Removing Belady's Anomaly from Caches with Prefetch Data

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

Abstract

Belady's anomaly occurs when a small cache gets more hits than a larger cache, given identical input conditions regarding the workload and caching algorithms. The implication of this anomaly is that upgrading a cache could result in a lowering of performance. This paper proves that Belady's anomaly may occur with almost all combinations of prefetch techniques and cache replacement policies. The repercussions of the ubiquity of this anomaly is that it is very difficult to evaluate the performance of a cache with prefetching. This paper analyzes why Belady's anomaly is an inherent feature of caching and prefetching systems. The anomaly occurs because the content and ordering of the cache replacement queue are dependent on the prefetch cache size. Based on this evaluation, the paper presents a prefetch technique and an LRU variant that does not exhibit the anomaly.

1 Introduction

The file system cache is an integral part of operating systems. A user read request is first submitted to the file system cache. If the requested data block is in the cache, a hit occurs. A user read request is submitted to the lower level cache/storage device only upon a cache miss. The missed data blocks are loaded into the file system cache on the assumption that more user requests for these blocks will arrive in the near future.

The file system's cache workload is a sequence of requests for data blocks ordered by the time at which the requests arrive at the cache. Thus, the workload submitted to a cache is a sequence of (logical) block numbers, where a block's position in the sequence refers to the relative time at which it was requested. For example, in the workload sequence $< 2, 5, 4, 7, 2, 10, 3 >$, block 2 was first requested before block 5; block 2 was requested again at a later point in time (after block 7). A cache contains blocks that are requested by the workload. Caches are small, so they may hold only a small fraction of the requested data blocks.

When a new data block has to be loaded into a full cache, the cache *replacement policy* determines which cached block to evict. The goal of a replacement policy is to keep blocks that may receive user requests in the near future. The *hit ratio* of a cache is defined to be the ratio of the total number of requests that hit in the cache to the total number of requests in the workload. The higher the hit ratio, the better is the performance. The cache workload, the cache size, and the cache replacement policy collectively determine the hit ratio of the cache. Some of the common replacement policies are LRU (Least Recently Used), FIFO (First In First Out), LFU (Least Frequently Used), and their variants.

For a fixed workload and replacement policy, the hit ratio is a function of the cache size. One would naturally assume that the hit ratio would not decrease as the cache size increases. Belady et al. [1] demonstrated that this assumption is false by using the workload $< 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 >$ and the FIFO replacement policy. Here, a cache of size 3 blocks gets three hits while a cache of size 4 blocks gets only two hits. **Belady's anomaly** refers to this seemingly illogical scenario of a small cache getting more hits than a larger cache.

The purpose of upgrading a system is to improve performance. Belady's anomaly is antithetical to this objective since increasing the cache size results in a worsening of performance. For a given workload and cache size, the contents of the cache are entirely dependent on the cache replacement policy. Hence, one can conclude that Belady's anomaly is caused by the replacement policy. Mattson et al. proved that the anomaly does not occur with the LRU replacement policy [9]. That is, with LRU and any workload, it is guaranteed that the hit ratio is a non-decreasing function of the cache size,

Almost all file system caches implement sequential prefetching (*i.e.,* read-ahead). Here, data blocks contiguous to the requested data block, are loaded into the cache

on the assumption that user requests for the prefetched blocks will arrive in the near future. Prefetching is handled by the cache *prefetch technique*.

The inclusion of prefetching changes the status quo of caches; the hit ratio depends not only on the workload, cache size and replacement policy but also on the prefetch technique. Prior work has demonstrated that Belady's anomaly may occur with the LRU replacement policy when a cache implements prefetching [7]. While it is well documented that the interaction between prefetching and caching is complex [3, 4, 5], there is very little documentation on why Belady's anomaly occurs with prefetching or how the anomaly could be removed. An understanding of why Belady's anomaly occurs may shed light on some aspects of the complex relationship between prefetch techniques and replacement policies, thereby paving the way for better prefetching and caching systems.

Contributions: This paper analyzes Belady's anomaly in the context of file system caches with sequential prefetching and LRU. Most file system caches implement LRU or a variant of LRU [6]. Sequential prefetching is common in file system and storage caches since files are often read sequentially [6]. An LRU cache without prefetching does not exhibit Belady's anomaly. Unfortunately, the addition of prefetching causes an LRU cache to exhibit the anomaly. While Belady's anomaly is a rare event in caches without prefetching, the anomaly is quite common in prefetch caches using LRU [7]. The paper shows that Belady's anomaly is inherent to (1) all but one class of prefetching techniques, regardless of the replacement policy; and (2) the LRU replacement policy, regardless of the prefetch technique. This paper then presents a prefetch and replacement algorithm that is free of Belady's anomaly.

This paper presents the following:

1. documentation of the prevalence of Belady's anomaly in caches with prefetching - the anomaly is an inherent feature of common prefetching techniques;

2. presentation of a proof that Belady's anomaly does not occur with LRU and a prefetch technique that prefetches exactly one block contiguous to the arriving user request;

3. presentation of a proof that Belady's anomaly can occur in prefetch caches with the LRU replacement policy and any prefetch technique that prefetches more than one block contiguous to an arriving user request;

4. formalization of an LRU variant that ensures the *stack* property [9] of prefetch blocks - this policy is named *StreamLRU*; demonstration that Belady's anomaly occurs even with StreamLRU;

5. presentation of a proof that Belady's anomaly can occur with all but one class of prefetch techniques and LRU/StreamLRU; and

6. presentation of a proof that a caching system with StreamLRU and a prefetch technique that ensures a fixed number of blocks contiguous to each arriving request are loaded in the cache is free of Belady's anomaly.

2 Workload

The operating system maps user read requests for bytes into file system read requests for blocks. Since this paper discusses file system caches, the unit of measurement used is blocks: a cache size is C blocks, a request is for one or more blocks. A cache workload consists of user requests. For notational simplicity, we assume that each user request is for a single block. Let t_n represent the relative time instant at which the n^{th} read request arrives. The n^{th} user read request submitted to the cache is described by:

$$x_n = i; \;\; i \in \{1, 2, 3, \cdots, \mathsf{MaxBlocks}\}$$

where i is the block number to be read. The relative time instant t_n at which the read request x_n arrives at the cache is implicitly defined in the notation. The cache workload of user requests is given by the sequence:

$$\mathcal{X} =< x_1, x_2, x_3, \cdots, x_{\mathsf{N}} >$$

where N is the number of read requests that arrive during the observation period. The notation x_i is used interchangeably to represent both block x_i and a request for block x_i; the distinction should be clear from the context in which the notation is used.

Example 1 *A file system's cache workload:* $\mathcal{X}$ *= < 1, 51, 99, 151, 89, 2, 3, 152, 999, 52, 4, 5, 251, 799, 53, 6, 351, 299, 199, 899, 7, 54, 699, 599, 252, 499, 253, 3999, 352, 353, 451, 399, 8999, 7999, 6999, 254, 452, 8, 453, 501, 5999, 4999, 502 >*

The workload appears as a single sequence of random block numbers. A more careful examination reveals the following interleaved requests to contiguous blocks:
Stream s1: $1, 2, 3, 4, 5, 6, 7, 8$ at time instants t_1, t_6, t_7, t_{11}, t_{12}, t_{16}, t_{21}, t_{38};
Stream s2: $51, 52, 53, 54$ at time instants t_2, t_{10}, t_{15}, t_{22};
Stream s3: $151, 152$ at time instants t_4, t_8;
Stream s4: $251, 252, 253, 254$ at time instants t_{13}, t_{25}, t_{27}, t_{36};

Stream s5: $351, 352, 353$ at time instants t_{17}, t_{29}, t_{30};
Stream s6: $451, 452, 453$ at time instants t_{31}, t_{37}, t_{39};
Stream s7: $501, 502$ at time instants t_{40}, t_{43}.

Definition 1 *A* **stream** *is a sequential or interleaved sequential access pattern in a workload.*

A typical file system/storage cache workload consists of user requests from various streams. The workload appears random since the streams are interleaved. During the observation period (t_1 to t_{43}), there are 17 block accesses that are not part of any of the above streams, namely, 99, 89, 999, 899, 799, 299, 199, 699, 599, 499, 399, 299, 8999, 7999, 6999, 5999, 4999. These requests may be for random blocks; they may also represent the end of streams whose requests arrived prior to the observation period or the start of streams whose future requests may arrive after the observation period. From this perspective, every request in a cache workload can be mapped to a stream. If every request is part of a stream, then the cache workload can be viewed as a set of interleaved requests from various streams.

It is difficult to extract streams from the seemingly random workload presented in Example 1. In order to quickly identify interleaved streams in an example workload, we reference contiguous blocks as:
$i, ia, ib, ic, ..., \quad i \in I, i \geq 0$

Example 2 *A cache workload using the new notation: < 2, 5, 4, 4a, 4b, 7g, 5a, 2, 5b, 10, 5c, 11b, 3, 3a, 4c, 2, 1, 1a, 4d, 4e >.*
The workload contains the following interleaved streams: <2>, <5, 5a, 5b, 5c>, < 4, 4a, 4b, 4c, 4d, 4e>, <7g>, <10>, <11b>, <3,3a>, <1, 1a>.

When referring to contiguous blocks that relate to stream s, the notation used is
$s =< s_1, s_2, s_3, ..., s_n >$, where $s_1 = i$, $s_2 = i + 1$, ... $s_n = i + n$; $i, n \geq 1$

3 Sequential prefetch

Prefetching is fundamental to the memory hierarchy model, since data blocks are uploaded before they are needed in order to reduce read access time. In order to predict future block accesses, recent block access patterns are identified. It is assumed that these access patterns will continue in the future. Files are often read sequentially, so they are likely to be stored on contiguous blocks. Hence, a cache workload may contain several interleaved streams as shown in Example 2.

The objective of a sequential prefetching technique is to prefetch data blocks from streams before user requests for the prefetched blocks arrive. For the workload in Example 2, a prefetching technique that prefetches blocks 5a, 5b, 5c when the user request for block 5 arrives (and misses in the cache) would result in 3 hits. Of course, this assumes that the prefetched blocks are loaded into the cache on time and that the prefetched blocks remain in the cache until the arrival of user requests.

The prefetch technique determines what blocks should be prefetched and when the blocks should be prefetched. The prefetch technique is initiated each time a user read request arrives at the cache, since a decision has to be made on whether to prefetch future blocks from this read request's stream. Prefetch techniques are broadly classified into two types [11]: **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 prefetch request is piggybacked onto the missed request, so little additional traffic is generated by the prefetch. Thus, PM assumes that the missed request block is the start of a stream. The PA technique generates synchronous or asynchronous prefetch requests whenever a user request arrives at the cache. The synchronous prefetch request is generated when a read request misses in the cache, and loads the start of a stream. If this synchronously prefetched block gets a hit, then future blocks from the stream are asynchronously prefetched. If a future stream block is already in the cache, both PM and PM do not fetch it again.

There are several versions of PA. In a common version of PA, 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.

The cache workload determines what blocks are loaded into the cache. With the inclusion of a prefetch technique, the blocks loaded into the cache are determined by both the user requests and the prefetch requests. Reconsider the workload in Example 2. Assume a PM technique which prefetches 2 blocks contiguous to the missed block. Suppose that the cache size is large enough to ensure that there are no evictions from the cache. Due to the prefetch actions of the PM technique, the sequence of requests that arrive at the cache in order are:
< 2, *2a, 2b*, 5, *5a, 5b*, 4, *4a, 4b*, 4a*, 4b*, 7g, *7h, 7i*, 5a*, 2*, 5b*, 10, *10a, 10b*, 5c, *5d, 5e*, 11b, *11c, 11d*, 3, *3a, 3b*, 3a*, 4c, *4d, 4e*, 2*, 1, *1a, 1b*, 1a*, 4d*, 4e* >.
The blocks in italics are the prefetched blocks. The * represents a cache hit. Now, suppose the cache implements a PA technique which prefetches 2 contiguous blocks 1) on a miss and 2) on a hit of last block in a prefetched stream. Prefetch is not initiated when the block contigu-

ous to the hit block is already in the cache. With the PA technique, the sequence of requests that arrive at the cache in order are:
$<$ 2, *2a, 2b*, 5, *5a, 5b*, 4, *4a, 4b*, 4a*, 4b*, *4c, 4d*, 7g, *7h, 7i*, 5a*, 2*, 5b*, *5c, 5d*, 10, *10a, 10b*, 5c*, 11b, *11c, 11d*, 3, *3a, 3b*, 3a*, 4c*, 2*, 1, *1a, 1b*, 1a*, 4d*, *4e, 4f*, 4e* $>$.

The examples highlight the fact that the incorporation of a prefetch technique impacts what blocks are inserted into the cache and the order in which the blocks are inserted. With prefetching, the cache contains both reference blocks and prefetch blocks.

Definition 2 *The* **reference blocks** *in a cache have received user requests, while* **prefetch blocks** *have not received user requests. Upon a hit, a prefetch block becomes a reference block since it has now received a user request.*

In the previous example, *4a* is a prefetched block, while 4a* is a referenced block.

3.1 Spatial locality

Prefetch techniques PA and PM, by their definition, make the following assumptions about the cache workload:

Assumption 1 *The cache workload is a sequence of interleaved requests from various streams.*

Assumption 2 *Every prefetched block is expected to receive a future user request from one of the interleaved streams.*

Assumption 3 *For stream s, when block s_i is requested, all blocks $s_j, j < i$ have already received user requests, and the contiguous block s_{i+1} is expected to be the next stream block to receive a user request.*

That is, both PA and PM assume that user requests for blocks in a stream arrive in order of increasing block numbers. Note that this condition may be relaxed so that any block in the current prefetch window (*i.e.,* the set of blocks in a prefetch stream that must be kept in cache [2]) may be requested. For simplicity of notation and clarity, this paper does not use the concept of prefetch windows, and instead, uses Assumption 3.

These assumptions refer to the spatial locality of the cache workload. If the workload does not follow these assumptions, then both PA and PM perform poorly. This is not surprising, since the success of sequential prefetch techniques depends on the degree to which the workload displays sequential locality of reference.

4 Belady's anomaly

Belady's anomaly was first demonstrated and studied with reference to demand paging in main memory. Belady et al. gave a specific example, shown in Table 1, that demonstrated the occurrence of the anomaly with the FIFO policy [1]. Mapping from memory pages to cache blocks, suppose the large cache is of size L blocks and the small cache is of size S blocks. The paper presented a production rule for generating a workload that would result in Belady's anomaly when the replacement policy is FIFO and the sizes of the caches satisfy the condition $S < L < 2S - 1$.

Table 1: Original example demonstrating Belady's anomaly

Reference cache size = 3, FIFO												
hits								h1	h2			h3
workload	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
Cache	1	2	3	4	1	2	5	5	5	3	4	4
		1	2	3	4	1	2	2	2	5	3	3
			1	2	3	4	1	1	1	2	5	5
Eject				**1**	**2**	**3**	**4**			**1**	**2**	

Reference cache size = 4, FIFO												
hits					h1	h2						
workload	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
Cache	1	2	3	4	4	4	5	1	2	3	4	5
		1	2	3	3	3	4	5	1	2	3	4
			1	2	2	2	3	4	5	1	2	3
				1	1	1	2	3	4	5	1	2
Eject							**1**	**2**	**3**	**4**	**5**	**1**

Mattson et al. proved that Belady's anomaly can never occur with LRU, regardless of the workload, since LRU belongs to the class of **stack algorithms** [9]. The reason that LRU is a stack algorithm is that the LRU replacement queue ordering is not dependent on the size of the cache. A cache of size C holds the C most recently used distinct blocks, while a cache of size C+1 holds all these C blocks plus the distinct block accessed just prior to these C blocks. The LRU queue ordering is entirely based on recency of access.

The above results are valid for caches that do not prefetch. Wilson et al. presented an example demonstrating that the LRU replacement policy with the PA technique is not a stack algorithm [12]. Gill et al. presented a simulation study demonstrating that the the LRU policy and the PA technique resulted in Belady's anomaly [7]. The focus of both papers was not Belady's anomaly, so they did not explain why the anomaly occurred. The rest of this section presents examples that show the occurrence of Belady's anomaly with the LRU policy and both classes of prefetch techniques, PA and PM.

Tables 2 and 3 show the occurrence of Belady's anomaly with the LRU policy and the PA, PM techniques, respectively. These examples show caches that

Table 2: Prefetch the next block always

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

Reference + Prefetch cache size = 8, PA, LRU								
hits				h1		h2		
workload	**1a**	**2a**	**3a**	**1a**	**4a**	**2a**	**5a**	**2b**
Cache	1a	2a	3a	1a	4a	2a	5a	2b
	1b	2b	3b	3a	4b	4a	5b	2c
		1a	2a	3b	1a	4b	2a	5a
		1b	2b	2a	3a	1a	4a	5b
			1a	2b	3b	3a	4b	2a
			1b	1b	2a	3b	1a	4a
					2b	2b	3a	4b
					1b	1b	3b	1a
Eject							**2b**	**3a**
							1b	**3b**

Table 3: Prefetch the next block on a miss

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

Reference + Prefetch cache size = 7, PM, LRU										
hits					h1	h2				
workload	**1a**	**2a**	**3a**	**4a**	**1a**	**2a**	**5a**	**1b**	**2b**	**1a**
Cache	1a	2a	3a	4a	1a	2a	5a	1b	2b	1a
	1b	2b	3b	4b	4a	1a	5b	1c	2c	2b
		1a	2a	3a	4b	4a	2a	5a	1b	2c
		1b	2b	3b	3a	4b	1a	5b	1c	1b
			1a	2a	3b	3a	4a	2a	5a	1c
			1b	2b	2a	3b	4b	1a	5b	5a
				1a	2b	2b	3a	4a	2a	5b
Eject				**1b**			**3b**	**4b**	**1a**	**2a**
							2b	**3a**	**4a**	

hold both reference blocks and prefetched blocks. Tables 4 and 5 show that Belady's anomaly occurs even if the cache contains no reference blocks, only prefetch blocks. In these examples, a prefetch block is evicted from the cache as soon as it receives a user request. These examples represent the case when a cache is logically partitioned into a prefetch cache and a reference cache [7]. The temporal locality of reference (rereference hits) are handled by the reference cache while the spatial locality of reference (sequential prefetch hits) are handled by the prefetch cache. The replacement policy is labeled as FIFO not LRU in the examples presenting the prefetch-only cache. The reason is that the cache contains prefetch blocks that by its very definition, have never been referenced; the last access time of a prefetch block is the time it was prefetched and inserted into the cache. As a result, ordering the queue by recency of access is equivalent to ordering the blocks by time of insertion.

Note that the following rule is used in all the examples: when several blocks, s_{i+1}, s_{i+2},.... from a stream are prefetched at the same time, the blocks are inserted into the replacement queue such that s_{i+1} is closest to the non-eviction end. The reason for using this rule is that prefetch techniques PA/PM assume that s_{i+1} is the next stream block predicted to receive a hit. If s_{i+2} gets evicted before a hit, s_{i+2} could be prefetched again when s_{i+1} gets a hit. In the next section, we evaluate LRU's interaction with streams.

5 LRU and Streams

While prefetch techniques assume that the workload is composed of interleaved sequential streams, the LRU replacement policy is completely oblivious to streams. Every prefetch block is associated with a stream, but LRU ignores this information. Upon a hit, LRU moves the accessed block to the insertion end, but the rest of the blocks from this accessed block's stream retain their position in the LRU queue. (In a prefetch-only cache, the accessed prefetched block is removed from the cache.) If new blocks from this stream are prefetched, then these new blocks are inserted into the insertion end.

When block s_i from stream s is accessed, and block s_{i+1} is in the cache, LRU does not update the position of s_{i+1} in the replacement queue. If s_{i+1} is not in the cache, then the prefetch technique may prefetch s_{i+1} and insert s_{i+1} into the MRU end of the queue. Thus, a prefetch block is inserted into the MRU end when it is prefetched, and the block moves toward the LRU (eviction) end until it is either evicted upon reaching the LRU end or it is accessed by a user request (and becomes a reference block). Even though LRU orders reference blocks by access time, the prefetch blocks are ordered by time of in-

Table 4: Prefetch the next 2 contiguous blocks on arrival of each request

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

Prefetch cache size = 8, PA, FIFO								
hits				h1		h2		
workload	**1**	**2**	**3**	**1a**	**4**	**2a**	**5**	**2b**
Cache	1a	2a	3a	1c	4a	2c	5a	2d
	1b	2b	3b	3a	4b	4a	5b	5a
		1a	2a	3b	1c	4b	2c	5b
		1b	2b	2a	3a	1c	4a	2c
			1a	2b	3b	3a	4b	4a
			1b	1b	2a	3b	1c	4b
					2b	2b	3a	1c
					1b	1b	3b	3a
Eject							**2b**	**3b**
							1b	

Table 5: Prefetch the next 2 contiguous blocks on Miss, do not prefetch on hit

Prefetch cache size = 6, PM, FIFO												
hits								h1	h2			h3
string	**1**	**2**	**3**	**4**	**1a**	**2a**	**5**	**1b**	**2b**	**3a**	**4a**	**5a**
Cache	1a	2a	3a	4a	1b	2b	5a	5a	5a	3b	4b	4b
	1b	2b	3b	4b	1c	2c	5b	5b	5b	3c	4c	4c
		1a	2a	3a	4a	1b	2b	2b	2c	5a	3b	3b
		1b	2b	3b	4b	1c	2c	2c	1c	5b	3c	3c
			1a	2a	3a	4a	1b	1c		2c	5a	5b
			1b	2b	3b	4b	1c			1c	5b	
Eject				**1a**	**2a**	**3a**	**4a**				**2c**	
				1b	**2b**	**3b**	**4b**				**1c**	

Prefetch cache size = 7, PM, FIFO												
hits					h1	h2						
string	**1**	**2**	**3**	**4**	**1a**	**2a**	**5**	**1b**	**2b**	**3a**	**4a**	**5a**
Cache	1a	2a	3a	4a	4a	4a	5a	1c	2c	3b	4b	5b
	1b	2b	3b	4b	4b	4b	5b	1d	2d	3c	4c	5c
		1a	2a	3a	3a	3a	4a	5a	1c	2c	3b	4b
		1b	2b	3b	3b	3b	4b	5b	1d	2d	3c	4c
			1a	2a	2a	2b	3a	4a	5a	1c	2c	3b
			1b	2b	2b		3b	4b	5b	1d	2d	3c
				1a			2b	3a	4a	5a	1c	2c
Eject				**1b**				**3b**	**4b**	**5b**	**1d**	**2d**
								2b	**3a**	**4a**	**5a**	**1c**

sertion, not stream access time. From a stream perspective, when a stream gets an access, the LRU replacement queue is not updated.

Result 1 *The prefetch blocks in a LRU queue are ordered by block insertion time, not by stream access time.*

Gill et al. showed via simulation that Belady's anomaly occurs with PA and LRU [7]. They stated, without proof, that this problem could be fixed by moving all blocks in a stream to the MRU end upon a hit. This stream version of LRU had been mentioned in earlier paper [8, 12]. The focus of both papers was neither the replacement policy nor Belady's anomaly. Therefore, these papers did not provide details of the modified replacement policy. There was also no analysis on why the modified LRU policy does not exhibit Belady's anomaly. In this paper, we formalize this updated LRU policy and demonstrate that Belady's anomaly occurs even with this updated replacement policy.

5.1 StreamLRU

The stream version of LRU is as follows: upon a hit of block s_i from stream s, the hit block and all of this stream's blocks $s_j, j > i$ are moved to the MRU end of the replacement queue. The blocks are inserted into the replacement queue in order with block s_{i+1} farthest from the eviction end. Earlier blocks $s_j, j < i$, are not moved and retain their position in the replacement queue. The referenced block s_i is moved to the MRU end. If the cache contains only prefetch blocks, then the hit block s_i is evicted, but the blocks contiguous to this block are moved to the MRU end. We name this replacement policy *StreamLRU*.

Unlike LRU, StreamLRU incorporates the temporal locality of streams since the prefetch stream blocks are ordered by recency of access. StreamLRU incorporates the spatial locality of streams by not moving the earlier blocks of the stream on a hit; only the later blocks are moved, and re-inserted into the queue such that s_{i+1} is farthest from the eviction end. The sequential access pattern implies that the following contiguous blocks from a stream will be accessed, not previous blocks. In fact, file blocks that are read sequentially are rarely re-referenced. The streams (sequential access pattern) display temporal locality, but the blocks within a stream display spatial locality. LRUstream incorporates both the temporal locality of streams and the spatial locality of blocks within a stream by ensuring that the least recently accessed prefetch stream is evicted, with the later blocks from the stream evicted first.

Result 2 *The prefetch blocks in a StreamLRU queue are ordered by stream access time.*

The StreamLRU's queue is ordered by recency of access of both reference blocks and streams, implying that StreamLRU should be a stack algorithm. Therefore, one would expect that a caching system employing StreamLRU with prefetching would not exhibit Belady's anomaly. Surprisingly, this is not the case: Table 6 demonstrates the occurrence of Belady's anomaly with StreamLRU and the PM technique. The prefetch blocks are shown in italics in the cache. Note that when block 1b is accessed in the small cache, block 1a is not moved to the MRU end since 1a precedes 1b; StreamLRU moves blocks following the hit block.

In Table 7, we demonstrate, by a prefetch-only cache, that it is not the presence of reference blocks in the cache that causes Belady's anomaly. In Table 8 we demonstrate the occurrence of Belady's anomaly with the PA technique. Belady's anomaly occurs with all combinations of LRU/StreamLRU and PA/PM.

For a given workload, in an LRU cache without prefetching, the hit ratio with all caches sizes can be computed in a single pass by just running the workload trace on the largest cache [9, 12]. As a result of Belady's anomaly, it is impossible to predict how a caching system with prefetching would perform if the size of the cache is increased or decreased. It is also intriguing as to why Belady's anomaly occurs with StreamLRU, a replacement policy that orders streams by recency of access. In the next section, we analyze why Belady's anomaly occurs with prefetching and show how the anomaly can be removed.

Table 6: Prefetch the next block on miss. The workload reference string is **1a, 2a, 3a, 4a, ...**; the insertions of 1a and 2a are not shown due to space constraints

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

Reference + Prefetch cache size = 7, PM, StreamLRU								
hits			h1	h2				
workload	**3a**	**4a**	**1a**	**2b**	**5a**	**1b**	**2c**	**1a**
Cache	3a	4a	1a	2b	5a	1b	2c	1a
	3b	*4b*	4a	1a	*5b*	*1c*	*2d*	1b
	2a	3a	*4b*	4a	2b	5a	1b	*1c*
	2b	*3b*	3a	*4b*	1a	*5b*	*1c*	2c
	1a	2a	*3b*	3a	4a	2b	5a	*2d*
	1b	*2b*	2a	*3b*	*4b*	1a	*5b*	5a
		1a	*2b*	2a	3a	4a	2b	*5b*
Eject		***1b***			***3b***	***4b***	**1a**	***2b***
					2a	**3a**	**4a**	

Table 7: Prefetch the next 2 blocks on Miss. The workload reference string is **1, 2, 3, 4, ...**; the insertions of 1 and 2 are not shown due to space constraints

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

Prefetch cache size = 7, PM, StreamLRU										
hits			h1		h2				h3	
string	**3**	**4**	**1a**	**5**	**2a**	**1b**	**2b**	**3a**	**5a**	**5b**
Cache	3a	4a	4a	5a	5a	1c	2c	3b	3b	5c
	3b	4b	4b	5b	5b	1d	2d	3c	3c	5d
	2a	3a	3a	4a	4a	5a	1c	2c	2c	3b
	2b	3b	3b	4b	4b	5b	1d	2d	2d	3c
	1a	2a	2a	3a	3a	4a	5a	1c	1c	2c
	1b	2b	2b	3b	3b	4b	5b	1d	1d	2d
		1a		2a		3a	4a	5a		1c
Eject		**1b**		**2b**		**3b**	**4b**	**5b**		**1d**
							3a			

6 Analysis

The examples in Sections 4 and 5 show that Belady's anomaly can occur with all combinations of LRU/StreamLRU and PA/PM. In order to understand why the anomaly occurs and if it can be prevented, one has to understand the characteristics of a caching and prefetching system. It is well known that caching and prefetching systems are difficult to evaluate [3, 4, 5]. Part of this difficulty is that caching or prefetching, each in its own right, is complex. Also, the term "caching and prefetching" is somewhat misleading since caching encompasses three variables, namely the cache size, the cache workload, and the replacement policy. Therefore, caching and prefetching refers to a cache, its workload, and two distinct algorithms, namely the replacement policy and the prefetch technique.

For a given cache (size) and workload, the contents of the cache at any instant is determined by both the replacement policy and the prefetch technique. The actions of a replacement policy are influenced by the prefetch technique since the prefetch technique determines what blocks are loaded into the cache. Similarly, the actions

Table 8: On miss, prefetch 1 block. On hit of trigger block, prefetch 3 blocks; the trigger block is marked -; when trigger block ejected, trigger moves to previous block in the stream. The workload reference string is **1, 2, 3, 1a, 2a, 4, ...**; the insertions of 1 and 2 are not shown due to space constraints.

Prefetch cache size = 5, PA, StreamLRU											
hits		h1	h2		h3				h4		h5
string	**3**	**1a**	**2a**	**4**	**1b**	**5**	**6**	**7**	**1c**	**8**	**5a**
Cache	3a-	1b	2b	4a-	1c	5a-	6a-	7a-	1d-	8a-	5b
	2a-	1c-	2c-	2b	1d-	1c	5a-	6a-	7a-	1d-	5c-
	1a-	1d	2d	2c-	1e	1d-	1c	5a-	6a-	7a-	5d
		3a-	1b	2d	4a-	1e	1d-	1c	5a-	6a-	8a-
		2a	1c-	1b-	2b-	4a	1e	1d-		5a-	1d-
Eject			**1d**	**1c**	**2c**	**2b**	**4a**	**1e**			**7a**
			3a		**2d**						**6a**
											5a

Prefetch cache size = 6, PA, StreamLRU											
hits		h1	h2		h3				h4		
string	**3**	**1a**	**2a**	**4**	**1b**	**5**	**6**	**7**	**1c**	**8**	**5a**
Cache	3a-	1b	2b	4a-	1c-	5a-	6a-	7a-	1d	8a-	5b-
	2a-	1c-	2c-	2b	4a-	1c-	5a-	6a-	1e-	1d	8a-
	1a-	1d	2d	2c-	2b	4a-	1c-	5a-	1f	1e-	1d
		3a-	1b	2d	2c-	2b	4a-	1c-	7a-	1f	1e-
		2a-	1c-	1b	2d	2c-	2b	4a-	6a-	7a-	1f
			1d	1c-		2d	2c-	2b-	5a-	6a-	7a-
Eject				**1d**			**2d**	**2c**	**4a**	**5a**	**6a**
									2b		

of a prefetch technique are influenced by the replacement policy, since the replacement policy determines what blocks are evicted. In order to understand a caching and prefetching system, one needs to understand the combined impact of two distinct, but dependent algorithms.

6.1 Assumptions & Notation

The examples in earlier sections demonstrate that Belady's anomaly can occur in caches with prefetching, regardless of the presence or absence of reference blocks. In the examples, it is the presence of prefetch blocks that causes the anomaly. In order to focus on why prefetching causes Belady's anomaly without the distraction of reference blocks, we analyze caches that contain only prefetch blocks.

Definition 3 *A* **prefetch cache** *partition stores prefetch blocks only; once a prefetch block gets a hit, it is evicted from the prefetch cache.*

The evicted hit block can be moved to the reference cache. Therefore, all re-references in the workload are handled by the reference cache. In order not to conflate the impact of re-referencing (temporal locality) with prefetching (spatial locality), it is assumed that the input workload contains no re-references. PA and PM are effective only if the workload displays sequential locality of reference. Therefore, it is assumed that the workload follows Assumption 3 listed in Section 3. In addition, it is assumed that prefetch blocks are loaded into the cache instantaneously, as soon as the prefetch request is generated.

Notation: The small prefetch cache has S blocks and the large prefetch cache has L blocks, where $\mathsf{S} < \mathsf{L}$. The notation for cache workload is specified in Section 2. When there are several streams, a particular stream is referred to as is or is_j, where $i, j > 0$. The blocks of stream is are $< is_1, is_2, is_3, ... >$. The blocks of stream is_j are $< is_{j+1}, is_{j+2}, is_{j+3}, ... >$.

6.2 Definitions

The prefetch technique is initiated upon arrival of a user request. The determination as to what, if any, blocks to prefetch from the user request's stream is based on the contents of the cache. For example, consider a PA technique that prefetches 2 blocks on a miss or on a hit of a trigger block. The blocks to be prefetched are contiguous to the missed block or to the trigger block. Suppose a user request for s_i arrives and hits in the cache. If the hit s_i is a non-trigger block, then no prefetch request is generated. If the hit s_i is a trigger block, but block s_{i+1} is already in the cache, then a prefetch request for only s_{i+2} is generated. Thus, the number of blocks prefetched by the technique varies.

At time t, suppose user request s_i arrives at the cache. At time $t+$, depending on the prefetch technique, the cache contains $D \geq 0$ blocks from stream s. If $D = 0$, then the prefetch cache contains no blocks from stream s; if $D > 0$, then the prefetch cache contains $s_{i+1}, ..., s_{i+D}$. Some, or all, of these blocks may have been prefetched before time t. If the cache contained $d \leq D$ of these blocks prior to time t, then $D - d$ blocks are prefetched upon arrival of s_i.

Definition 4 *The* **stream degree***, $D \geq 0$, for a prefetch technique is the number of blocks from stream s that are in the prefetch cache immediately after an user request for a block from stream s arrives.*

Depending on the prefetch technique, the stream degree may be a constant or a variable. Note that the stream degree differs from the *prefetch degree*, which is the number of blocks from stream s that are prefetched when the user request s_i arrives. For example, in Table 7, the stream degree is 0, 1 or 2, while the prefetch degree is 0 or 2. Table 4 is an example of a prefetch technique where stream degree $D = 2$, while the prefetch degree is 1 or 2.

A prefetch technique predicts the future cache workload by identifying streams in the workload. At a given instant, not all blocks in a stream are of relevance. At time t, suppose s_i is the last block from stream s to have received a user request. That is, s_i is the largest block number accessed from the stream.

Definition 5 *The* **access time** *of stream s is the time at which s_i received a user request.*

The access time is updated regardless of whether the user request for s_i hits or misses in the cache.

When several blocks are prefetched at the same time, the blocks are inserted into the cache such that s_{i+1} is farthest from the eviction end. The next block from stream s expected to receive a user request is s_{i+1}. Thus, s_{i+1} is the only relevant stream block from a performance perspective.

Definition 6 *For each stream s, the* **relevant** *block, s_{i+1}, is the next block from this stream expected to receive a user request. Stream s gets a hit if s_{i+1} gets a hit, else s gets a miss.*

Definition 7 *The* **insertion time** *of stream s is the time at which block s_{i+1} is loaded into the cache.*

Definition 8 *The* **eviction time** *of stream s is either (1) when prefetch block s_{i+1} is evicted from the cache before a hit, or (2) when prefetch block s_i gets a hit, but s_{i+1} is not in the cache.*

At time t, even if earlier or later blocks from this stream are present in the cache, the stream is evicted since the next request for this stream will result in a miss. Note that a stream may be evicted by the replacement policy (point 1 in Definition 8) or by the prefetch technique (point 2 in Definition 8). Since cache size is limited, streams may be evicted and then reinserted into the cache multiple times. The above definitions capture the dynamic nature of streams.

6.3 Stack Property

An algorithm is said to have the the stack property if, for all workloads, it ensures that the set of blocks in a cache of size C is a subset of the set of blocks in a cache of size C+1 [9]. Such an algorithm is referred to as a stack algorithm. A fundamental result is [9]:

Stack algorithm $\Rightarrow$ no Belady's anomaly.

Therefore,

Belady's anomaly $\Rightarrow$ not stack algorithm.

Prefetching is all about streams, and the relevant block of a stream determines whether the stream gets a hit or a miss. The eviction of the relevant block results in the eviction of the corresponding stream from the cache.

Definition 9 *A* **prefetch stack algorithm** *ensures that the set of relevant stream blocks in a small cache is a subset of the relevant stream blocks in a larger cache.*

The examples in Sections 4 and 5 demonstrate Belady's anomaly, thereby proving that the corresponding prefetch/replacement algorithms are not stack algorithms. Note that stack algorithm is a necessary condition for the non-existence of Belady's anomaly in a caching system. It is not known whether "not stack algorithm" is a sufficient condition to show that Belady's anomaly will occur.

The examples cover combinations of LRU/PA, LRU/PM, StreamLRU/PA, and StreamLRU/PM. While the examples demonstrate that these combinations are not stack algorithms, it is not clear what destroys the stack property - is it the prefetch technique, the replacement policy, or the combined impact of both? Mattson et al. have shown that a replacement policy whose queue ordering is not dependent on the queue size (*i.e.,* cache size) is a stack algorithm. Consequently, both LRU and StreamLRU are stack algorithms. However, when either policy is combined with prefetching, the caching system does not have the stack property. The rest of this section explains why the stack property is destroyed when prefetching is implemented in a cache. The prefetch technique can be either PA or PM, and the replacement policy is either LRU or StreamLRU.

Result 3 *PA that prefetches exactly one block and LRU/StreamLRU is a prefetch stack algorithm.*

Proof: Since the prefetch technique prefetches only 1 block, StreamLRU is equivalent to LRU.
When user request s_i arrives, regardless of a hit/miss, the block s_{i+1} is prefetched and loaded into the MRU end of the replacement queue. Therefore, the queue is ordered by recency of stream access.
Exactly one block per stream is loaded into the cache. As a result, a cache of size C contains the last C distinct streams, while a cache of size C+1 contains the last C distinct streams, in addition to the last distinct stream accessed prior to that. Hence, the set of blocks in a cache of size C is a subset of the set of blocks in a cache of size C+1.
□

A prefetch technique that prefetches exactly one block is referred to as *One Block Lookahead (OBL)* technique [10]. Thus, OBL PA and LRU/StreamLRU will never exhibit Belady's anomaly.

The next result proves that PM with either LRU/StreamLRU is not a stack algorithm. The result is proved by showing that a small cache is not a subset of a large cache for a specific workload. Intuitively, the reason for the violation of the stack property is that upon hit of the last prefetched block from a stream, the stream is evicted from the cache. Regardless of the replacement policy, PM evicts streams from caches. Construct a workload where a stream is

first evicted from the small cache but exists in the large cache. When the next request for this stream arrives, it misses in the small cache resulting in the reinsertion of the stream; the request hits in the large cache resulting in no prefetch action. Consequently, the small cache contains more of this stream's prefetch blocks than the large cache violating the stack property. Thus, the small cache is not a subset of the large cache.

Result 4 *PM and LRU/StreamLRU is not a prefetch stack algorithm.*

Proof: Consider PM, where on a miss of block s_i, a fixed $P > 0$ blocks contiguous to s_i are loaded. Therefore, $s_{i+1}, ...s_{i+P}$ prefetch blocks are prefetched.
Suppose $\mathsf{L} \geq \mathsf{S} + P$, and $\mathsf{S} = (n-1) \times P$.
Initially, the small cache and large cache are empty.
Consider the following cache workload: < 1s, 2s, 3s, ..., (n-1)s, ns, $1s_1$, $1s_2$, ..., $1s_P$, $1s_{P+1}$, ...>, where each is represents a distinct stream.
The arrival of each block is results in a prefetch of P blocks, $is_1, is_2, ...is_P$.
Block ns is the first workload block that results in an eviction of stream $1s$, from the small prefetch cache. Stream $1s$ is not evicted from the large cache. All blocks of stream $1s$ are evicted from the small cache, but none of the blocks of stream $1s$ are evicted from the large cache.
The next workload request is for $1s_1$. This will result in a hit in the large cache, but no prefetch action. In the small cache, the request for $1s_1$ will miss, and blocks $1s_2, 1s_3, ...1s_{P+1}$ are prefetched. Note that the large cache contains blocks $1s_2, 1s_3, ...1s_P$. Therefore, the stack property is violated.
The next workload requests are for $1s_2$, ..., $1s_P$. These requests hit in both caches. After the hit of block $1s_p$, block $1s_{P+1}$ is in the small cache, but it is not in the large cache.
Stream $1s$ is evicted from the large cache, but it is in the small cache. Hence, the prefetch stack property is violated.
□
The PM technique is widely implemented in caches. In PM, every time there is a miss in the small cache, but a hit in the large cache, the stack property gets violated. This can lead to the violation of the stream stack property, a sufficient condition for Belady's anomaly to occur.

Result 5 *LRU and any prefetch technique, with the exception of OBL PA, is not a prefetch stack algorithm.*

Proof: Result 1 states that prefetch blocks are ordered by time of insertion in the LRU queue. The position of a prefetch block is not updated when its corresponding stream is accessed. From a stream perspective, the LRU queue is ordered by stream insertion time, not stream access time. A replacement policy that orders the replacement queue by insertion time is not a stack algorithm [9]. The LRU replacement queue orders relevant blocks by insertion time, not stream access time. Hence, the LRU replacement queue is not a prefetch stack algorithm.
□

A replacement queue that orders prefetch blocks by insertion time, not stream access time, is not a stack algorithm. A prefetch technique that results in eviction of streams is not a stack algorithm. Thus, PM-LRU, PM-StreamLRU and LRU-PA are not stack algorithms, and several examples in Sections 4 and 5 have demonstrated that Belady's anomaly occurs with these algorithms. This leaves the PA with StreamLRU - PA does not evict streams, and StreamLRU orders its queue by recency of stream access - as the lone algorithm that could have the stack property. Table 8, however, demonstrates that Belady's anomaly occurs in the PA-StreamLRU cache. It is a conundrum as to why this algorithm does not possess the prefetch stack property.

Instead of explaining why the anomaly occurs for the PA-StreamLRU algorithm presented in Table 8, we first prove that a certain class of the PA-StreamLRU algorithm has the stack property.

Result 6 *StreamLRU and PA, where the stream degree D is fixed, is a prefetch stack algorithm.*

Proof: Let the stream degree $D = d > 0$. When a user request s_i arrives at the cache, PA ensures that s_{i+1}, ..., s_{i+d} are in the cache.
If $d = 1$, then the algorithm is StreamLRU with OBL PA, which is a stack algorithm (Result 3).
Suppose $d > 1$. When a workload request s_i arrives, the prefetch technique ensures that the next d blocks from stream s are in the cache.
The StreamLRU policy ensures that the d blocks from stream s are at the MRU end of the replacement queue, with s_{i+1} farthest from the LRU end.
A stack of size C holds the most recently accessed $\lceil \frac{\mathsf{C}}{d} \rceil$ distinct streams.
Hence, for all workloads, a cache of size S will be a subset of a cache of size L when $\mathsf{S} \leq \mathsf{L}$.
□
With regard to streamLRU, Result 6 identifies the essential property - a fixed stream degree - required of a prefetch technique in order for the caching system to not exhibit Belady's anomaly. When D is variable, it is possible to construct a workload where the small cache holds more streams that the larger cache as a result of having fewer blocks per stream.

Corollary 1 *When the stream degree D is a variable, PA and StreamLRU is not a prefetch stack algorithm.*

6.4 Discussion

The results show that the common prefetching techniques, PM and trigger PA, are not prefetch stack algorithms. The fixed stream degree PA is a stack algorithm, but from a performance perspective it is generally inferior to PM and trigger PA (which is why it is not often implemented in real systems). Here, we analyze whether the fixed stream degree condition can be relaxed and still allow for prefetch caches to have the stack property.

The essential property required of a non-prefetch cache (*i.e.,* reference cache) to be a stack algorithm is that the replacement queue ordering of blocks must be independent of the queue size. To avoid ambiguity, we specify the meaning of replacement queue ordering: it refers to the state of the queue immediately after a user request arrives and is processed. In the reference cache, every user request, whether hit or miss, is in the replacement queue after the request is processed. The state of the queue refers to the blocks, both the cached blocks and the new user request block, and the relative position of the blocks in the queue.

In a prefetch cache, similar to a reference cache, replacement queue ordering refers to the state of the queue immediately after a user request arrives and is processed. The user request, whether hit or miss, is not in the prefetch cache after processing, but the request's stream blocks may be present in the cache. Some of these blocks may have been present prior to the arrival of the user request, while some may have been newly prefetched. Similar to a reference cache, the state of the replacement queue refers to the prefetch blocks and their position in the queue. The prefetch blocks in the cache refer to the blocks present prior to arrival of the user request and the new blocks prefetched upon arrival of the user request.

Proposition 1 *A prefetch cache system satisfies the prefetch stack property if the replacement queue ordering of stream blocks is independent of the cache size.*

The reasoning behind Proposition 1 is presented below:

StreamLRU and fixed stream degree PA: With a fixed stream degree $D = d \geq 1$, when request s_i arrives, the blocks $< s_{i+1}, ..., s_{i+d} >$ are moved to the MRU end of the queue (in reverse order of the sequence). This action is completely independent of the cache size. Therefore, the replacement queue ordering of stream blocks is independent of cache size.

LRU and fixed stream degree PA: With a fixed stream degree $D = d > 1$, when request s_i arrives, only the newly prefetched stream blocks are moved to the MRU end; the rest retain their position. Therefore, the replacement queue ordering of stream blocks is dependent on the cache size.

StreamLRU and a variable stream degree PA, say, trigger PA: When request s_i arrives, the value of D is dependent on whether s_i misses or hits in the cache; if s_i hits, it matters whether the hit block is a trigger block. Since a hit or miss of s_i is dependent on the cache size, the value of D is dependent on the cache size. Thus, the blocks $< s_{i+1}, ..., s_{i+d} >, d \geq 0$ moved to the MRU end, vary depending on the cache size. Hence, the replacement queue ordering of stream blocks is dependent on the cache size.

PM and StreamLRU: When request s_i arrives, the value of D is dependent on whether s_i hits or misses in the cache. Thus, the blocks $< s_{i+1}, ..., s_{i+d} >, d \geq 0$ moved to the MRU end, vary depending on the cache size.

The essential property required of a stack algorithm is the same in a reference cache and a prefetch cache. In a reference cache, queue order implicitly assumes that the blocks in the cache are the workload blocks, so order explicitly refers to the position of blocks. In a prefetch cache, queue order explicitly refers to the stream blocks and their position in the queue.

The question is whether Proposition 1 can be relaxed. The prefetch stack property is weaker than the stack property. The prefetch stack property only requires that the set of relevant blocks in the small cache be a subset of the relevant blocks in the large cache.

stack property $\Rightarrow$ prefetch stack property, but
prefetch stack property $\not\Rightarrow$ stack property.

Proposition 2 *A prefetch cache system satisfies the prefetch stack property if the replacement queue ordering of relevant stream blocks is independent of the cache size.*

The results in this paper continue to hold with Proposition 2: fixed degree PA and StreamLRU is the only algorithm, from amongst those analyzed in this paper, where the queue ordering of relevant blocks is independent of the cache size. Proposition 2 implies that regardless of the replacement policy, PM is not a stack algorithm since it evicts streams. PM is a popular prefetch technique that is widely implemented, but this paper proves that PM is inherently not a prefetch stack algorithm. Trigger PA is another popular prefetch technique. Trigger PA and StreamLRU is not a prefetch stack algorithm and can exhibit Belady's anomaly as demonstrated in Table 8. The key question is whether trigger PA and another replacement policy could satisfy Proposition 2.

7 Conclusion

Almost all file system caches perform prefetching which interacts with the cache replacement policies. The ensuing cache system is hard to evaluate since the performance varies erratically [3]. This paper demonstrates

that the unpredictable performance may be a consequence of the prefetch cache system not exhibiting the stack property. The introduction of prefetching almost always causes the violation of the stack property, which in some case leads to Belady's anomaly. It is difficult to isolate the reasons for the violation since the actions of the prefetch technique and replacement policy are entwined.

This paper is successful, to a degree, in (1) explaining the reasons for stack property violation, and (2) proposing remedies. A side bar contribution of the analysis is that it shows how a prefetch technique can be made independent of the replacement policy (by using a fixed stream degree). This may be useful when evaluating performances of replacement policies for prefetch caches.

References

[1] 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.

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

[3] 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.

[4] CAO, P., FELTEN, E. W., KARLIN, A. R., AND LI, K. A study of integrated prefetching and caching strategies. In *SIGMETRICS '95/PERFORMANCE '95: Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems* (1995), ACM Press, pp. 188–197.

[5] 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.

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

[7] 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.

[8] LAM, C.-Y., AND MADNICK, S. E. Propeties of storage hierarchy systems with multiple page sizes and redundant data. *ACM Trans. Database Syst. 4* (September 1979), 345–367.

[9] MATTSON, R., GECSEI, J., SLUTZ, D., AND TRAIGER, I. Evaluation techniques for storage hierarchies. *IBM Systems Journal 9*, 2 (1970), 78 –117.

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

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

[12] 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.