Improve Prefetch Performance by Splitting the Cache Replacement Queue

Chapter · January 2013

DOI: 10.1007/978-3-642-38227-7_13

3 authors, including:

Elizabeth Varki
University of New Hampshire
40 PUBLICATIONS **410** CITATIONS
SEE PROFILE

Arif Merchant
Google Inc.
118 PUBLICATIONS **4,449** CITATIONS
SEE PROFILE

Improve Prefetch Performance by Splitting the Cache Replacement Queue

Elizabeth Varki,
Allen Hubbe *, and
Arif Merchant **

University of New Hampshire,
EMC,
Google
varki@cs.unh.edu
Allen.Hubbe@emc.com
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. This paper shows that even with a simple prefetch technique, there is an increase in hit rate when the LRU replacement queue is split into two equal sized queues. A more significant performance improvement is possible with a sophisticated prefetch technique and by splitting the queue unequally.*

Keywords: prefetching, caching, replacement policies, disk array, RAID

1 Introduction

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. For example, if a file is being read sequentially, the prefetch technique associated with the file system cache may prefetch several blocks contiguous to the requested file block. Thus, a prefetch technique is responsible for leveraging the *spatial* locality of reference in the cache workload.

The task of a prefetch technique is to determine what data blocks to prefetch and when to prefetch the blocks. The prefetch technique, however, does not control when a prefetched block is evicted from the cache. 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. Therefore, it is the replacement policy that is responsible for keeping a prefetch block in the cache until it is requested.

The goal of cache software is to ensure that the cache contains blocks that will be requested in the near-future. The cache software essentially consists of two

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

algorithms, namely, the prefetch technique and the replacement policy. The two algorithms 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.

Both the prefetch technique and the replacement policy have a say in the contents of the cache via the replacement queue. The cache replacement queue orders cache blocks by eviction priority and it is the meta-data used by caching software. Even though the prefetch technique and the replacement policy are distinct standalone algorithms, they share this data structure. 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. The cache replacement queue encapsulates the combined impact of the two algorithms, and ultimately determines the performance of the prefetch cache.

Contribution: This paper shows that the performance of a prefetch cache can be improved by merely splitting the single replacement queue into two queues. The replacement policy should be aware of the two queues and of the prefetched blocks. This paper demonstrates the Split queue approach using the sequential prefetch technique and the LRU replacement policy.

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: We explain Split using an example workload that displays sequential locality:

$<$ 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 [5]: *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.

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, 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. The * represents a prefetch hit.

Now, consider a second technique that prefetches 2 blocks on miss, and prefetches 2 blocks on hit of 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* $>$.

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 [3]. 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 Split

This is a how-to section that explains the mechanics of Split queue with respect to sequential prefetch and LRU. The next section presents experimental evidence of Split's superior performance when the workload displays sequential locality. Section 5 presents the intuition behind Split and explains why the Split queue is better than the single queue.

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 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 $1c$

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	

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.

4 Experimental Evaluation

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.

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. The workload used in our experiments contains no re-references to ensure that all hits are prefetch block hits. Our cache workload generator is composed of sequence generators; we have 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 our experiment's workload traces is composed of a total 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

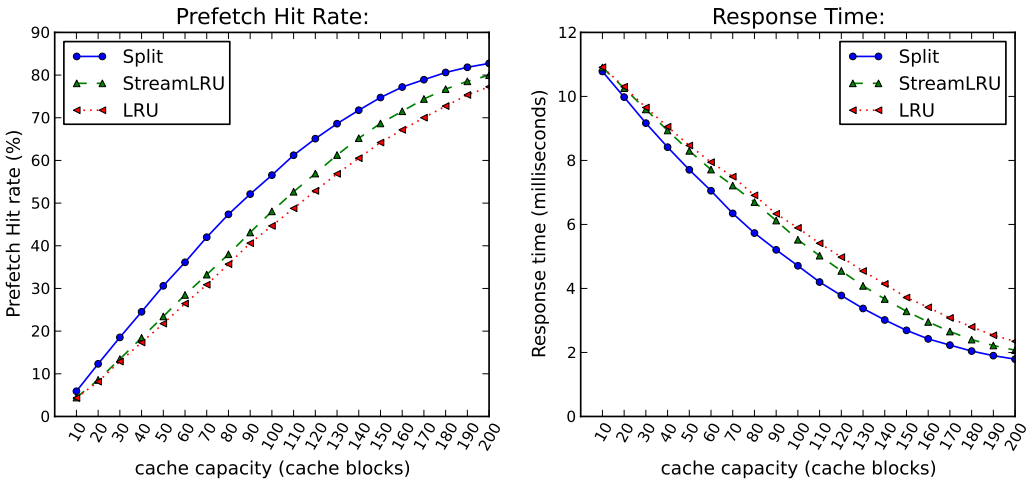

Fig. 1. Workload with 90 multiple sequential sequences and 10 random sequences.

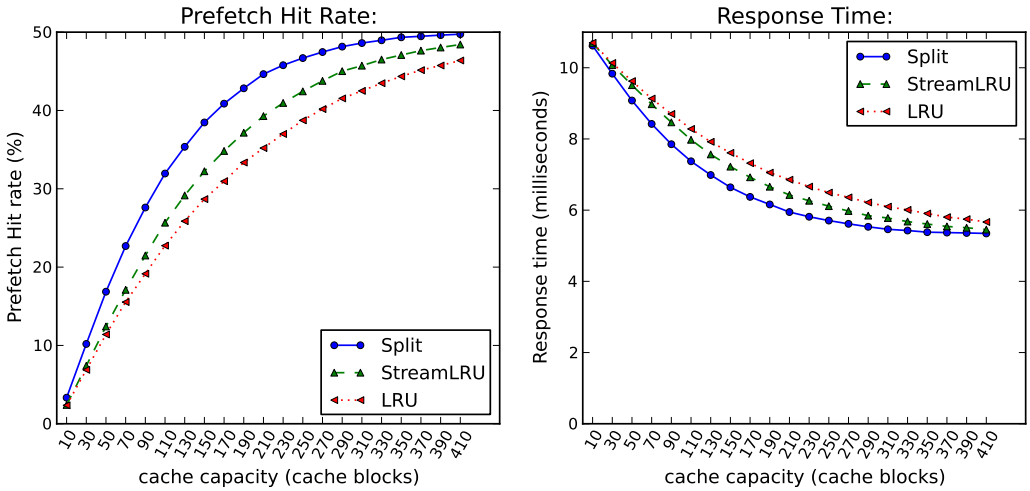

Fig. 2. Workload with 50 single sequential and 50 random sequences.

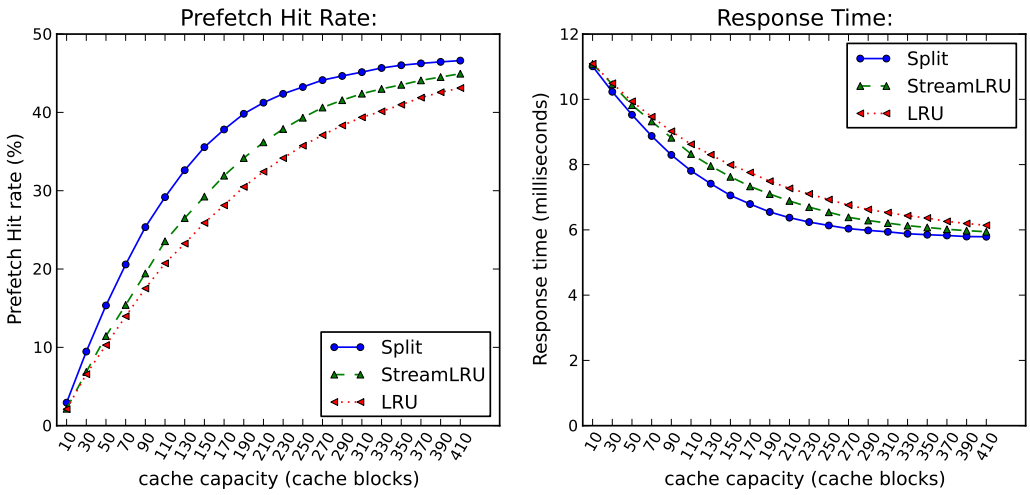

Fig. 3. Workload with 10 single sequential, 40 multiple sequential, 50 random se-

interarrival times are drawn from an exponential distribution. Therefore, the final workload submitted to our cache simulator are 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 $>$, $<$ 9999 $>$, ... 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, our 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: In our experiments, we 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. The workload has no rereferences of blocks, so a prefetch block is evicted from the cache as soon as it gets a hit (similar to the examples presented in the last section).

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 27% greater than that of LRU and StreamLRU; resulting in a 23% 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 38% greater than that of LRU and SplitLRU. The Split queue achieves maximal performance for the workload before the single queue.

5 Analysis

In all our experiments, the Split queue performs better than the single queue. In order to understand why the Split queue performs better than the single queue, it is necessary to explain sequential (spatial) locality in relation to the actions of a prefetch technique. In order to amortize the cost of prefetching - reduce the traffic at the disks - several blocks are prefetched 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. Therefore, when there are 2 prefetched blocks, as in our examples and experiments, the first block is expected to receive an user request before the second block. Split incorporates this characteristic by inserting the 1st block in the Up queue and the 2nd block in the Down queue. Even if the 2nd block gets evicted, the first block remains in the cache longer than it would in the single queue approach; if the first block gets a prefetch hit, there is time to prefetch the 2nd block.

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. A possible reason for the similarity between LRU and StreamLRU's performance may be found by looking at the replacement queues in Tables 1 and 2 (Section 2): when user request for 3 is processed, LRU's replacement queue contains streams 3, 1, 2, while StreamLRU's replacement queue only contains 3 and 1. Since LRU does not keep stream blocks together, it is possible for LRU's replacement queue to hold more streams than StreamLRU's replacement queue. In general, the Split queue holds more streams than the single queue and consequently, Split performs better.

Summarizing, this paper shows that given a prefetch technique and LRU, performance is improved by simply splitting the replacement queue. Prior work has shown that the performance of a prefetch technique cannot be studied in isolation of the replacement policy and vice versa [1] [2] [4]. The intuition behind Split is the recognition that this dependency is a result of the shared data structure - the cache replacement queue - that both algorithms update. Instead of targeting the prefetch technique or the replacement policy, Split manipulates

the queue to maximize performance. In fact, the Split queue is a single queue with the 2nd prefetched block being inserted midway in the replacement queue.

6 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. In the experiments, the lengths of the Up and Down queue are equal. By setting the Up queue to twice the length of the Down queue, we have found that there is a greater improvement of Split's performance. This paper analyzes the Split queue with a basic prefetch technique that prefetches at most 2 blocks. The Split queue improves performance more significantly if more blocks are prefetched per stream. Incorporating a combination of the above - increasing length of Up queue and using a more sophisticated prefetch technique that varies the amount of prefetch may result in greater improvement of performance.

We plan to evaluate the Split queue approach with other replacement policies. 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. Other issues that could be addressed include analysis of traffic generation by the policies, theoretical analysis of Split, and prefetch cache sizing based on hit rates in Up and Down queues.

References

1. 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.
2. 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.
3. 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.
4. 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).
5. 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.