The Split replacement policy for caches with prefetch blocks

Abstract

Prefetching is an inbuilt feature of file system and storage caches. The cache replacement policy plays a key role in the performance of prefetching techniques, since a miss occurs if a prefetch block is evicted before the arrival of the on-demand user request for the block. Prefetch blocks display spatial locality, but existing cache replacement policies are designed for blocks that display temporal locality. This paper develops a new replacement policy called *Split* for caches that implement prefetching. The relative performances of Split and other replacement policies, such as LRU, are evaluated using simulation and theoretical analysis. The evaluation shows that the Split replacement policy gets a higher hit rate and a lower response time than any of the other compared policies, at the cost of slightly more disk accesses.

1 Introduction

The memory hierarchy model is integral to computer systems, and the success of this model hinges on the performance of caches. When a user requests data, the data have to be uploaded to the user's level. The speed of the upload depends on the distance (*i.e.*, levels) separating the data-user from the data-store. The closer the data is to the user, the faster the upload. In order to facilitate fast upload, caches are placed at each level of the memory hierarchy, and data predicted to be needed in the near future are kept in the caches.

The request predictions are based on the adage, history repeats itself, and result in two types of blocks being stored in a cache. The first type, known as *reference* blocks are data blocks that recently received on-demand requests. They are stored in the cache on the assumption of temporal locality - blocks may be referenced multiple times during a time period. The second type, known as *prefetch* blocks are data blocks that have not received on-demand requests. These blocks are prefetched and cached before they are requested on the assumption of spatial locality - data access patterns relate to the logical/physical placement ordering of the blocks. A prefetch block is a future block that relates to a recent request access sequence. For example, if the first few blocks of a file are read sequentially, then contiguous blocks could be prefetched on the assumption that the sequential access of this file will continue.

When a cache is full and a new block is to be inserted, the cache replacement policy determines which cached block to eject to make space for the new block. 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 [22, 23], 2Q [15], LIRS [13], LRFU [16], MQ [30], ARC [21]: 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.

The majority of prior research in prefetching has focused on the prefetching technique itself, namely, what to prefetch, when to prefetch and how much to prefetch [8, 11, 18, 19, 26, 27, 28]. Surprisingly, there are no papers on replacement policies for blocks with spatial locality. It has been recommended that a prefetch block be evicted immediately on a hit since prefetch blocks have low temporal locality [24]. Several papers have analyzed the optimum size of a prefetch cache [4, 10, 12, 17, 18, 25]. Most of these papers determine the optimum size based on the location of hits in the prefetch cache [4, 12, 17, 18]. SARC [10] is also about prefetch cache sizing, but unlike previous papers, this paper 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. A couple of papers [10, 29] have mentioned, in passing, that a prefetch access sequence should be considered as a single unit by a replacement policy. However, the replacement policy was not the focus of these papers, so they did not delve further.

Contributions: This paper develops a replacement policy called **Split**LRU, for caches with prefetch blocks. As the name suggests, the replacement policy is a combination of Split and LRU. The LRU policy determines which prefetch sequence to evict, while the Split policy determines which blocks, if any, to evict from the sequence chosen for eviction. Thus, the SplitLRU replacement policy addresses both the temporal locality displayed by sequences and the spatial locality displayed by prefetch blocks of the sequence.

The Split policy can be combined with any of the temporal replacement policies mentioned above. In this paper, we limit our analysis to the combination of Split and LRU since most replacement policies are variants of LRU. The Split approach has the computational complexity of its partner, so SplitLRU has LRU's constant complexity. This paper analyzes Split with respect to sequential prefetching since this type of prefetching is widely used in file system and storage device caches.

The contributions of this paper are the following:

1. proof that LRU ejects prefetch blocks (and sequences) in FIFO order;

2. formalization and analysis of the replacement policy that treats prefetch blocks from a sequence as a single stream unit [10, 29]. The paper combines this approach with LRU; we name the ensuing policy *StreamLRU*.

3. development of the Split replacement policy for caches with prefetch blocks;

4. proof that the SplitLRU cache holds more sequences than either the LRU cache or the StreamLRU cache;

5. proof that the hit rate of SplitLRU is an upper bound for the hit rate of StreamLRU;

6. evidence via simulation that SplitLRU results in lower response time and higher hit rate than LRU and StreamLRU.

A sidebar contribution of this paper is a demonstration of the complexity of comparing two replacement policies when the cache implements prefetching. The paper shows that a comparison between two prefetch cache replacement policies is valid only if the prefetch technique does not depend on the replacement policy. Without this condition, performance depends on both the replacement policy and the prefetch technique.

While this paper names and evaluates the StreamLRU policy, the policy has been used in prior papers [10, 29]. In fact, SARC uses StreamLRU policy to determine which block to evict, and so SARC reduces to the StreamLRU policy if the reference list is removed from consideration.

In the rest of the paper, StreamLRU and SplitLRU are sometimes written simply as Stream and Split. The next three sections present the LRU, Stream and Split policies. Section 5 presents a theoretical comparison of the three replacement policies. In Section 6, the performances of the replacement policies are compared via simulation.

2 Sequential prefetch

Sequential prefetching is the most widely used prefetching technique. Here, we describe a version of sequential prefetching implemented in file system caches. For uniformity, the unit used in the paper is blocks - a cache size is C blocks, a read request is for x blocks, etc.

Read-ahead (*i.e.,* prefetch) is performed by most file system caches. When a read request misses in the file system cache, a *synchronous* prefetch request for one or more blocks contiguous to the missed blocks is issued. For example, if an on-demand request for block 1 misses in the cache, a read request for blocks $< 1, 1a, 1b, 1c >$ may be issued, where blocks 1a, 1b and 1c are contiguous to block 1. The *prefetch degree*, which is the number of blocks prefetched, is 3. The prefetch degree may be fixed or varying, depending on the prefetch technique. Typically, a prefetch is not initiated on a hit of every prefetched block. Instead, one of the prefetched blocks is marked as a trigger block, and a prefetch is issued only if the trigger block gets a hit. Referring back to the above example, suppose block $1b$ is the trigger. When an on-demand request for block $1a$ arrives, it receives a hit, but prefetch is not initiated. When an on-demand request for block $1b$ arrives, it receives a hit, and an *asynchronous* prefetch for blocks $< 1d, 1e, 1f, 1g >$ may be issued. The trigger is reset to block $1f$. In this example, the prefetch degree varies since 3 blocks are prefetched on miss, 0 blocks are prefetched on hit of non-trigger block, and 4 blocks are prefetched on hit of trigger block.

Definition 1 *An access sequence is an ordered series of contiguous block addresses in increasing order.*

In the above example, the access sequence is $< 1, 1a, 1b, 1c, 1d, 1e, 1f, 1g >$. The I/O workload submitted to a cache consists of interleaved requests from various accesses, such as,

$< 1, 5, 1a, 2, 2a, 6, 9, 5a, 2b, 1b, ... >$. On-demand requests for blocks from a sequence are interleaved amongst blocks from other accesses. From a sequential prefetch technique's viewpoint, the I/O workload presented in the above example, potentially, has the following access sequences: $< 1, 1a, 1b, 1c, .. >$, $< 5, 5a, 5b, 5c, ... >$, $< 2, 2a, 2b, 2c, ... >$, $< 6, 6a, 6b, 6c, ... >$ and $< 9, 9a, 9b, 9c, ... >$.

Prefetching is all about access sequences - the prefetching technique implicitly or explicitly identifies a sequential access stream and prefetches contiguous blocks on the assumption that the sequential access will continue. The prefetched blocks are inserted into the cache along with the missed blocks. The cache contains both reference blocks and prefetch blocks. When an on-demand request arrives for a prefetch block, it gets a prefetch hit; the hit block is now a reference block.

Definition 2 *A prefetch hit is the first hit of a prefetch block; once the block has been referenced, any further hits to the block are counted as reference hits.*

3 LRU vs. StreamLRU

Prefetching techniques make the following two assumptions about the workload:

Assumption 1 *Each prefetch block is associated with an access sequence.*

Assumption 2 *The later blocks of a sequence are likely to have a later first access time than the blocks that precede them.*

Standard replacement policies like LRU, FIFO and LFU ignore access sequences. The LRU policy evicts the least recently used block, and does not distinguish between reference blocks and prefetch blocks. Since a prefetch block, by definition, has never been accessed, the last access time of a prefetch block is the time it was prefetched and inserted into the MRU end of the replacement queue. When a prefetch block gets a hit, the LRU policy moves the referenced block to the MRU end of the replacement queue, but leaves the rest of the prefetched blocks from the hit block's sequence in their replacement queue position. Referring to the example in Section 2, when block $1a$ receives a hit, the LRU policy moves $1a$ to the MRU end, but blocks $1b, 1c$ retain their position in the replacement queue. A prefetched block reaches the eviction end when its access time (*i.e.*, prefetch time) is the oldest. Thus, if an evicted block is a prefetch block, it is guaranteed that it is the oldest prefetch block in the cache.

Result 1 *The LRU replacement policy ejects prefetch blocks in FIFO order.*

Table 1: LRU vs. StreamLRU - LRU better: Prefetch technique ensures that the next 2 contiguous blocks related to each on-demand block is in the cache.

Prefetch cache size = 4, LRU							
hits			h1		h2		h3
workload	**1**	**2**	**1a**	**3**	**2a**	**4**	**2b**
Cache	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**	

Prefetch cache size = 4, StreamLRU							
hits			h1				h2
workload	**1**	**2**	**1a**	**3**	**2a**	**4**	**2b**
Cache	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 2: LRU vs. StreamLRU - LRU worse: Prefetch technique prefetches 2 blocks on miss; prefetches 2 blocks on hit if the block is the last block in the sequence. Same workload as Table 1

Prefetch cache size = 4, LRU							
hits			h1		h2		
workload	**1**	**2**	**1a**	**3**	**2a**	**4**	**2b**
Cache	1a	2a	2a	3a	3a	4a	2c
	1b	2b	2b	3b	3b	4b	2d
		1a	1b	2a	2b	3a	4a
		1b		2b		3b	4b
Eject				**1b**		**2b**	

Prefetch cache size = 4, StreamLRU							
hits			h1		h2		h3
workload	**1**	**2**	**1a**	**3**	**2a**	**4**	**2b**
Cache	1a	2a	1b	3a	2b	4a	2c
	1b	2b	2a	3b	2c	4b	4a
		1a	2b	1b	3a	2b	4b
		1b		2a	3b	2c	
Eject				**2b**	**1b**	**3a**	
						3b	

Table 3: LRU vs. StreamLRU - equal hits: Prefetch technique prefetches 1 block on hit or miss. Same workload as previous tables.

Prefetch cache size = 2, LRU							
hits			h1				h2
workload	**1**	**2**	**1a**	**3**	**2a**	**4**	**2b**
Cache	1a	2a	1b	3a	2b	4a	2c
		1a	2a	1b	3a	2b	4a
Eject				**2a**	**1b**	**3a**	

Prefetch cache size = 2, StreamLRU							
hits			h1				h2
workload	**1**	**2**	**1a**	**3**	**2a**	**4**	**2b**
Cache	1a	2a	1b	3a	2b	4a	2c
		1a	2a	1b	3a	2b	4a
Eject				**2a**	**1b**	**3a**	

For clarity of exposition, in the rest of this paper, we assume that the cache is partitioned into a prefetch and reference cache. Upon a hit in the prefetch cache, the hit block is ejected from the prefetch list and may be inserted into the reference list.

The standard replacement policies can be modified to incorporate Assumptions 1 and 2 by doing the following simple step: on a hit, the hit block and all blocks following it in the sequence are moved in the replacement queue. The blocks preceding the referenced block are not moved on a hit, since sequential prefetching anticipates requests to arrive for following blocks, not preceding blocks. Note that this condition may be relaxed so that all blocks in the current prefetch window (*i.e.,* the set of blocks in an access sequence that must be kept in cache [5]) are moved upon a hit. In this paper, we limit our analysis to the LRU policy, and the modified scheme is called StreamLRU. The policy evicts blocks from the Least Recently Used sequence.

Prefetch sequences display temporal locality, since recently accessed sequences generally have a higher probability of getting future requests than sequences with no recent accesses. The Stream policy captures this behavior by moving all blocks in the hit sequence to the MRU end of the replacement queue. The LRU policy ignores access sequences in the workload and just moves the hit block to the MRU end. As a result, the Stream policy performs better than the LRU policy for workloads with access sequences that display temporal locality. It is possible to find workloads more suitable for LRU, and Table 1 presents such a scenario where Stream gets less hits than LRU.

Another factor that must be taken into consideration when evaluating the performances of LRU and Stream is the prefetching technique. The interaction between the replacement policy and the prefetching technique is complex and barely charted [7]. For the same workload, one prefetching technique may result in LRU getting more hits than Stream while another prefetching technique may reverse this result. This scenario is demonstrated in Tables 1, 2 and 3. Using the same workload, LRU gets more hits than Stream in Table 1, LRU gets less hits than Stream in Table 2, and LRU gets the same number of hits as Stream in Table 3.

The tables demonstrate that the relative performances of replacement policies vary with the prefetch technique. It is easy to demonstrate this dependence with other replacement policies too. While it is easy to come up with any number of examples that demonstrate the interaction between the replacement policy and the prefetch technique, it is hard to quantify the dependence and its impact on performance. The next section presents the Split policy and more examples that demonstrate the inconsistent performance of replacement policies when caches implement prefetching.

4 SplitLRU vs. LRU & StreamLRU

Split is a new replacement approach along the lines of StreamLRU - it is an improved version of Stream. Stream stores sequences as a unit, so this scheme gives equal priority to all blocks within a sequence. An assumption made by prefetching techniques is that an earlier block of a sequence has a higher probability of getting accessed before a later block of the sequence (reference Section 3). Split incorporates this assumption by evicting the later blocks of all the cached sequences before it starts evicting the earlier blocks of these sequences. Thus, unlike Stream, Split integrates the spatial locality of prefetch blocks. We explain the policy below.

As mentioned in Section 2, a common prefetching approach is to prefetch several blocks from a sequence at a time. One of these blocks is marked the trigger block. When a trigger block gets a hit, the prefetching technique jumps into action and prefetches several future blocks from the sequence. The trigger blocks demarcates each sequence into 2 parts, the *prefix* and the *suffix*; the trigger block can be included in either the prefix or the suffix, but not both. If the trigger is in the suffix, then the prefix of a sequence is from the start to the block preceding the trigger; the suffix is from the trigger block to the end of the sequence. For prefetch techniques without a trigger, an arbitrary block in the sequence can be chosen as the point of demarcation of a sequence into the prefix and suffix. If a sequence has only one block loaded in the cache, then the sequence just has a prefix.

Stream evicts all the blocks of the least recently used sequence before it starts evicting blocks from the next sequence in the LRU list. Split, on the other hand, evicts

the suffix of the least recently used sequence and then moves to the suffix of the next sequence in the LRU list. (Note that when a trigger is evicted, the trigger moves to the preceding block in the sequence.) The prefix of a sequence gets ejected only when the suffixes of all sequences in the cache are ejected. In Split, sequences stay longer in the cache than they would in the Stream scheme.

Table 4: Demonstrating Split's performance, using the same workload and prefetch technique presented in Table 1: 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**
Up Q	1a	2a	1b	3a	2b	4a	2c
		1a	2a	1b	3a	2b	4a
Down Q	1b	2b	1c	3b	2c	4b	2d
		1b	2b	2a	1b	3a	4b
Eject				**1c**	**3b**	**2c**	**3a**
				2b		**1b**	

Table 5: Demonstrating Split's performance, using the same workload and prefetch technique presented in Table 2: 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**
Up Q	1a	2a	1b	3a	2b	4a	2c
		1a	2a	1b	3a	2b	4a
Down Q	1b	2b	2b	3b	2c	4b	2d
		1b		2a	1b	3a	4b
Eject				**2b**	**3b**	**2c**	**3a**
						1b	

We demonstrate the mechanics of Split in Tables 4 and 5. These examples correspond to Tables 1 and 2 that compare the performances of LRU versus StreamLRU. In Split, the replacement queue is divided into two queues, the Up queue and the Down queue. When a sequence is inserted into the cache, the prefix of the sequence is loaded into the MRU end of the Up queue while its suffix is loaded into the MRU end of the Down queue. For clarity, let's assume that the prefix (suffix) of each sequence fit into one block. Therefore, each prefetch request is for at most 2 blocks. When 2 blocks are prefetched, the first block is loaded into the Up queue and the second block is loaded into the Down queue. When a block is ejected from the LRU end of the Up

Table 6: Split vs. LRU: demonstrating that Split may perform worse than LRU. Prefetch 2 blocks on miss; prefetch on hit of last block in sequence

Prefetch cache size = 6, SplitLRU								
hits				h1	h2			
workload	**1**	**2**	**3**	**2a**	**1a**	**4**	**5**	**3a**
Up Q	1a	2a	3a	2b	1b	4a	5a	3b
		1a	2a	3a	2b	1b	4a	5a
			1a	1a	3a	2b	1b	4a
Down Q	1b	2b	3b	3b	3b	4b	5b	3c
		1b	2b	1b		3a	2b	1b
			1b			3b	4b	5b
Eject							**3a**	**2b**
							3b	**4b**

Prefetch cache size = 6, LRU								
hits				h1	h2			h3
workload	**1**	**2**	**3**	**2a**	**1a**	**4**	**5**	**3a**
Cache	1a	2a	3a	3a	3a	4a	5a	3b
	1b	2b	3b	3b	3b	4b	5b	5a
		1a	2a	2b	2b	3a	4a	5b
		1b	2b	1a	1b	3b	4b	4a
			1a	1b		2b	3a	4b
			1b			1b	3b	
Eject							**2b**	
							1b	

Table 7: Split vs. StreamLRU: demonstrating that StreamLRU may perform better than SplitLRU. Prefetch 2 blocks on miss; no prefetch on hit

Prefetch cache size = 4, SplitLRU											
hits				h1		h2			h3		
workload	**1**	**3**	**2**	**1a**	**4**	**4a**	**5**	**1b**	**5a**	**6**	**4b**
Up Q	1a	3a	2a	2a	4a	4b	5a	1c	1c	6a	4c
		1a	3a	3a	2a	2a	4b	5a		1c	6a
Down Q	1b	3b	2b	2b	4b	3a	5b	1d	1d	6b	4d
		1b	1a		3a		2a	4b	4b	1d	1c
Eject			**3b**		**2b**		**3a**	**2a**		**4b**	**6b**
			1b					**3a**			**1d**

Prefetch cache size = 4, StreamLRU											
hits						h1		h2	h3		h4
workload	**1**	**3**	**2**	**1a**	**4**	**4a**	**5**	**1b**	**5a**	**6**	**4b**
Cache	1a	3a	2a	1b	4a	4b	5a	5a	5b	6a	6a
	1b	3b	2b	1c	4b	1b	5b	5b	4b	6b	6b
		1a	3a	2a	1b	1c	4b	4b		5b	5b
		1b	3b	2b	1c		1b			4b	
Eject			**1a**	**3a**	**2a**						
			1b	**3b**	**2b**						

Table 8: Demonstrating 2 anomalies: Using the same workload and prefetch technique as in Table 7, showing that Split gives more hits than StreamLRU when the cache size is increased. Also demonstrating Belady's anomaly with StreamLRU: 4 hits when cache size is 4 (Table 7), while 3 hits when cache size is 5

Prefetch cache size = 5, SplitLRU											
hits				h1		h2			h3		h4
workload	**1**	**3**	**2**	**1a**	**4**	**4a**	**5**	**1b**	**5a**	**6**	**4b**
Up Q	1a	3a	2a	2a	4a	4b	5a	1c	1c	6a	6a
		1a	3a	3a	2a	2a	4b	5a	4b	1c	1c
			1a		3a	3a	2a	4b		4b	
Down Q	1b	3b	2b	2b	4b	2b	5b	1d	1d	6b	6b
		1b	3b	3b	2b		3a	2a	2a	1d	1d
Eject			**1b**		**3b**		**2b**	**5b**		**2a**	
								3a			

Prefetch cache size = 5, StreamLRU											
hits				h1		h2			h3		
workload	**1**	**3**	**2**	**1a**	**4**	**4a**	**5**	**1b**	**5a**	**6**	**4b**
Cache	1a	3a	2a	2a	4a	4b	5a	1c	5b	6a	4c
	1b	3b	2b	2b	4b	2a	5b	1d	1c	6b	4d
		1a	3a	3a	2a	2b	4b	5a	1d	5b	6a
		1b	3b	3b	2b	3a	2a	5b	4b	1c	6b
			1a		3a		2b	4b		1d	5b
Eject			**1b**				**3a**	**2a**		**4b**	**1c**
								2b			**1d**

queue, it is inserted into the Down queue, behind the block that was just inserted into the Down queue (the second position from the MRU end of Down queue). All evictions from the cache are from the Down queue. Thus, the Split policy allows sequences to stay longer in the cache than the Stream approach. The SplitLRU policy has the computational complexity of LRU.

When the Split policy is viewed from a single queue perspective, a newly inserted sequence is inserted into the top two MRU positions of the LRU queue. This results in the ejection of 2 blocks from the LRU end of the queue. If the block to be ejected is the suffix of a sequence, then it is evicted; if the block is a prefix and it is appearing at the LRU end for the first time, then this block is re-inserted into the LRU queue just after the newly inserted sequence (that caused the eviction of this block). When the prefix of a sequence appears at the LRU end for the second time, it gets evicted. Thus, the Split approach gives sequences a second chance.

The examples presented in Table 4 and 5 show that, in each case, Split does as well as the better of the Stream and LRU policies (reference Tables 1 and 2). SplitLRU uses the underlying principle that the later blocks in a sequence are expected to receive hits later than the earlier blocks. Sequences are given a longer life in the cache than in the traditional StreamLRU and LRU approaches, so the superior hit rate of SplitLRU is not surprising. This reasoning would lead one to believe that the hit rate of SplitLRU is an upper bound to the hit rate of LRU and LRUseq. Unfortunately, this is not the case, as demonstrated by the examples in Tables 6 and 7.

Table 6 presents a scenario where LRU results in more hits than Split. The reason for LRU's superior performance is that the workload is biased to the FIFO replacement policy, and LRU evicts prefetched blocks in FIFO order. Table 7 presents a scenario where StreamLRU gets more hits than Split. Table 8 presents a rather bizarre scenario: when the cache size is increased, StreamLRU gets less hits than Split using the same workload and prefetch technique as in Table 7. The advantage that Stream has over Split is lost when the cache size is increased. Therefore, in addition to the prefetch technique, the cache size is another factor that determines the relative performances of replacement policies. On a side note, Tables 7 and 8 demonstrate the occurrence of Belady's anomaly [3] with StreamLRU since the large cache gets less hits than the smaller cache. This contradicts an assumption made in an earlier paper [10] about StreamLRU being foolproof to Belady's anomaly.

The examples in Sections 3 and 4 show that the relative performances of prefetch cache replacement policies vary with the workload, the prefetch technique, and the size of the cache. It is hard to understand how one replacement policy compares with another when there are so many factors to consider. This raises the question, how does one determine the optimal replacement policy when performance is so erratic? We answer this question in the next two Sections.

5 Replacement policy invariant prefetch

In order to compare two replacement policies, all other parameters must be identical - the same cache size, the same prefetch technique, the same input workload. In the examples given so far, all these parameters are identical, but one is unable to draw any conclusion regarding the relative performances of the replacement policies. The reason is that the action of a prefetch technique is affected by the replacement policy. For example, consider a prefetch technique that only prefetches on misses: suppose an on-demand request gets a miss with one replacement policy and a hit with another. The miss results in a prefetch of contiguous blocks, while the hit results in no prefetch. On a hit/miss, the prefetch action may load different blocks into the two caches, but the prefetching of different blocks may result in a hit in one cache and a miss in another. That is, the hits/misses are a result of the combined action of the replacement policy and prefetch technique. Instead of evaluating the relative performances of replacement policies, one is really evaluating the relative performances of the replacement policy and prefetch technique combination.

In a sense, comparing two prefetch cache replacement

policies is like trying to compare apples and oranges, even when the cache size, the workload, and the prefetch technique are identical. The reason is that the blocks prefetched by a prefetch technique depends on whether an on-demand request hits or misses in the cache. Instead of comparing replacement policies against the backdrop of identical prefetch techniques, one should compare policies against the backdrop of *identical prefetch actions*. That is, a fixed number of blocks should be prefetched upon arrival of an on-demand request, independent of whether the request gets a hit or a miss. If some of the blocks to be prefetched are already present in the cache, then they are not prefetched again. A prefetch technique that prefetches a fixed number of blocks upon arrival of each on-demand request, regardless of whether the on-demand request hits or misses, is impervious to the replacement policy.

Classify prefetch techniques into two groups depending on whether the prefetch degree (*i.e.*, the number of blocks prefetched upon arrival of an on-demand request) is *fixed* or *variable*. The technique in Table 1 belongs to the fixed class since the prefetch technique ensures that two blocks contiguous to the arriving on-demand request are in the cache. The prefetch technique in Table 2 belongs to the variable class: on miss, the prefetch technique ensures that 2 blocks contiguous to the missed block are in cache; on hit of non-trigger block, there is 0 prefetch; on hit of trigger block, the prefetch technique ensures that 2 contiguous blocks are loaded in cache. The trigger block is the suffix of the sequence, but if the suffix is evicted, the trigger moves to the prefix (*i.e.*, the trigger is the last in-cache block of the sequence). Techniques that prefetch only on hit or only on miss, and techniques that prefetch on hit of trigger blocks are all examples of variable class prefetch techniques.

In this section, we theoretically compare the performances of LRU, StreamLRU and SplitLRU when the prefetch technique belongs to the fixed class. The prefetch actions are identical in all caches, so performance variations are a result of the replacement policy alone. Consequently, we are able to isolate the performance characteristics of the replacement policies with regard to prefetch blocks. While it is true that commonly used prefetch techniques belong to the variable class, the performance is a result of the interaction between the prefetch technique and the replacement policy. In order to evaluate the integrated performance of a variable prefetch technique and a specific replacement policy, it is necessary to first isolate the performance characteristics of the replacement policy.

5.1 Assumptions & notation

The measurement unit used in the paper is blocks. The prefetch technique ensures that $x \geq 2$ blocks contiguous to the newest on-demand request are in the cache. A minimum of 2 blocks must be prefetched in order to see differences between the three replacement policies. For simplicity, let x be an even number, and let the prefix and suffix of a sequence contain equal number of blocks, $x/2$.

The key to modeling and analysis is the ability to keep essential features of the system being modeled, while blocking out the noise. In this spirit, we impose simplifying assumptions:

1. The cache size is C blocks, where C is a multiple of $x/2$. This assumption ensures that the cache, including the Up queue and Down queue, does not store/evict partial prefixes/suffixes.

2. When a block receives a hit, the hit block is evicted from the prefetch cache. The hit block may be moved to the reference cache. With this assumption, the analysis focuses on the prefetch blocks, without the distraction of reference blocks.

3. The prefetch technique is from the fixed class.

4. The on-demand workload follows Assumption 2 (refer to Section 3), namely, that later blocks of a sequence have a later first access time than the blocks that precede them. A consequence of this assumption is that if earlier blocks of a sequence are evicted from the cache, but the cache holds later blocks of the sequence, then the next on-demand request for a block from this sequence will miss. That is, the sequence is effectively evicted from the cache.

Prefetching is all about access sequences, so all results pertain to sequences. Let $\#_{\mathsf{Stream}}$, $\#_{\mathsf{Split}}$ and $\#_{\mathsf{LRU}}$ represent the number of sequences 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 sequences in the cache when the replacement policy is StreamLRU, SplitLRU and LRU, respectively.

5.2 Analysis

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 sequence to the MRU end upon a hit.

Result 2 *The StreamLRU policy ensures that the most*

recently accessed $\lceil \frac{C}{x} \rceil$ sequences are in the cache.

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

Result 3 *The SplitLRU policy ensures that at least $\lceil \frac{C}{x} \rceil$ and at most $\lceil \frac{C}{x} \rceil + \lfloor \frac{C}{2x} \rfloor$ of the most recently accessed sequences are in the cache.*

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

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

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

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

When a prefix is ejected from the Up queue, it is moved to the Down queue, behind the suffix of the newly inserted sequence. With access of each new sequence, 2 prefix/suffix are inserted in the Down queue - the suffix from the new sequence and the prefix evicted from the Up queue. Therefore, with each sequence insertion, every Down block moves x positions downward toward the eviction end. Thus, a prefix stays in the Down queue for $\lfloor \frac{C}{2x} \rfloor$ unique sequence accesses.
□

The next results follow directly from Results 2 and 3.

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

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

We now compare LRU against Split and Stream. The prefetch technique ensures that x blocks contiguous to the on-demand request are loaded in the cache, but the LRU policy ignores these access sequences. 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. The prefetch blocks are evicted in FIFO order by the LRU replacement policy. The older contiguous blocks would get evicted before the newer contiguous blocks. Thus, the LRU policy may evict the prefix of a sequence before the suffix of this sequence. As a result, it is possible for an on-demand request to get a miss even though a later block of the sequence is present in the cache. If a later block of a sequence is present in the cache, but the earlier block is evicted, then the corresponding sequence is said to be evicted from the cache. A consequence of evicting prefetch blocks in FIFO order is:

Result 4 *The LRU policy evicts sequences in FIFO order and holds at most $\lceil \frac{C}{x} \rceil$ access sequences in the cache.*

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

Note that the StreamLRU and SplitLRU policies keep the most recently accessed sequences in the cache. The LRU policy evicts sequences in FIFO order, so the sequences in the LRU cache are not necessarily the most recently accessed sequences. As a result, even though the LRU cache may hold less sequences, the LRU cache need not be a subset of the SplitLRU cache or the StreamLRU cache. In the LRU cache, between two accesses of a sequence, it is possible to have $2 * \lceil \frac{C}{x} \rceil - 2$ unique sequence accesses. Table 1 demonstrates that even though the LRU cache may contain less sequences than the Stream cache, it is possible for a LRU cache to get more hits than the Stream cache. The reason for LRU's superior performance is that the workload, $< 1, 2, 1a, 3, 2a >$, is biased toward the FIFO policy. In order to show that LRU may get more hits than Split, consider the workload $< 2, 3, 4, 1, 2a, 3a, 4a, 5, 6, 7, 1a >$ when the cache size C=16 blocks and the degree of prefetch $x = 4$. LRU gets 4 hits while Split gets 3 hits since the workload is biased to the FIFO policy. While we have provided specific scenarios where LRU gets more hits than Split and Stream, these cases are atypical. Since Split and Stream hold at least as many sequences as LRU, the Split and Stream policies are expected to get more hits than the LRU cache.

The theoretical analysis in this section allows one to understand the essential traits of LRU, StreamLRU and SplitLRU. However, there are a couple of major issues that have not been addressed here. The first issue is whether the superior hit rate of Split translates into lower response time. The response time is determined not only by the hit rate but also by the workload traffic submitted from the cache to the lower level. Even though Split has a higher hit rate, it does evict and then reload the suffixes of sequences. It is important to theoretically and experimentally evaluate the impact of the traffic submitted from the cache to the lower level. The second issue is to evaluate the relative performances of the three

Table 9: 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

policies against a variable class technique. If the relative performances violate the analysis here, then the causal agent is the prefetch technique. It would be easier to understand the interaction between the replacement policy and the prefetch technique, now that the properties of the replacement policy are known. In the next section, we address both these issues by a simulation analysis of the relative performances of the three replacement policies when combined with a popular variable class prefetch technique.

6 Experimental evaluation

The performances of replacement policies LRU, StreamLRU and SplitLRU are evaluated when the prefetch technique is from the variable class. We point out that SARC is a combination of StreamLRU and adaptive sizing of prefetch versus reference cache. To avoid conflating the effects of adaptive resizing of the prefetch cache with Split, the paper compares Split with LRU and StreamLRU.

Since the focus of this study is replacement policies, not prefetch techniques, we simulate a simple version of a common prefetch technique, namely, the trigger based prefetch. If a requested block misses in the prefetch cache, then an additional two blocks are prefetched synchronously, and piggy-backed with the missed disk request. If a requested block hits in the prefetch cache, and the following block in the sequence is not present in the prefetch cache, two blocks are prefetched asynchronously. That is, the last in-cache block of a sequence is the trigger block. No prefetch is initiated if a non-trigger block receives a hit.

We developed an event driven simulator to evaluate the prefetch cache replacement techniques. The front end model is our cache simulator, while the back end storage model is a single disk simulated by the Disksim 4.0 simulator [6]. Table 9 gives the setup used for our experiments. We chose a simple disk setup in favor of a more complicated RAID configuration, because the performance of a single disk is more predictable. By minimizing idiosyncrasies of the back end storage model, we more effectively isolate the performance impact of the cache replacement policy.

Performance metrics: The hit rate, mean response time, wastage rate and disk rate are measured. The hit rate is the ratio of the number of hits to the total number of on-demand requests to the prefetch cache; the mean response time is the product of miss rate and mean disk response time; wastage rate is the ratio of the number of prefetches that get misses (due to early eviction) to the total number of on-demand requests to the prefetch cache; disk rate is the ratio of number of requests to disks to the total number of on-demand requests to prefetch cache.

Input parameters: The cache size and the I/O workload are varied in our experiments. The cache size and I/O workload have to be set in tandem; if the cache is too large or too small for the workload, replacement policies will have little impact on performance.

Workload characteristics: Using real workload traces is not an option since prefetching depends on timing [9, 10, 18]. Synthetic workload traces are generated based on published SPC1 and SPC2 specifications for read only workloads. SPC1 [1, 14, 20] and SPC2 [2] are popular benchmark that simulate sequential access patterns of business and desktop applications, respectively. Each of our workload traces is composed of several independent, concurrent sequences of requests with exponentially distributed inter-arrival times. Three distinct types of workload sequences are generated, completely random, completely sequential, and partially sequential. Completely random sequences issue requests with random block addresses. Completely sequential sequences request a random starting block address, and subsequent requests sequentially increment the block address by one. Partially sequential sequences request a random starting block address, followed by a number of requests for sequential block addresses, then restart at another random starting block address. Thus, a partially sequential sequence consists of a number of sequential *runs*. The run length is sampled exponentially resulting in sequences with both short and long runs.

Cache size: The cache size is varied across the range appropriate for each workload. For a workload of 100 sequences and two blocks prefetched per sequence, the cache performance is evaluated up to a cache size of 300 blocks, because performance metrics for all replacement techniques level when the cache size approaches this value. In Split replacement policy, we also vary the relative sizes of the Up and Down queues. In the default Split configuration, the Up and Down queue are the same size, each half of the total cache capacity. In the Split23 configuration, the Up queue is two thirds, and the Down queue is the remaining one third of the cache capacity. The ability to vary the relative cache sizes provides a simple mechanism to tune the behavior of the replacement policy.

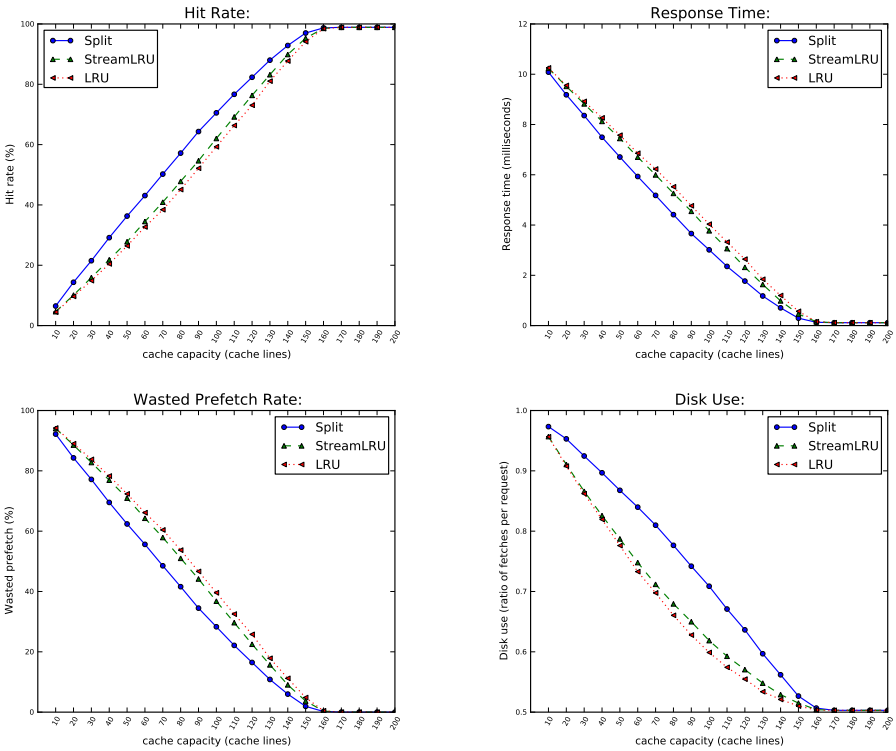

Figure 1: SPC2-like workload with 100 sequential sequences

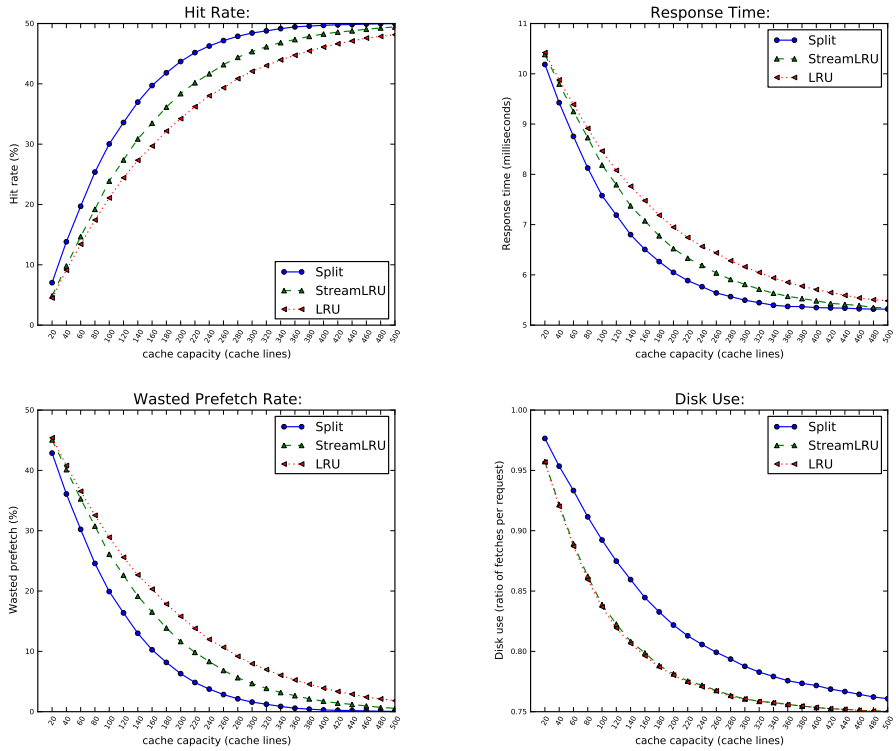

Figure 2: SPC2-like workload with 50 sequential sequences and 50 random sequences

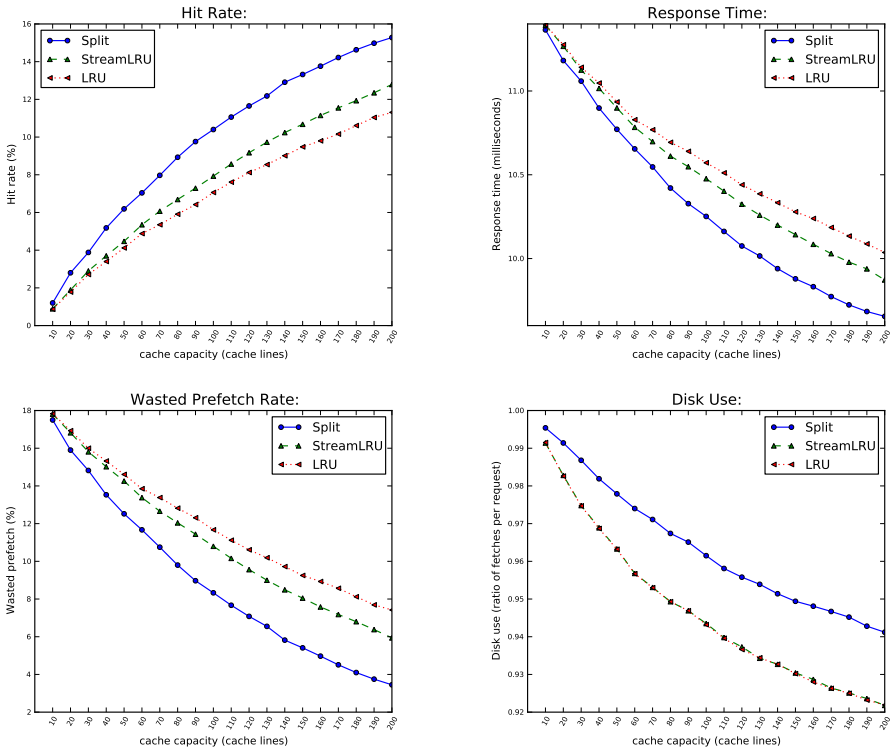

Figure 3: SPC1-like workload with 80 random sequences and 20 partly sequential sequences

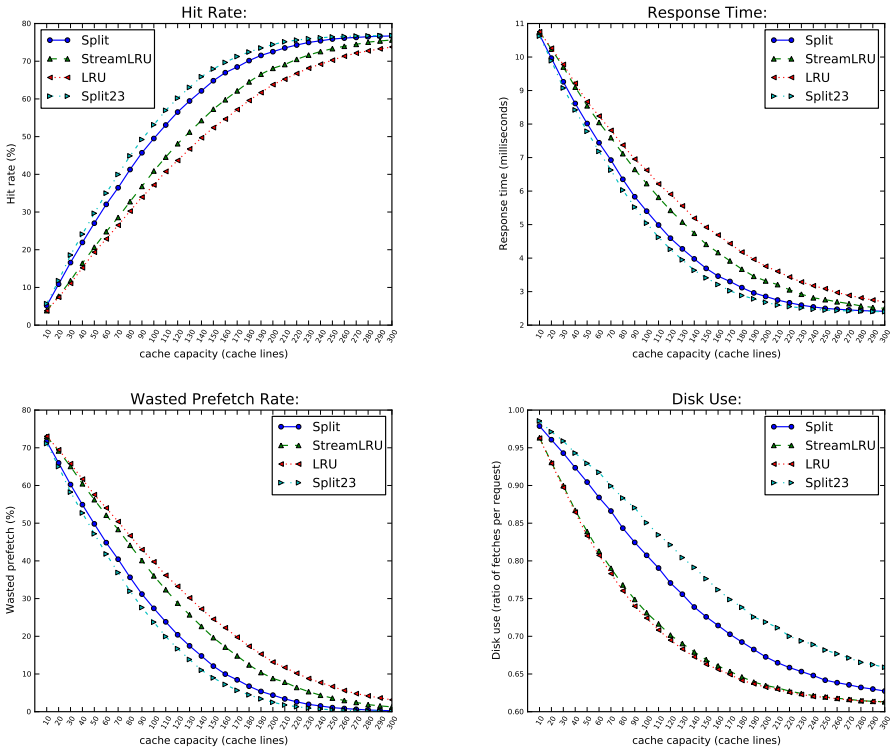

Figure 4: Adding the Split23 replacement policy: 50 Sequential, 20 random and 30 partly sequential sequences

Analysis: Figures 1, 2, 3 and 4 show that the Split policy consistently gets a higher hit rate and a lower response time than either of the other policies. This result is not surprising given that Split holds more sequences than either of the other policies. Split has a higher disk use rate than the other two policies since fewer blocks per sequence remain in the cache compared to the other policies. However, Split also has a lower wastage rate than other policies since the evicted suffixes of sequences are prefetched when their corresponding prefixes get a hit. Consequently, the early evictions are prefetched again before the arrival of corresponding on-demand requests. Therefore, in Split, the higher disk rate is counter balanced by a lower wastage rate.

By varying the relative sizes of the Up and Down queues, one can tune the Split replacement policy to the workload. When the size of the Up queue is larger than the Down queue, the prefix of each sequence remains longer in the cache at the cost of evicting the suffix sooner. The result is a higher cache hit rate and disk use rate, and a lower wastage rate and response time (Figure 4). The Split policy degenerates to single block read ahead as the size of the Up queue approaches the total cache size. 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 sequences.

The Split Up and Down queues can be used to dynamically extract information regarding 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 sequences may be getting evicted too soon. This can be addressed in the Split replacement policy by increasing the size of the Up queue. External to the replacement policy, the degree of prefetch can be reduced, or prefetching can be disabled until a sequential sequence is detected. These approaches may be used individually or cooperatively. By extracting this information, the Split policy and the prefetch technique can be tuned dynamically to optimize the cache to react to changes in the workload.

7 Conclusion

The Split replacement policy is unique in that it incorporates both the temporal and spatial locality of workloads. Theoretical analysis and simulation studies demonstrate that a Split cache has a lower response time and higher hit rate than a LRU cache and a StreamLRU cache. Since the StreamLRU policy is used by SARC to determine which block to evict, the Split prefetch cache performs better than the SARC prefetch cache.

The paper reinforces the conclusion drawn in an earlier paper [7], namely, the interaction between prefetching and caching is complex. A contribution of this paper is a demonstration of the importance of using a fixed class prefetch technique while comparing the relative performances of replacement policies. Without enforcement of this rule, the cache performance is the combined impact of the replacement policy and the prefetch technique.

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 need to be addressed include a study of the impact of varying the sizes of the Up and Down queues, analysis of traffic generation by the policies, theoretical analysis of Split when combined with a variable-class prefetch technique, and prefetch cache sizing based on hit rates in Up and Down queues. As future work, we plan to look into these issues.

References

[1] SPC Benchmark-1(SPC-1) Official Specification, revision 1.10.1. Tech. rep., Effective 27 Sept. 2006. `http://www.storageperformance.org/specs`.

[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] BELADY, L. A., NELSON, R. A., AND SHEDLER, G. S. An anomaly in space-time characteristics of certain programs running in a paging machine. *Commun. ACM 12* (June 1969), 349–353.

[4] 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.

[5] BOVET, D. P., AND CESATI, M. *Understanding the Linux Kernel, Third Edition*. O'Reilly Media, 2005.

[6] BUCY, J. S., AND GANGER, G. R. The DiskSim simulation environment version 4.0 reference manual. Tech. Rep. CMU-PDL-08-101, Carnegie Mellon University, School of Computer Science, May 2008.

[7] 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.

[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] HARTSTEIN, A., SRINIVASAN, V., PUZAK, T. R., AND EMMA, P. G. Cache miss behavior, is it $\sqrt{2}$? *Proceedings of the 3rd conference on computing frontiers* (2006), 313 – 320.

[13] 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.

[14] JOHNSON, S., MCNUTT, B., AND REITH, R. The making of a standard benchmark for open system storage. In *J. Comput. Resouce Management* (Winter 2001), no. 101, pp. 26–32.

[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] 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.

[17] LI, C., AND SHEN, K. Managing prefetch memory for data-intensive online servers. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies* (2005), vol. 4, pp. 253 – 266.

[18] 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.

[19] 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–.

[20] MCNUTT, B., AND JOHNSON, S. A standard test of I/O cache. In *Proceedings on Computer Measurement Group's 2001 International Conference* (2001).

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

[22] 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.

[23] 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.

[24] PAPATHASIOU, A. E., AND SCOTT, M. L. Energy efficient prefetching and caching. In *Proceedings of the USENIX Annual Technical Conference* (June,2004).

[25] SINGH, J., STONE, H., AND THIEBAUT, D. A model of workloads and its use in miss-rate prediction for fully associative caches. *IEEE Trans. on Computers 41*, 7 (1992), 811–825.

[26] SMITH, A. J. Sequentiality and prefetching in database systems. *ACM Transanctions on Database Systems 3*, 3 (1978), 223–247.

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

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

[29] WILSON, P. R., KAKKAD, S. V., AND MUKHERJEE, S. S. Anomalies and adaptation in the analysis and development of prepaging policies. *Journal of Systems and Software 27* (1994), 147–153.

[30] 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).