Significance of Replacement Queue in Sequential Prefetching

Elizabeth Varki
University of New Hampshire
varki@cs.unh.edu

Allen Hubbe [*]
EMC
Allen.Hubbe@emc.com

Arif Merchant [†]
Google
aamerchant@google.com

Abstract

The performance of a prefetch cache is dependent on both the prefetch technique and the cache replacement policy. Both these algorithms execute independently of each other, but they share a data structure - the cache replacement queue. The replacement queue captures the impact of prefetching and caching. This paper shows that even with a simple sequential prefetch technique, there is an increase in hit rate and a decrease in response time when the LRU replacement queue is split into two equal sized queues. A more significant performance improvement is possible with sophisticated prefetch techniques and by splitting the queue unequally.

1 Introduction

Prefetching and caching refers to a cache system that loads data blocks not yet requested by the cache workload. The goal is to ensure that the cache contains blocks that will be requested in the near-future. Prefetching and caching software essentially consists of two algorithms, namely, the prefetch technique and the replacement policy. The software is responsible for leveraging the *spatial* and *temporal* locality of reference in the cache workload.

A prefetch technique is the software responsible for identifying access patterns in the cache workload and loading data blocks from these patterns into the cache before the blocks are requested. The task of a prefetch technique is to determine what data blocks to prefetch and when to prefetch the blocks. For example, if a file is being read sequentially, the sequential prefetch technique associated with the file system cache may prefetch several blocks contiguous to the requested file block. The replacement policy is the cache software that determines which block is evicted from the cache when a new block is to be loaded into a full cache. Thus, the replacement policy is responsible for keeping a prefetch block in the cache until it is requested.

The prefetch technique and the replacement policy are standalone - capable of executing independently of each other - the prefetch technique does not decide when a prefetched block is evicted; the replacement policy does not decide when a block is prefetched. The two algorithms together determine the contents of the cache. Consequently, even though the algorithms are independent, the performance of a prefetch technique cannot be studied in isolation of the replacement policy and vice versa. The performance - cache hit ratio, response time - of a prefetch cache depends on the combined impact of both algorithms.

Prefetching and caching are fundamental to computer and network systems, so this topic has been extensively researched. However, this topic is not well understood since it is complex. There are several issues to be considered - for prefetch techniques: how much to prefetch, when to prefetch, what to prefetch; what replacement policy to use with a particular prefetch technique, and what prefetch technique to use with a particular replacement policy. A prefetch/replacement policy that works well with one workload could perform poorly with another workload. Relative performances vary in a seemingly arbitrary manner [6], so there are no optimal prefetch techniques nor optimal replacement policies for prefetch caches. Some papers have developed integrated prefetch and replacement algorithms, but these algorithms requires a priori knowledge of the workload for optimum performance. Most modern file system and storage caches implement separate prefetching techniques and replacement policies, not an integrated algorithm combining prefetch and cache replacement.

What this paper does: This paper looks at prefetching and caching from a different perspective than that of prior papers. We view prefetching and caching as a synchronization problem, similar to problems such as producer-consumer, reader-writer, and dining philoso-

[*]This work was done while Allen was in UNH.
[†]This work was done before Arif joined Google.

phers [26]. In all these problems, there are standalone algorithms that access a common resource. Similarly, in prefetching and caching, the prefetch technique and the replacement policy operate in unison on the cache. From a software perspective, the common resource is encapsulated in the replacement queue. The cache replacement queue orders cache blocks by eviction priority and it is the meta-data used by caching software. The prefetch technique controls what prefetch blocks are inserted into the replacement queue and when they are inserted, while the replacement policy controls where a prefetch block is placed in the replacement queue. This paper leverages the replacement queue to improve the performance of prefetching and caching, without changing the prefetching technique or the replacement policy.

Contributions: By analyzing the role of the replacement queue, the paper shows that the performance of a prefetch cache can be improved by merely splitting the single replacement queue into two queues. Both algorithms, the prefetch technique and the replacement policy, stay the same. This paper demonstrates the *Split* queue approach using the sequential prefetch technique and LRU, the most common of the prefetch techniques and the replacement policies [8]. The paper proves that the the Split queue always has a higher hit rate than the single queue, regardless of the workload. Evidence via simulation shows that Split results in lower response time and higher hit rate than the single queue.

2 Sequential locality

Prefetching is carried out by caches at all levels of the memory hierarchy; this papers discusses prefetching in the context of file system and storage caches. The operating system maps user read requests for bytes into read requests for blocks. Therefore, the unit of measurement used is blocks: a cache size is C blocks; the cache workload consists of user requests, where each request is for a single block.

Workload: A file/storage cache workload consists of interleaved requests from various users [2, 8, 20]. A file/storage cache workload with sequentiality is of the form:

$<$ 1001, 64, 1002, 72345, 65, 323, 66 $>$

The workload is a sequence of block numbers that represent user requests; the position in the sequence represents the relative time at which the request arrives at the cache. That is, requests for blocks 1001, 64, 1002, 72345, 65, 323, 66 arrive at times t_1, t_2, t_3, t_4, t_5, t_6, t_7, respectively.

At first glance, it is difficult to see the sequentiality in the workload. There are two interleaved *streams* of sequential requests: $<$ 1001, 1002 $>$, $<$ 64, 65, 66 $>$. The lone requests $<$ 72345 $>$, $<$ 323 $>$ may also be considered as streams - they represent the start of streams whose future requests arrive after the observation period. Thus, the example workload has four streams, namely,

stream 1: $<$ 1001, 1002 $>$,
stream 2: $<$ 64, 65, 66 $>$,
stream 3: $<$ 72345 $>$, and
stream 4: $<$ 323 $>$.

To make it easier to identify the sequentiality in the workload, we represent block numbers in a stream as follows:

stream 1: $<$ 1, 1a $>$,
stream 2: $<$ 2, 2a, 2b $>$,
stream 3: $<$ 3 $>$, and
stream 4: $<$ 4 $>$.

Thus, each block number is mapped to its stream number and its sequential position within the stream. Using this new notation, the example workload is written as follows:

$<$ 1, 2, 1a, 3, 2a, 4, 2b $>$

A number i in the workload represents the first request from stream i; the variable ia represents the next request in stream i, and so on.

Sequential prefetch techniques are broadly classified into two types [20]: *Prefetch on Miss (PM)* and *Prefetch Always (PA)*. The PM technique generates synchronous prefetch requests for blocks contiguous to the missed block whenever a user request misses in the cache. The basic PA technique generates prefetch requests whenever a user request arrives at the cache. A synchronous prefetch request is generated when a read request misses in the cache. If this synchronously prefetched block gets a hit, then an asynchronous prefetch request is generated for the next block. If a future stream block is already in the cache, both PM and PA do not fetch it again.

There are several versions of PA. In a common version of PA, implemented in Linux and BSD, a synchronous prefetch request is generated on every miss, but an asynchronous prefetch request is not generated on every hit. Several blocks are prefetched at a time, and one of the prefetched blocks in each stream is marked as a *trigger* block. An asynchronous prefetch for this stream is initiated only when the trigger block gets a hit.

Reconsider the example workload: The maximum number of sequential prefetch hits possible is three - for blocks 1a, 2a, 2b. We now present examples that illustrate that for a given workload, the prefetch technique determines (1) the order in which blocks are inserted into the cache, and (2) the number of prefetch hits that can be achieved. The examples assume that (1) the cache is large enough so that no blocks are evicted, and (2) prefetched blocks are instantaneously loaded into the cache.

Let the first prefetch technique ensure that for each user request, the next two contiguous blocks are prefetched. For the example workload, $<$ 1, 2, 1a, 3, 2a, 4, 2b $>$, the order in which requests - user requests or prefetch requests - arrive at the cache is:

$<$ 1, *1a*, *1b*, 2, *2a*, *2b*, 1a*, *1c*, 3, *3a*, *3b*, 2a*, *2c*, 4, *4a*, *4b*, 2b*, *2d* $>$.

The requests in italics are the prefetched requests. When user request 1 arrives, 1a, 1b are prefetched into the cache; when user request 1a arrives, only 1c is prefetched since 1b is already in the cache. The * represents a prefetch hit.

Now, consider a second technique that prefetches 2 blocks on miss, and prefetches 2 blocks on hit of only the last cached block of the corresponding stream. For the example workload, the order in which user/prefetch requests arrive at the cache is:

$<$ 1, *1a*, *1b*, 2, *2a*, *2b*, 1a*, 3, *3a*, *3b*, 2a*, 4, *4a*, *4b*, 2b*, *2c*, *2d* $>$.

There is no prefetching action when user request 1a arrives since 1b is in the cache; when user request 2b arrives, 2c and 2d are prefetched.

With a PM technique, where 1 block is prefetched on each miss, the order of requests is:

$<$ 1, *1a*, 2, *2a*, 1a*, 3, *3a*, 2a*, 4, *4a*, 2b, *2c* $>$.

Note that with this last technique, the workload only gets 2 prefetch hits.

Replacement policy becomes relevant when the cache is too small to hold all the blocks. For the example workload, consider a cache of size 4 blocks. Table 1 demonstrates the ordering of the LRU replacement queue with the first prefetch technique. The example workload does not display temporal (rereference) locality, so a prefetch block is moved out of the cache as soon as it receives a hit.

The LRU policy is designed for workloads that display temporal locality, but even so, the prefetch cache achieves the maximum prefetch hit rate of 3. However, in general, it can be argued that the prefetch cache would perform better if the replacement policy is aware of prefetching and sequential locality. Consequently, file system and storage caches often implement a prefetch aware version of LRU - all blocks of a stream are placed contiguously in the replacement queue and moved as a unit [10]. When a block gets a hit, all prefetched blocks from the stream are moved to the MRU end of the replacement queue. The least recently used stream blocks are evicted from the cache when a new stream is to be inserted. This version of LRU, called StreamLRU, is presented in Table 2. The computational complexity of StreamLRU is the same as that of LRU.

Note that for the example workload and this prefetch technique, LRU gets 3 prefetch hits while StreamLRU only gets 2 hits. In general, however, StreamLRU gets more prefetch hits than LRU since StreamLRU recognizes the temporal locality of streams and keeps the most recently used stream blocks in the cache. In fact, for the second prefetch technique presented earlier - prefetch 2 blocks on miss, prefetch 2 blocks on hit of last stream block - LRU gets 2 prefetch hits, while StreamLRU gets 3 prefetch hits.

3 Replacement queue

The replacement queue is the link between the prefetch technique and the replacement policy. The name “replacement” queue is misleading since it suggests that the queue belongs to (*i.e.,* is controlled by) the replacement policy. However, the prefetch technique determines what blocks are loaded into the queue, while the replacement policy determines what blocks are evicted. Therefore, both the prefetch technique and the replacement policy manipulate the queue. The queue captures the combined impact of prefetching and caching. Using this insight, our next step is to use the queue to improve the performance of prefetching and caching.

StreamLRU captures the temporal locality of streams by moving all blocks from the most recently accessed stream to the non-eviction end of the replacement queue. In order to amortize the cost of prefetching - reduce the traffic at the disks - a prefetch technique loads several blocks at a time. Not all these blocks are expected to receive user requests immediately. In fact, the sequential access pattern dictates that blocks are accessed contiguously, in sequence. When there are 2 prefetched blocks, as in our examples, the first block is expected to receive an user request before the second block. Therefore, while inserting the blocks in the replacement queue, give placement priority to the first block of every stream, thereby ensuring that the second block is evicted before the first block. The materialization of this idea to the queue is done by splitting the replacement queue into two queues and inserting the first block in the higher priority queue and the second block into the lower priority queue - we call this the split queue technique.

3.1 Split

Split divides the single replacement queue into two queues, the Up queue and the Down queue - a block evicted from the LRU end of the Up queue is inserted into the MRU end of the Down queue; all evictions from the prefetch cache are from the LRU end of the Down queue. When two blocks of a stream are prefetched, the earlier (*i.e.,* first) block of the stream is inserted into the Up queue; if any block is evicted from the LRU end of the Up queue then this block is inserted into the MRU

Table 1: LRU, 1st prefetch technique: ensures that for each request the next 2 contiguous blocks are loaded.

Prefetch cache size = 4, LRU							
hits			h1		h2		h3
workload	**1**	**2**	**1a**	**3**	**2a**	**4**	**2b**
rep.Queue	1a	2a	1c	3a	2b	4a	2d
	1b	2b	2a	3b	2c	4b	4a
		1a	2b	1c	3a	2b	4b
		1b	1b	2a	3b	2c	2c
eject				**2b**	**1c**	**3a**	
				1b		**3b**	

Table 2: StreamLRU, 1st prefetch technique: ensures that for each request the next 2 contiguous blocks are loaded.

Prefetch cache size = 4, StreamLRU							
hits			h1				h2
workload	**1**	**2**	**1a**	**3**	**2a**	**4**	**2b**
rep.Queue	1a	2a	1b	3a	2b	4a	2c
	1b	2b	1c	3b	2c	4b	2d
		1a	2a	1b	3a	2b	4a
		1b	2b	1c	3b	2c	4b
eject				**2a**	**1b**	**3a**	
				2b	**1c**	**3b**	

Table 3: SplitLRU, 1st prefetch technique: ensures that for each request the next 2 contiguous blocks are loaded. Split gives the same number of hits as LRU and more hits than StreamLRU.

Prefetch cache size = 4, SplitLRU							
hits			h1		h2		h3
workload	**1**	**2**	**1a**	**3**	**2a**	**4**	**2b**
rep.Up Q	1a	2a	1b	3a	2b	4a	2c
		1a	2a	1b	3a	2b	4a
rep.Down Q	1b	2b	1c	3b	2c	4b	2d
		1b	2b	2a	1b	3a	4b
eject				**1c**	**3b**	**2c**	**3a**
				2b		**1b**	

Table 4: SplitLRU, 2nd prefetch technique: ensures that 2 contiguous blocks are prefetched on miss and on hit of last cached stream block. Split gives the same number of hits as StreamLRU and more hits than LRU

Prefetch cache size = 4, SplitLRU							
hits			h1		h2		h3
workload	**1**	**2**	**1a**	**3**	**2a**	**4**	**2b**
rep.Up Q	1a	2a	1b	3a	2b	4a	2c
		1a	2a	1b	3a	2b	4a
rep.Down Q	1b	2b	2b	3b	2c	4b	2d
		1b		2a	1b	3a	4b
eject				**2b**	**3b**	**2c**	**3a**
						1b	

end of the Down queue; finally, the later (*i.e.*, second) block of the newly prefetched stream is inserted into the MRU end of the Down queue. Split, like StreamLRU, assumes that the replacement policy recognizes streams, but unlike StreamLRU (and like LRU), Split does not require that all stream blocks be placed together in the replacement queue.

Table 3 demonstrates Split LRU using the example workload and the first prefetch technique presented in the last section. In the example, when a request arrives for prefetched block $1a$, the block $1a$ is removed from the prefetch cache (since it is now referenced). The prefetch cache contains block $1b$. Since the replacement policy is prefetch aware LRU, block $1b$ is moved from the Down queue to the insertion end of the Up queue, and the newly prefetched block $1c$ is inserted into the insertion end of the Down queue. When a request arrives for block 3, blocks $3a, 3b$ are prefetched; block $3a$ is inserted into the Up queue which results in block $2a$ being evicted from the Up queue; this block is inserted into the Down queue, and then the second block from stream 3, $3b$, is inserted into the Down queue.

Table 4 demonstrates Split LRU using the example workload and the second prefetch technique presented in the last section. Due to space limitations, LRU and StreamLRU's performance with this second prefetch technique is not shown. With this prefetch technique too, Split achieves the maximum prefetch hit rate of 3.

Summarizing, Split incorporates the assumptions made by the replacement policy and the prefetch technique. The replacement policy, StreamLRU, assumes that recently accessed streams will receive requests again; the prefetch technique prefetches several blocks to reduce cost of prefetch, but assumes that earlier stream blocks will receive requests before the later blocks. By moving the later blocks in a stream to the lower priority queue, Split mimics Belady's optimum MIN replacement policy [4] - first throw out the block needed in the most distant future. The power of Split lies in the recognition that the queue captures the actions of both algorithms, so the independent actions of the two algorithms can be synchronized to work toward the common goal of keeping blocks required in the near future in the cache.

4 Theory

We theoretically compare the performances of LRU, StreamLRU and SplitLRU. The prefetch technique ensures that $x \geq 2$ blocks contiguous to the latest workload request are in the cache. For simplicity, let x be an even number, $x/2$ stream blocks are loaded into the Up queue, while the latter $x/2$ stream blocks are loaded into the Down queue. We refer to the first $x/2$ blocks of a stream as the first half, and the second $x/2$ blocks as the last half of the stream. The cache size is C blocks, where C is a multiple of $x/2$.

Prefetching is about streams, so all results pertain to streams. If earlier blocks of a stream are evicted from the cache, but the cache holds later blocks of the stream then the next workload request for a block from this stream will miss; thus, the stream is effectively evicted from the cache. Let $\#_{\mathsf{Stream}}$, $\#_{\mathsf{Split}}$ and $\#_{\mathsf{LRU}}$ represent the number of streams in the cache when the replacement policy is StreamLRU, SplitLRU and LRU, respectively. Let $\{\mathsf{Stream}\}$, $\{\mathsf{Split}\}$ and $\{\mathsf{LRU}\}$ represent the set of streams in the cache when the replacement policy is StreamLRU, SplitLRU and LRU, respectively.

We provide informal explanations for all the results, but due to space limitations, we provide proofs only when the informal explanation is insufficient. The next result follows from the definition of StreamLRU, namely, moving the entire stream to the MRU end upon a hit.

Result 1 *StreamLRU ensures that the most recently accessed* $\lceil \frac{\mathsf{C}}{x} \rceil$ *streams are in the cache.*

$$\#_{\mathsf{Stream}} = \lceil \frac{\mathsf{C}}{\mathsf{x}} \rceil$$

Result 2 *Split LRU ensures that at least* $\lceil \frac{\mathsf{C}}{x} \rceil$ *and at most* $\lceil \frac{\mathsf{C}}{x} \rceil + \lfloor \frac{\mathsf{C}}{2x} \rfloor$ *of the most recently accessed streams are in the cache.*

$$\lceil \frac{\mathsf{C}}{x} \rceil \leq \#_{\mathsf{Split}} \leq \lceil \frac{\mathsf{C}}{\mathsf{x}} \rceil + \lfloor \frac{\mathsf{C}}{2\mathsf{x}} \rfloor$$

Proof: We prove the result by showing that between two accesses to a stream it is possible to have $\lceil \frac{\mathsf{C}}{x} \rceil + \lfloor \frac{\mathsf{C}}{2x} \rfloor - 1$ accesses to unique streams.

The Up queue size contains $\lceil \frac{\mathsf{C}}{x} \rceil \times \frac{x}{2}$ blocks, while the Down queue contains $\lfloor \frac{\mathsf{C}}{x} \rfloor \times \frac{x}{2}$ blocks.

The Split policy ensures that the first half of the most recently accessed $\lceil \frac{\mathsf{C}}{x} \rceil$ are in the Up queue. With each new first half stream insertion into the Up queue, an Up block moves $x/2$ positions toward the eviction end. Hence, the first half of a stream stays in the Up queue for at least $\lceil \frac{\mathsf{C}}{x} \rceil - 1$ unique stream accesses.

When a block is ejected from the Up queue, it is moved to the Down queue, behind the second half of the newly inserted stream. When a new stream is inserted, 2 stream halves are inserted into the Down queue - the second half from the new stream and the first half evicted from the Up queue. Therefore, with each stream insertion, every Down block moves x positions downward toward the eviction end. Thus, a block stays in the Down queue for $\lfloor \frac{\mathsf{C}}{2x} \rfloor$ unique stream accesses.
$\square$

The next results follow directly from Results 1 and 2.

Corollary 1 *With respect to streams, the StreamLRU cache is a subset of the Split cache. That is,* $\{\mathsf{Stream}\} \subseteq \{\mathsf{Split}\}$.

Theorem 1 *The hit rate of a Split LRU cache is an upper bound to the hit rate of a StreamLRU cache.*

We now compare LRU against SplitLRU and StreamLRU. The prefetch technique ensures that x blocks contiguous to the workload request are loaded in the cache. New prefetch blocks are inserted into the MRU end of the cache, but if the cache already contains a contiguous block, this block retains its position in the replacement queue. In Table 1, when the workload request is 1a, the prefetch technique loads 1c into the MRU end, but block 1b retains its position at the eviction end of the queue. The prefetch blocks are evicted in FIFO order by LRU. The older contiguous blocks would get evicted before the newer contiguous blocks. Thus, LRU may evict the first half of a stream before the second half of this stream. As a result, it is possible for a workload request to get a miss even though a later block of the stream is present in the cache. A consequence of evicting prefetch blocks in FIFO order is:

Result 3 *LRU evicts streams in FIFO order and holds at most* $\lceil \frac{\mathsf{C}}{x} \rceil$ *streams in the cache.*

$$\#_{\mathsf{LRU}} \leq \lceil \frac{\mathsf{C}}{\mathsf{x}} \rceil$$

Note that StreamLRU and SplitLRU keep the most recently accessed streams in the cache. The LRU policy evicts streams in FIFO order, so the streams in the LRU cache are not necessarily the most recently accessed streams. The theoretical analysis in this section allows one to understand the essential traits of LRU, StreamLRU and SplitLRU. However, we need to evaluate whether the superior hit rate of Split translates into lower response time. In the next section, we address this issue by a simulation analysis of the relative performances of the single queue and split queue approach.

5 Experiments

We developed a simulator to evaluate Split; the front end model is our cache simulator, while the back end storage model is the Disksim 4.0 simulator. Table 5 gives the setup used for our experiments. The replacement policy is LRU, and the prefetch technique is the 2nd one presented in Section 2: prefetch the next 2 contiguous blocks on miss and on hit of last cached stream block.

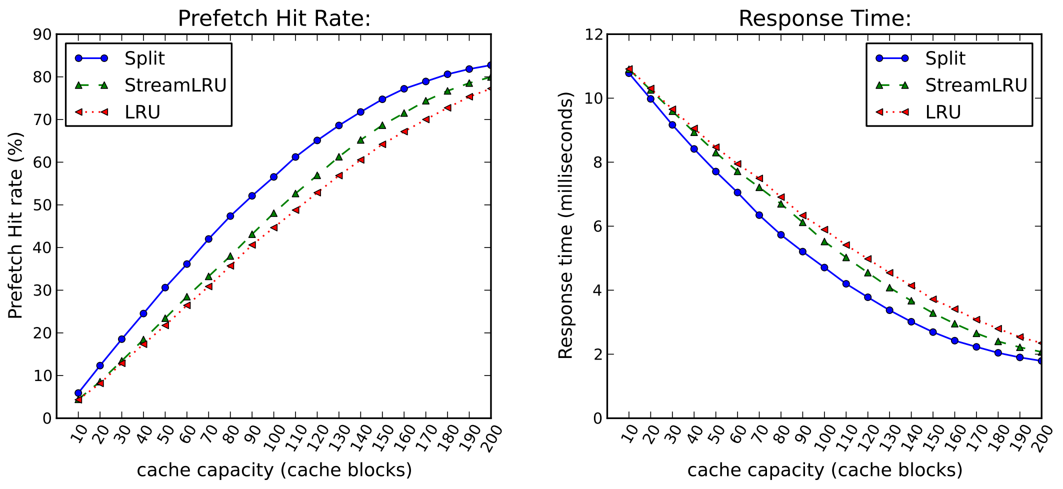

Figure 1: Workload with 90 multiple sequential sequences and 10 random sequences.

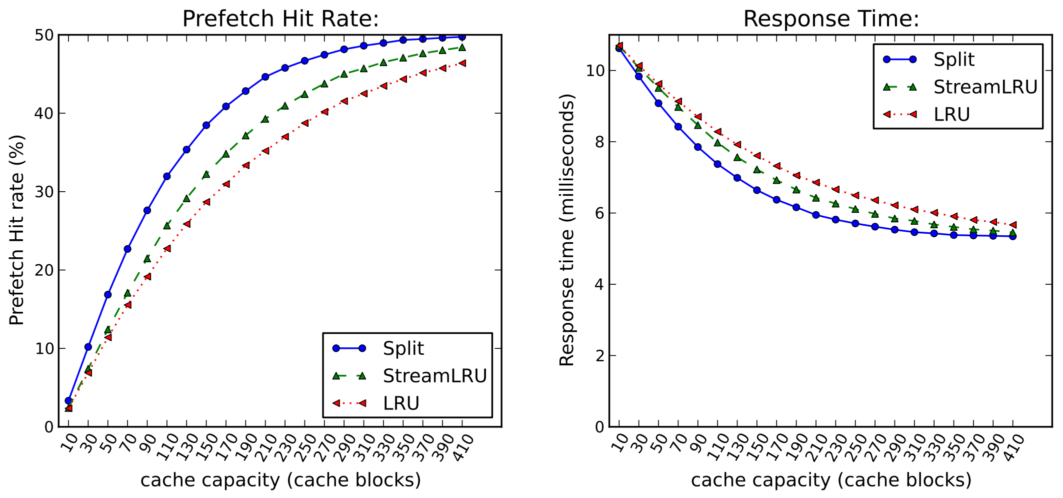

Figure 2: Workload with 50 single sequential and 50 random sequences.

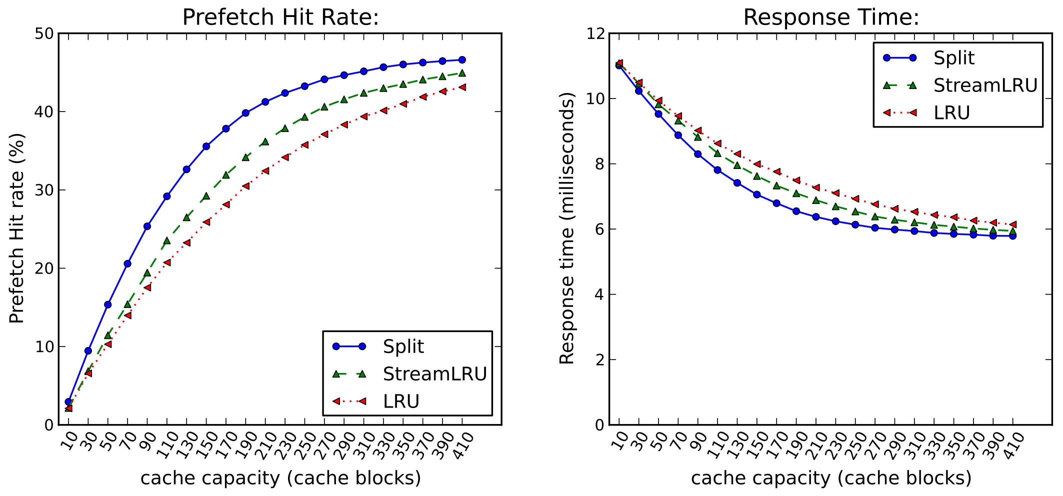

Figure 3: Workload with 10 single sequential, 40 multiple sequential, 50 random sequences.

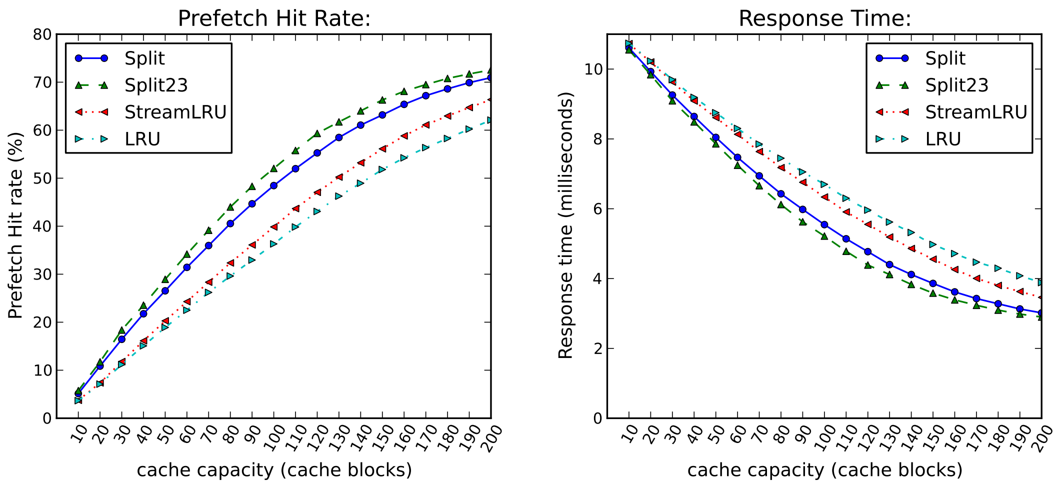

Figure 4: The Up queue is twice as long as the Down Queue in Split23; Workload with 80 multiple sequential and 20 random sequences.

Table 5: Storage simulator setup

Disksim parameter	Value
disk type	cheetah9LP
disk capacity	17783240 blocks
mean disk read seek time	5.4 msec
maximum disk read seek time	10.63 msec
disk revolutions per minute	10045 rpm

Workload: Sequential prefetching is effective only if the workload has some sequential locality. We follow the approach used in earlier papers [9, 10, 20] and generate a workload based on SPC2 specifications [2]. The cache workload generator is composed of sequence generators; based on SPC2 specifications, there are three types of sequence generators - single sequential, multiple sequential, and random. Example output from the three types of sequence generators:

single sequential: $<$ 234, 235, 236, 237, ... $>$; a single sequence of requests for contiguous blocks.

multiple sequential: $<$ 100, 101, 102, 103, ..., 4567, 4568, 4569, ..., 95489, 95490, 95491, ... $>$; two or more subsequences of requests for contiguous blocks; the number of requests in each subsequence is drawn from a Poisson distribution.

random sequence: $<$ 45, 1982, 99999, 247, 8174, ... $>$; a single sequence of requests for random blocks.

Each of the experiment's workload traces is composed of 100,000 requests from 100 independent, concurrent sequence generators; all generators start up at the beginning of the simulation. For each sequence, the request interarrival times are drawn from an exponential distribution. Therefore, the final workload submitted to the cache simulator consists of interleaved requests from the 100 sequence generators. An example of a workload containing the above 3 interleaved sequences is:

$<$ 100, 45, 234, 235, 101, 236, 102, 103, 104, 1982, 99999, 237, 238, 105, 4567, 247, 8174, 4568, ... $>$.

Note that from the perspective of PA or PM, this example workload contains several (more than 3) interleaved streams: $<$ 100, 101, 102,... $>$, $<$ 234, 235, ... $>$ $<$ 45 $>$, $<$ 1982 $>$, $<$ 99999 $>$, ... Each request from the random sequence is viewed by a prefetch technique as a start of a stream, and therefore, each random request results in a (wasted) prefetch. The last request in a subsequence of a multiple sequential stream also results in a wasted prefetch.

Performance metrics: The goal of this experimental evaluation is to verify that a Split queue improves the performance of a prefetch cache. The mean response time is the performance metric of relevance to end users. In a cache without prefetching, the higher the cache hit rate, the lower is the mean response time. In a cache with prefetching, this simple relationship between hit rate and response time need not hold due to varying intensity of disk traffic generated by each prefetch technique. For example, PM has a lower prefetch hit rate than PA, but PM piggybacks prefetch requests onto missed requests and generates less traffic at the disks which may result in a lower response time for PM. Therefore, the experiments measure both the prefetch hit rate and the mean response time. The prefetch hit rate is the ratio of the number of hits to the total number of user requests in the cache workload. The mean response time is the product of miss rate (1 - prefetch hit rate) and mean disk response time.

Experiments: The experiments evaluate the single queue with regards to both LRU and streamLRU since streamLRU is the version of LRU that recognizes prefetch blocks. We measure hit rate and response time for a fixed workload as the cache size varies. The cache size is increased until the maximum prefetch hit ratio for the workload is achieved.

In the first experiment shown in Figure 1, the workload consists of 90 interleaved multiple sequential sequences and 10 random sequences. From the prefetch technique's viewpoint, every random request is the start of a new stream, every start of a new subsequence in a multiple sequential sequence is a new stream. Therefore, there are more than 100 streams in the workload from the viewpoint of the prefetch technique. The cache size is varied from 10 to 200 blocks. When the cache size is 110 blocks, the hit rate of Split is approximately 30% greater than that of LRU and StreamLRU; resulting in a 25% decrease in response time of Split.

In the next experiment (Figure 2), the workload is generated by 50 single sequential sequence generators and 50 random sequence generators. Therefore, the maximum hit rate for this workload is 0.5. Since 2 blocks are prefetched into the cache for every random request, the cache contains a lot of useless blocks. The maximum hit rate is achieved when the cache approaches 400 blocks. The greater the randomness in the workload, the larger is the cache size required to achieve the maximum prefetch hit rate. In the last experiment - Figure 3 - the workload consists of 40 multiple sequential sequences, 10 sequential sequences, and 50 random sequences. Again, Split queue performs better than the single queue. When the cache size is 150 blocks, the hit rate of Split is 40% greater than that of LRU and SplitLRU. The Split queue achieves maximal performance for the workload before the single queue. Note that with a more sophisticated prefetch technique, one that prefetches more blocks from recognized streams, the performance improvements in Split could be greater.

Split23: By varying the relative sizes of the Up and Down queues, one can improve the performance of Split. When the size of the Up queue is larger than the Down queue, the first half of each stream remains longer in the cache at the cost of evicting the second half sooner. The result is a higher cache hit rate and a lower response time (Figure 4). A long Up queue may be undesirable for workloads with higher request rates because the efficiency of prefetching multiple blocks is lost. On the other hand, a long Up queue may benefit workloads with a large number of intermittent streams. The Split Up and Down queues can be adjusted to the sequentiality of the workload. If the number of hits in the Down queue is greater than the number of hits in the Up queue, it indicates that streams may be getting evicted too soon. This can be addressed by either increasing the size of the Up queue, or by reducing the degree of prefetch.

Summary: The Split queue performs better than the single queue since it incorporates both the temporal and spatial locality of streams. By evicting blocks from the least recently used stream, StreamLRU incorporates the temporal locality of streams. For the workloads considered, StreamLRU performs better than LRU, but their performances are close, almost statistically identical. This is somewhat surprising given that LRU does not recognize streams.

6 Prior papers

The sequential access pattern is common in file/storage cache workloads. Consequently, a large number of sequential prefetch techniques exist for file and storage caches [5, 8, 11, 20, 22, 28, 31]. These papers differ in when they initiate prefetch, and in how many blocks they prefetch. Prefetching techniques that try to identify other access patterns in the workload have also been developed [1, 12, 21, 27]. However, commercial storage systems rarely implement these complex prefetch techniques since they are computationally expensive and may actually slow down the system [10].

The replacement policy is critical to the performance of caches, and so there are a large number of papers on this topic. Several replacement policies have been developed, and the policies can be classified into 4 types, namely, FIFO (LIFO): based on time of insertion into the cache; LRU (MRU): based on time of last access (recency); LFU: based on frequency of access; and LRU-2 [24, 25], 2Q [15], LIRS [14], LRFU [19], MQ [32], ARC [23]: based on both recency and frequency of access. An underlying theme to all these policies, with the exception of FIFO, is that they are designed for data blocks with temporal locality. These policies are not necessarily optimal for prefetch blocks with spatial locality.

There are very few replacement policies that consider the impact of prefetch blocks. The SARC technique uses StreamLRU [10] as the replacement policy for the prefetch cache. SARC focuses on prefetch cache sizing, and compares the number of hits in the prefetch cache and the reference cache to determine whether to eject a block from the prefetch list or the reference list. Since Split performs better than Stream, replacing the StreamLRU policy by Split would improve the performance of SARC. DULO [13] is a replacement policy that accounts for both temporal and sequential locality; DULO gives priority to random blocks over sequential blocks since the cost of loading several sequential blocks is less than the cost of loading random disk blocks. Consequently, DULO increases temporal hit rate at the cost of spatial hit rate.

Several papers analyze integrated prefetch and caching algorithms. These algorithms normally have prior knowledge of the workload [3, 7, 16, 17, 18, 29, 30].

The prevalent approach to improving the performance of a prefetch cache is to develop a new prefetch technique. Prior research has largely ignored the dependency between the prefetch technique and the replacement policy. Replacement policies are normally evaluated in caches without prefetching; prefetch techniques are evaluated with a fixed replacement policy, usually LRU. A recent paper has demonstrated via simulations that the relative performances of replacement policies are unpredictable when combined with prefetching [6]. This paper is the first to identify prefetching and caching as a synchronization problem. By viewing the problem from a different light, this paper shows that the performance of prefetching and caching can be improved without modifying the basic algorithms. In fact, by framing prefetch caching as a synchronization problem, Split integrates the actions of standalone prefetch and replacement algorithms.

7 Discussion

Researchers have a good comprehension of prefetch techniques and replacement policies, in isolation. However, we are a long way from understanding the combined impact of the two algorithms. Relating to classical synchronization problems such as reader-writer: it is not the individual reader algorithm and writer algorithm that is fascinating, it is the interaction between the two algorithms that makes this problem interesting and challenging. Since prefetching and caching is inherent to today's computers, an understanding of the interaction between the two key algorithms could lead to performance improvements without investing in new hardware/software as demonstrated by Split. The Split queue is a single queue with the 2nd prefetched block being inserted midway in the replacement queue. It is simple, but simplicity requires understanding.

It is not really surprising that prefetching and caching has not been cast as a synchronization problem. In classical synchronization problems, the individual algorithm is not very complex or interesting by itself; for example, the reader and writer algorithms are pretty straightforward. In the prefetch cache problem, however, the prefetch technique and the replacement policy are both research problems, each in its own right. Each algorithm has to deal with several parameters, and seemingly arbitrary performance changes occur by tweaking any parameter [6]. Some of these parameters/issues are: the cache size, the workload, identification of workload access patterns, the number of blocks to prefetch, when to prefetch, when to evict, and what to evict. Since there are so many parameters, it is difficult to see anything, let alone understand what and why. Belady's anomaly [4], a rare occurrence in caches without prefetching, is common in caches with prefetching. By focusing on the replacement queue as the software entity that captures the impact of all the parameters involved in prefetching and caching, it may be possible to understand and develop better systems.

8 Conclusion

The impact of prefetching and caching is encapsulated in the cache replacement queue. This paper shows that by splitting the queue, the performance of a prefetch cache improves, without changing the prefetch technique or the replacement policy. The contributions of this paper are the following:

1. formalization and analysis of StreamLRU;
2. development and analysis of Split;
3. proof that the SplitLRU cache holds more streams than either the LRU cache or the StreamLRU cache;
4. proof that the hit rate of SplitLRU is an upper bound for the hit rate of StreamLRU;
5. evidence via simulation that SplitLRU results in lower response time and higher hit rate than LRU and StreamLRU.

In most of the experiments, the lengths of the Up and Down queue are equal. In future work, we plan to show that adjusting the length of Up queue and using a more sophisticated prefetch technique that varies the amount of prefetch results in greater improvement of performance. We plan to evaluate Split queue with other replacement policies and prefetch techniques. The Split policy is evaluated here for single-level caches. However, the Split policy with its 2-queue structure is naturally geared for multiple-level cooperative caches and it would be interesting to analyze Split in a multiple-level setting. Another interesting problem is prefetch cache sizing based on hit rates in Up and Down queues.

While this paper is focused on the Split queue, the real contribution of the paper is the identification of the replacement queue as a significant player in the performance of prefetching and caching. While it is a well known fact that the performances of prefetch techniques and replacement policies are dependent on each other, it is not known how to evaluate this dependency. This is the first paper to establish that the replacement queue could be used to understand the dependency between the two key algorithms behind prefetching and caching.

References

[1] ZFS performance. online, 2007. `http://www.solarisinternals.com/wiki/index.php/ZFS_Performance`.

[2] SPC Benchmark-2(SPC-2) Official Specification, version 1.2.1. Tech. rep., Storage Performance Council, Effective 27 Sept. 2006. `http://www.storageperformance.org/specs`.

[3] ALBERS, S., AND BTTNER, M. Integrated prefetching and caching in single and parallel disk systems. *Inf. Comput.* (2005), 24–39.

[4] BELADY, L. A. A study of replacement algorithms for virtual-storage computer. *IBM Systems Journal* (1966), 78–101.

[5] BHATIA, S., VARKI, E., AND MERCHANT, A. Sequential prefetch cache sizing for maximal hit rate. In *18th Annual IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (2010), pp. 89 – 98.

[6] BUTT, A. R., GNIADY, C., AND HU, Y. C. The performance impact of kernel prefetching on buffer cache replacement algorithms. *IEEE Transactions on Computers 56*, 7 (2007), 889–908.

[7] CAO, P., FELTEN, E. W., KARLIN, A. R., AND LI, K. Implementation and performance of integrated application-controlled caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems 14*, 4 (1996), 311–343.

[8] GILL, B. S., AND BATHEN, L. A. D. Optimal multistream sequential prefetching in a shared cache. *ACM Transactions on Storage (TOS) 3* (2007).

[9] GILL, B. S., AND BATHEN, L. A. D. AMP: Adaptive multi-stream prefetching in a shared cache. In *Proc. of USENIX 2007 Annual Technical Conference* (Feb 2007), 5th USENIX Conference on File and Storage Technologies.

[10] GILL, B. S., AND MODHA, D. S. SARC: Sequential prefetching in adaptive replacement cache. In *Proc. of USENIX 2005 Annual Technical Conference* (2005), pp. 293–308.

[11] GINDELE, J. D. Buffer block prefetching method. *IBM Tech Disclosure Bull. 20*, 2 (July 1977), 696 – 697.

[12] GRIFFIOEN, J., AND APPLETON, R. Reducing file system latency using a predictive approach. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference* (1994), vol. 1, USENIX Association Berkeley, CA, USA, pp. 197–208.

[13] JIANG, S. Dulo: An effective buffer cache management scheme to exploit both temporal and spatial localities. In *In USENIX Conference on File and Storage Technologies (FAST* (2005).

[14] JIANG, S., AND ZHANG, X. Lirs: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (New York, NY, USA, 2002), SIGMETRICS '02, ACM, pp. 31–42.

[15] JOHNSON, T., AND SHASHA, D. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the International conference on very large databases* (1994).

[16] KALLAHALLA, M., AND VARMAN, P. J. Pc-opt: Optimal offline prefetching and caching for parallel i/o systems. *IEEE Trans. Computers* (2002), 1333–1344.

[17] KIM, J. M., CHOI, J., KIM, J., NOH, S. H., MIN, S. L., CHO, Y., AND KIM, C.-S. A low-overhead, high-performance unified buffer management scheme that exploits sequential and looping references. In *OSDI'00* (2000), pp. 119–134.

[18] KIMBREL, T., CAO, P., FELTEN, E. W., KARLIN, A. R., AND LI, K. Integrating parallel prefetching and caching. In *SIGMETRICS'96* (1996), pp. 262–263.

[19] LEE, D., CHOI, J., KIM, J., S.H. NOH, S. M., CHO, Y., AND KIM, C. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers* (December 2001), 1352–1361.

[20] LI, M., VARKI, E., BHATIA, S., AND MERCHANT, A. TaP: Table-based prefetching for storage caches. In *6th USENIX Conference on File and Storage Technologies (FAST '08)* (2008), pp. 81–97.

[21] LI, Z., CHEN, Z., SRINIVASAN, S. M., AND ZHOU, Y. C-miner: Mining block correlations in storage systems. In *Proceedings of the 3th USENIX Conference on File and Storage Technologies (FAST)* (2004), pp. 173–186.

[22] LIANG, S., JIANG, S., AND ZHANG, X. STEP: Sequentiality and thrashing detection based prefetching to improve performance of networked storage servers. In *Distributed Computing Systems, 2007. ICDCS '07. 27th International Conference on* (2007), pp. 64–.

[23] MEGIDDO, N., AND MODHA, D. Outperforming lru with an adaptive replacement cache algorithm. *Computer 37*, 4 (2004), 58–65.

[24] O'NEIL, E. J., O'NEIL, P. E., AND WEIKUM, G. The lru-k page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1993), SIGMOD '93, ACM, pp. 297–306.

[25] O'NEIL, E. J., O'NEIL, P. E., AND WEIKUM, G. An optimality proof of the lru-k page replacement algorithm. *J. ACM 46* (January 1999), 92–112.

[26] SILBERSCHATZ, A., GALVIN, P. B., AND GAGNE, G. *Operating System Concepts*, 8th ed. Wiley Publishing, 2008.

[27] SIVATHANU, M., PRABHAKARAN, V., POPOVICI, F. I., DENEHY, T. E., AIPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Semantically-smart disk systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (2003).

[28] SMITH, A. J. Cache memories. *ACM Computing Surveys 14*, 3 (1982), 473–530.

[29] TEMAM, O. An algorithm for optimally exploiting spatial and temporal locality in upper memory levels. *IEEE Trans. Computers* (1999), 150–158.

[30] TOMKINS, A., PATTERSON, R. H., AND GIBSON, G. A. Informed multi-process prefetching and caching. In *SIGMETRICS'97* (1997), pp. 100–114.

[31] VANDERWIEL, S. P., AND LILJA, D. J. Data prefetch mechanisms. *ACM Computer Survey 32*, 2 (2000), 174–199.

[32] ZHOU, Y., PHILBIN, J. F., AND LI, K. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the USENIX Annual Technical Conference* (2001).