (compress)	95.67	98.26	94.35
	sort	91.80	92.06	88.56
Group 3	tar (extract)	77.96	83	75.43
	gzip (uncompress)	59.70	50.91	47.54
	grep	130.42	127.42	122.55

Table 5: Average execution time of groups of applications executing concurrently with a CPU bound process (in seconds).

	Applications	Original Linux	LRU-OBL	SA-W^2R
	CPU bound process only	222	222	222
Group 1	cp	329.57	315.82	311.94
	tar (create)	331.95	319.74	314.65
	CPU bound process	235	235	236
Group 2	gzip (compress)	138.77	143.08	137.52
	sort	138.74	141.71	137.17
	CPU bound process	315.07	315.47	314.96
Group 3	tar (extract)	90.8	91.58	85.26
	gzip (uncompress)	79.83	61.32	58.01
	grep	146.02	138.91	129.57
	CPU bound process	246.01	247.51	248.1

7 Acknowledgement

The authors would like to thank Gerhard Weikum and Theodore Johnson for providing us with the DB2 and OLTP traces and Jongmoo Choi for helping with the application traces. We would also like to thank our shepherd, Stephen C. Tweedie, for helping us finalize this paper.

References

[1] Mary G. Baker, John H. Hartman, Michael D. Kupfer, KenW. Shirriff, and John K. Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th ACM SOSP*, pages 198–212, Pacific Grove, CA, October 1991.

[2] Pei Cao and Edward W. Felten. Implementation and Performance of Integrated Appplication-Controlled File Caching, Prefetching, and Disk Scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, November 1996.

[3] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. In *Proceedings of 1995 Joint ACM SIGMETRICS and Performance Evaluation Conference*, pages 188–197, 1995.

[4] Fay Chang and Garth A. Gibson. Automatic I/O Hint Generation through Speculative Execution. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation*, pages 1–14, February 1999.

[5] K. Curewitz, P. Krishnan, and J.S. Vitter. Practical Prefetching via Data Compression. In *Proceedings of the 1993 ACM SIGMOD Conference*, pages 257–266, May 1993.

[6] Theodore Johnson and Dennis Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th VLDB Conference*, pages 439–450, 1994.

[7] David Kotz and Carla Schlatter Ellis. Practical Prefetching Techniques for Multiprocessor File Systems. *Journal of Distributed and Parallel Databases*, 1(1):33–51, January 1993.

[8] Donghee Lee, Jongmoo Choi, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies. In *Proceedings of the 1999 ACM SIGMETRICS Conference*, pages 134–143, 1999.

[9] Hui Lei and Dan Duchamp. An Analytical Approach to File Prefetching. In *Proceedings of the 1997 USENIX Annual Technical Conference*, pages 275–288, January 1997.

[10] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, October 1996.

[11] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD Conference*, pages 297–306, May 1993.

[12] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite Network Operating System. *IEEE Computer*, 21(2):23–36, February 1988.

[13] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed Prefetching and Caching. In *Proceedings of the 15th ACM SOSP*, pages 79–95, December 1995.

[14] J. T. Robinson and N. V. Devarakonda. Data Cache Management Using Frequency-Based Replcement. In *Proceedings of the 1990 ACM SIGMETRICS Conference*, pages 134–142, 1990.

[15] Alan Jay Smith. Sequential program prefetching in memory heirarchies. *IEEE Computer*, 3(3):7–21, December 1978.

[16] Alan Jay Smith. Disk cache-miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, 3(3):161–203, August 1985.

[17] Andrew Tomkins, R. Hugo Patterson, and Garth A. Gibson. Informed Multi-Process Prefetching and Caching. In *Proceedings of the 1997 ACM SIGMETRICS Conference*, pages 100–114, June 1997.