

AMP: Adaptive Multi-stream Prefetching in a Shared Cache

Binny S. Gill and Luis Angel D. Bathen

IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120

Emails: binnyg@us.ibm.com, lbathen@uci.edu

Abstract—Prefetching is a widely used technique in modern data storage systems. We study the most widely used class of prefetching algorithms known as *sequential prefetching*. There are two problems that plague the state-of-the-art sequential prefetching algorithms: (i) *cache pollution*, which occurs when prefetched data replaces more useful prefetched or demand-paged data, and (ii) *prefetch wastage*, which happens when prefetched data is evicted from the cache before it can be used.

A sequential prefetching algorithm can have a fixed or adaptive degree of prefetch and can be either synchronous (when it can prefetch only on a miss), or asynchronous (when it can also prefetch on a hit). To capture these distinctions we define four classes of prefetching algorithms: Fixed Synchronous (FS), Fixed Asynchronous (FA), Adaptive Synchronous (AS), and Adaptive Asynchronous (AA). We find that the relatively unexplored class of AA algorithms is in fact the most promising for sequential prefetching. We provide a first formal analysis of the criteria necessary for optimal throughput when using an AA algorithm in a cache shared by multiple steady sequential streams. We then provide a simple implementation called AMP, which adapts accordingly leading to near optimal performance for any kind of sequential workload and cache size.

Our experimental set-up consisted of an IBM xSeries 345 dual processor server running Linux using five SCSI disks. We observe that AMP convincingly outperforms all the contending members of the FA, FS, and AS classes for any number of streams, and over all cache sizes. As anecdotal evidence, in an experiment with 100 concurrent sequential streams and varying cache sizes, AMP beats the FA, FS, and AS algorithms by 29-172%, 12-24%, and 21-210% respectively while outperforming OBL by a factor of 8. Even for complex workloads like SPC1-Read, AMP is consistently the best performing algorithm. For the SPC2 Video-on-Demand workload, AMP can sustain at least 25% more streams than the next best algorithm. Finally, for a workload consisting of short sequences, where optimality is more elusive, AMP is able to outperform all the other contenders in overall performance.

I. INTRODUCTION

Over the last several decades, we have witnessed remarkable improvements in the information processing capabilities of computing systems. A large number of data storage technologies have also been developed with diverse speeds, capacities, reliability and affordability characteristics. We often find that cost considerations force us to design systems with a data storage component which runs significantly slower than the processing unit. To bridge this gap between the data supplier and

the data consumer, faster data caches are placed between the two. Since caches are expensive, they can typically keep only a subset of the entire data-set. Consequently, it is extremely important to manage the cache wisely in order to maximize its performance. The cornerstone of read cache management is to keep recently requested data in the cache in the hope that such data will be requested again in the near future. Data is placed in the cache only when requested by the consumer (*demand-paging*). Another, and rather competing method, is to fetch into the cache data that is predicted to be requested in the near future (*prefetching*).

A. Where is Prefetching Applied

The technique of prefetching dates as far back as the mid-sixties when multiple words were prefetched in processors in the form of a cache line. It was soon realized that increasing the size of the cache line can decrease performance due to false sharing. So, numerous hardware-initiated prefetching techniques were introduced in both uniprocessor and multiprocessor architectures [1], [2], [3]. Subsequently, software-initiated methods for prefetching were introduced where applications disclosed access patterns to the hardware or controlled prefetching directly [4], [5]. For other applications, compiler techniques were used to predict access patterns and insert fetch requests in the compiled executables [6], [7]. Compiler-assisted prefetching was also extended for pointer-based accesses [8], [9], [10].

Today prefetching is ubiquitously applied in web servers and clients [11], databases [12], file servers [13], [14], on-disk caches [15], and multimedia servers [16].

B. When is Prefetching Useful

The goal of prefetching is to make data available in the cache before the data consumer places its request, thereby masking the latency of the slower data source below the cache. However, prefetching is not without cost. It requires (i) cache space to keep the prefetched data; (ii) network bandwidth to transfer the data to the cache; (iii) data source bandwidth to read the data; and (iv) processing power to carry out the prefetch. If the prefetched data is not subsequently used by the data consumer, the extra cost of prefetching normally reduces performance. Only in over-provisioned systems,

can prefetching with low predictive accuracy improve performance. However, the data cache is obviously under-provisioned as it can keep only a subset of the data-set. The prefetched data typically shares the cache space with demand-paged data. Therefore, the utility of the prefetched data should not be lower than the utility of the demand-paged data it replaces. To maximize the performance, the marginal utility of both kinds of data should be equalized [17]. Since the utility of prefetched data that is not subsequently used is zero, it is extremely important to prefetch judiciously, keeping the number of wasted prefetches to a minimum. Furthermore, any prefetching algorithm needs to be able to predict accesses sufficiently in advance to allow for the time it takes to prefetch the data. As a rule of thumb, prefetching is useful when the long-term prediction accuracy of access patterns is high.

C. What to Prefetch

The most common prefetching approach is to perform sequential readahead. The simplest form is One Block Lookahead (OBL), where we prefetch one block beyond the requested block [18]. OBL can be of three types: (i) *always prefetch* – prefetch the next block on each reference, (ii) *prefetch on miss* – prefetch the next block only on a miss, (iii) *tagged prefetch* – prefetch the next block only if the referenced block is accessed for the first time. *P-Block Lookahead* extends the idea of OBL by prefetching P blocks instead of one, where P is also referred to as the *degree of prefetch*. Dahlgren [19] proposed a version of the P-Block Lookahead algorithm which dynamically adapts the degree of prefetch for the workload. Tcheun [20] suggested a per stream scheme which selects the appropriate degree of prefetch on each miss based on a prefetch degree selector (PDS) table. For the case where cache is abundant, *Infinite-Block Lookahead* has also been studied [21].

Stride-based prefetching has also been studied mainly for processor caches where strides are detected based on information provided by the application [22], a lookahead into the instruction stream [23], or a reference prediction table indexed by the program counter [24]. Dahlgren [25] found that sequential prefetching is a better choice because most strides lie within the block size and it can also exploit locality.

History-based prefetching has been proposed in various forms. Grimsrud [26] uses a history-based table to predict the next pages to prefetch. Prefetching using Markov predictors has been studied in [27], wherein multiple memory predictions are prefetched at the same time. Data compression techniques have also been applied to predict future access patterns [12]. Vitter [28] provided an optimal (in terms of the miss ratio) prefetching technique based on the Lempel-Ziv algo-

rithm. Lei [29] suggested a file prefetching technique based on historical access correlations maintained in the form of access trees.

The fact is, most commercial data storage systems use very simple prefetching schemes like sequential prefetching. This is because only sequential prefetching can achieve a high long-term predictive accuracy in data servers. Strides that cross page or track boundaries are uncommon in workloads and therefore not worth implementing. History-based prefetching suffers from low predictive accuracy and the associated cost of the extra reads on an already bottlenecked I/O system. The data storage system cannot use most hardware-initiated or software-initiated prefetching techniques as the applications typically run on external hardware. Further, offline algorithms [30], [31], [32], [33] are not applicable as they require knowledge of future data accesses.

D. The Problem of Cache Pollution

In the context of prefetching, *cache pollution* is said to occur when prefetched data replaces more useful data (demand-paged or prefetched) from the cache. There have been attempts to reduce cache pollution by restricting the amount of cache the prefetched data can occupy [34], or via software hints [35]. The SARC algorithm [17] provides an adaptive and autonomous solution to limit this problem by allocating cache space so as to equalize the marginal utility of the demand paged and prefetched data. However, we are not aware of any prior online solution for minimizing cache pollution that occurs when new prefetched data replaces more useful prefetched data from the cache.

E. The Problem of Wasted Prefetches

In data storage systems, the disks are typically the bottleneck. If pages are prefetched speculatively and are not subsequently used, then not only does this cause cache pollution and increase in the backend bandwidth usage, but more importantly, it causes additional I/O load on the disks. This additional load can lead to degradation in performance, defeating the purpose of prefetching. This is the reason why most history-based prefetching schemes which do not have high prediction accuracy are not used in commercial systems.

F. Our Contributions

A prefetching algorithm can have a fixed or adaptive degree of prefetch and can be either asynchronous (when it can prefetch on a hit) or synchronous (when it can prefetch only on a miss). This naturally leads to four classes which we call Fixed Synchronous (FS), Fixed Asynchronous (FA), Adaptive Synchronous (AS), and Adaptive Asynchronous (AA).

Although sequential and non-sequential data typically occupy the same cache, it is worthwhile to examine the prefetched data alone as most known prefetching algorithms suffer from cache pollution and prefetch wastage, and thus, can be improved.

We examine the case where an LRU (Least Recently Used) cache houses prefetched data for multiple concurrent sequential streams. We provide a theoretical analysis and prove the sufficient conditions for optimal online cache management for steady-state sequential streams. We also provide a simple implementation called AMP, the first member of the AA class which optimally adapts both the degree of prefetch and the timing thereof according to the workload and cache size constraints.

With a theoretically optimal design, AMP minimizes prefetch wastage and cache pollution within the prefetched data while maximizing the aggregate throughput achieved by the sequential streams. To demonstrate the effectiveness of AMP, we compare it with 9 other prefetching algorithms including the best representatives from the FA, FS, and AS classes, over a wide range of cache sizes, request rates, request sizes, number of concurrent streams, and workloads.

We observe that AMP convincingly outperforms all the FA, FS, and AS algorithms for any number of streams, and over all cache sizes. In an experiment with a 100 concurrent sequential streams and varying cache sizes, AMP beats the FA, FS, and AS algorithms by 29-172%, 12-24%, and 21-210% respectively while outperforming no prefetching and OBL by a factor of 8. AMP is consistently the best performing algorithm in both the small cache and large cache scenarios, even for complex workloads like SPC1-Read. For SPC2 Video-on-Demand workload, AMP can support at least 25% more streams than the next best algorithm. For streams with short sequences, as well, for which optimality is more elusive, AMP surpasses all the other contenders in its overall performance.

G. Outline of the Paper

In Section II, we suggest a useful classification of sequential prefetching algorithms and examine each in detail. In Section III, we provide a formal analysis and proof for the conditions necessary for optimal sequential prefetching. We also provide an implementation called AMP. In Section IV, we describe the workloads used in this paper. In Section V, we delineate the experimental setup used for our experiments. In Section VI, we present the experimental results and conclude with our findings in Section VII.

II. Sequential Prefetching

A. Rules of engagement

The cost of caching an entity is equal to the size of the entity multiplied by the amount of time for which it is present in the cache. The benefit of caching an entity, on the other hand, is the number of hits it produces. We define two self-evident rules that any prefetching algorithm should follow:

- *Avoid Wastage Rule*: Do not prefetch any page that will be evicted before it is requested by the workload.
- *Avoid Cache Pollution Rule*: Do not prefetch any page that will evict other prefetched pages without providing any net gain in performance.

B. Synchronous Vs. Asynchronous Prefetching

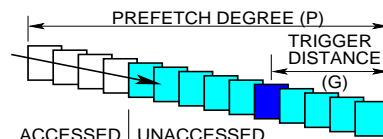


Fig. 1. Asynchronous Prefetching

There are two kinds of prefetch requests: (i) *synchronous* prefetch, and (ii) *asynchronous* prefetch. A *synchronous* prefetch is when on a *miss* on page x , we prefetch p extra pages beyond page x . It merely extends the extent of the client's read request to include more pages. On the other hand, an *asynchronous* prefetch is when on a cache *hit* on a page x , we create a new read request to prefetch p pages beyond those already in the cache. In each set of p prefetched pages, a *trigger* page is identified at a *trigger distance* of g from the end of the prefetched set of pages (Figure 1). When $g = 0$, the trigger is set on the last page of the prefetched set. When a *trigger* page is hit, an asynchronous prefetch is requested for the next set of p sequential pages. Unlike synchronous prefetching, asynchronous prefetching enables us to always stay ahead of sequential read requests and for suitable values of p and g , never incur a read miss after the initial miss for a sequential stream [17]. Asynchronous prefetching is always used in conjunction with some form of synchronous prefetching to prefetch the initial set of pages.

Notice that asynchronous prefetching creates new read requests on its own and, therefore, in cases where prefetches are wasted, asynchronous prefetching will have more disk seeks on the backend for the same workload than synchronous prefetching. However, for larger prefetches on data striped across disks, even synchronous prefetches will result in new read requests.

As a guideline, asynchronous prefetching should be avoided in cases where prefetch wastage is high.

C. A Classification of Prefetching Algorithms

Sequential prefetching is the most promising and widely deployed prefetching technique for data servers. It has a high predictive accuracy and is extremely simple to implement. Simple methods are used to isolate the sequential components of workloads [17], upon which prefetching is applied. We can classify the known sequential prefetching techniques as follows:

- *Fixed Synchronous (FS) prefetching*: The simplest form of sequential prefetching is where we prefetch the next page on a miss (OBL [18]). Another variant is where a fixed number of pages (p) are prefetched on every miss.
- *Adaptive Synchronous (AS) prefetching*: This is a popular form of sequential prefetching where the number of pages prefetched on every miss (p) is gradually increased as the length of the sequence referenced becomes longer. The degree of prefetch, p starts with 1 and is either linearly incremented on every miss [20] (*Linear-AS*) or exponentially incremented (*Exp-AS*). Usually, there is a predefined upper limit for incrementing p . Although *Exp-AS* adapts faster than the *Linear-AS* method, it is prone to more wastage in workloads with many short sequences or when cache space is limited.
- *Fixed Asynchronous (FA) prefetching*: In this class, a hit on a trigger page causes a prefetch. Tagged prefetching [18], a variant of OBL, is the earliest example of asynchronous prefetching where the degree of prefetch was 1 and the trigger distance was 0. Subsequently, the idea has been extended to any pair of fixed values of p and g [17]. Unlike synchronous prefetching methods, this class of algorithms can achieve zero misses for a workload when the chosen values of p and g are adequate. However, since the algorithm is hand-tuned and not adaptive, it does not work well for all workloads.
- *Adaptive Asynchronous (AA) prefetching*: To the best of our knowledge, there is no published work that dynamically adapts both the degree of prefetch (p) and the trigger distance (g). This is the most promising class of prefetching algorithms. The only algorithm that comes close is the one proposed by Dalhgren [19] where the degree of prefetch is the same for all streams and every page is a trigger page. It is not truly applicable in the context of data servers because it is wasteful. As each page is a trigger page it inefficiently prefetches one page at a time for sequential streams. It also *requires* some amount of prefetch wastage to adapt p which is blindly applied for all sequential streams.

D. Interaction between demand-paged and prefetched data

Since demand-paged data, prefetched data, and sometimes modified data, share the same cache in most data server architectures, we normally would require a way to divide the cache between the various types, and manage each portion optimally, so as to maximize the overall performance of the system. While a large number of demand-paging cache replacement algorithms have been devised (for example, LRU, CLOCK, FBR, 2Q, LRFU, LIRS, MQ, and ARC), surprisingly, and to the best of our knowledge, there has been no research towards an online optimal cache replacement policy for prefetched data.

In this paper, we provide this missing link and present a provably optimal algorithm for multiple sequential streams sharing a cache and a very simple practical implementation thereof. We believe that much of the work on understanding the interactions between various types of cached data ([4], [17], [30], [33]) will benefit from and incorporate our algorithm and analysis.

III. AMP

A. Replacement Policy for Prefetched data

The most widely used data structure for cache replacement policy is LRU, mainly because of its simplicity. This policy leverages temporal locality in the workload to improve cache hit ratios. Even within sequentially prefetched data, it is possible to have non-sequential accesses that exhibit temporal locality. This has encouraged most commercial systems to use the LRU data structure and replacement policy even for prefetched data rather than simply evicting prefetched data immediately after use. In this paper, we improve this LRU policy by making it aware of the difference between prefetched and demand-paged data. A prefetched page is moved to the most recently used (MRU) position only on repeated access and not on the first access.

B. Theoretical Analysis: Optimality Criteria

In this section we theoretically analyze the case when an LRU cache is shared by prefetched data for multiple steady sequential streams. We assume an AA algorithm operating with a cache of size C . L is defined as the average life of a page in the cache. Each stream has a maximum request rate, r , which is achieved when all read requests are hits in the cache. The aggregate throughput, B , is the sum of individual stream throughputs ($B = \sum_i b_i$).

Observation III.1 $p/t(p)$ is a monotonically non-decreasing function, where $t(p)$ is the average time to prefetch p pages.

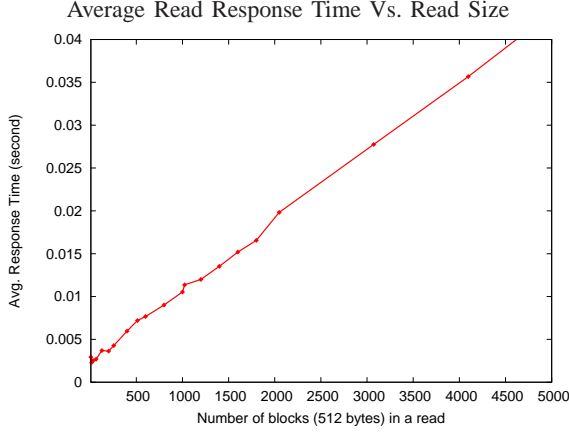


Fig. 2. Starting at 3 ms, the average response time grows linearly with the read size. We observed similar behavior for RAID-5 implying that the optimality criteria in this paper apply to RAID as well.

Proof: From Figure 2 we observe that $t(p)$ is of the form $kp + c$. Hence, $p/t(p) = \frac{p}{kp+c}$, and its slope

$$\frac{db}{dp} = \frac{c}{(kp+c)^2}$$

is positive since c is positive. ■

Definition III.1 A stream is said to be satisfied at (p, g) , if it experiences no misses for the given p and g .

Lemma III.1 A stream is satisfied at (p, g) iff $t(p) \leq (g+1)/r$.

Proof: By definition, if a stream is satisfied at (p, g) then it experiences no misses in the steady state. That implies that the prefetch issued at the trigger distance g completes before the $g+1$ pages are read by the client. Therefore, $t(p) \leq (g+1)/r$, where r is the request rate of the stream. The reverse is also true. If the time it takes to prefetch p pages is not more than the time it takes the client to read the g pages, then there cannot be any misses in the steady state implying that the stream is satisfied. ■

Observation III.2 The throughput of a satisfied stream is equal to r , its request rate.

Proof: By definition, a satisfied stream experiences no misses in the steady state. No misses implies no stall time and the stream proceeds at its desired request rate of r . ■

Lemma III.2 Cache Pollution occurs if $(g+1) > \lceil r \cdot t(p) \rceil$.

Proof: If $g+1 > \lceil r \cdot t(p) \rceil$ then $g \geq r \cdot t(p)$ or $t(p) \leq g/r < (g+1)/r$.

By Lemma III.1, the stream is satisfied at the chosen (p, g) but is also satisfied for $g-1$ at $(p, g-1)$. By Observation III.2, the throughput with $g-1$ will remain the same as the stream will remain satisfied. However, the cost of the case where a lower g is used is smaller as the average number of pages that have to be kept in the cache is smaller. Hence, cache space is being wasted without any gain in throughput. ■

Lemma III.3 If there is no cache pollution, wastage occurs iff $p/r > L$.

Proof: If there is no cache pollution, $(g+1) \leq \lceil r \cdot t(p) \rceil$ (Lemma III.2). By their definitions, p pages are requested when $g+1$ pages from the previous prefetch are still unaccessed. The number of these pages that are consumed in the time it takes to prefetch is $r \cdot t(p)$, which is roughly all of the unaccessed pages. Hence, as soon as the next p pages are prefetched, they begin to be consumed at the request rate r . Therefore, the time it takes for the p pages to be consumed after prefetch is p/r . Now, if the average life (L) of pages in the cache is less than p/r , then some of the prefetched pages will be evicted before they are requested. Conversely, if L is greater than p/r then all the p pages will be requested before they reach the LRU end of the list and face eviction. ■

Lemma III.4 If there is no cache pollution, throughput of a stream (b) = $\min(r, \frac{p}{t(p)}, \frac{r \cdot L}{t(p)})$.

Proof: The throughput of a stream cannot exceed its request rate (r). Further, since we use a single outstanding prefetch for a stream at any time, the throughput cannot exceed the amount prefetched (p) divided by the time it takes to prefetch that amount $t(p)$. In the case where $p > r \cdot L$, wastage occurs (Lemma III.3) and only $r \cdot L$ pages out of p will be accessed before being evicted. In this case, the throughput is limited by $r \cdot L/t(p)$. ■

Lemma III.5 If there is no wastage $B \cdot L = C$

Proof: The life of the cache (L) is equal to the time it takes to insert C new pages in the top end of the cache. If there is no wastage, the rate of insertion in the cache is equal to the aggregate read throughput of all the streams (B). Therefore, $C/B = L$. ■

Lemma III.6 For a fixed choice of p_1, p_2, \dots, p_n , and cache of size C , the aggregate throughput (B) is unique when there is no wastage.

Proof: Suppose, the aggregate throughput (B) was not unique. Without loss of generality, we would have

$B' > B$, such that

$$B' = \sum_{i=1..n} \min(r_i, p_i/t(p_i), r_i \cdot L'/t(p_i))$$

$$B = \sum_{i=1..n} \min(r_i, p_i/t(p_i), r_i \cdot L/t(p_i))$$

Since there is no wastage, we have $p \leq r \cdot L$ for all streams. So, the only different term in the \min expression is not significant. Thus, $B' = B$, which is contrary to our assumption. ■

Theorem III.1 *The aggregate throughput of n streams sharing a cache of size C with average cache life L , is maximized if $\forall i, p_i = \lfloor r_i \cdot L \rfloor$.*

Proof: Given n streams with request rates of r_1, r_2, \dots, r_n and a cache of size C , let the throughput obtained by the choice: $\forall i, p_i^T = \lfloor r_i \cdot L \rfloor$ be B_T . The theorem claims that B_T is the maximum aggregate bandwidth obtainable (B_{max}) through any choice of p_i^T .

We will prove by contradiction. Let us assume that $B_T < B_{max}$. Therefore, there exists some choice of p_1, p_2, \dots, p_n such that the aggregate bandwidth of all streams is B_{max} .

Since wastage and cache pollution can never increase aggregate throughput, we assume, without loss of generality, that B_{max} is free from these inefficiencies.

If the choice of p_i is the same as that specified by this theorem, then by Lemma III.6, $B_T = B_{max}$, which is contrary to our assumption.

$$\therefore \exists i : p_i \neq \lfloor r_i \cdot L \rfloor \quad (1)$$

By Lemma III.3, it must be the case for B_{max} that

$$\forall i, p_i \leq \lfloor r_i \cdot L \rfloor \quad (2)$$

It follows from (1) and (2), without loss of generality, that $p_1 < \lfloor r_1 \cdot L \rfloor$.

Let us define a new set: $p_1^1, p_2^1, \dots, p_n^1$, where $p_1^1 = \lfloor r_1 \cdot L \rfloor$, and $\forall i \neq 1, p_i^1 = p_i$. L^1 and B^1 are the new cache life and aggregate throughput values.

Since $B^1 \leq B_{max}$ (by defn. of B_{max}), $L^1 \geq L$ (Lemma III.5). By Lemma III.4, $\forall i \neq 1, b_i^1 \geq b_i$ as $p_i^1 = p_i$ and $L^1 \geq L$. By Observation III.1 and Lemma III.4, $b_1^1 \geq b_1$ as $p_1^1 > p_1$.

$$\therefore B^1 = \sum b_i^1 \geq \sum b_i = B_{max}$$

Since $B^1 \leq B_{max}$, it follows that $B^1 = B_{max}$.

By repeating the above procedure for every stream with $p_i < \lfloor r_i \cdot L \rfloor$, we will arrive at a set $p_1^n, p_2^n, \dots, p_n^n$, where $\forall i, p_i^n = \lfloor r_i \cdot L \rfloor$ and $B^n = B_{max}$.

Since, the choice of p for each stream will then be the same for B^n and B_T , $B^n = B_T$ by Lemma III.6.

$$\therefore B^n = B_{max} = B_T$$

which contradicts our assumption that $B_T < B_{max}$. ■

C. AMP Algorithm

The AMP algorithm, which adapts to achieve the optimality criteria set in the previous section, is outlined in Figures 3 and 4. We now draw attention to the important portions of the algorithm and the logic behind the choices we have made.

We make a conscious effort to avoid a separate data structure to track the adapted values of p and g for each detected sequential stream. We store the value of p and g in the page data structure. This removes any restriction on the number of streams that can be tracked.

Lines 20-23 implement the synchronous prefetching component of the algorithm. The number of pages to be prefetched on a read miss is not fixed (as in FS algorithms) but is the adapted value of p stored in the metadata of the previous page.

Whenever the current p is greater than the Asynchronous Prefetch Threshold (APT), asynchronous prefetching is activated. APT is set to an empirically reasonable value of 4. A page at a distance of $APT/2$ from the last page prefetched is chosen as the prefetch trigger page and the tag is set (Lines 41, 49). When there is a hit on a tag page, the tag is reset and an asynchronous prefetch is initiated for p pages as specified in the last page of the current set (Lines 27-30).

Adapting the degree of prefetch (p): As per Theorem III.1, we desire to operate at a point where $p = r \cdot L$. If p is more than this optimal value, the last page in a prefetched set will reach the LRU end unaccessed. We give such a page another chance by moving it to the MRU position and setting the *old* flag (Line 53). Whenever an unaccessed page is moved to the MRU position in this way, it is an indication that the current value of p is too high, and therefore, we reduce the value of p (Line 56). In Lines 31-35, p is incremented by the *readsize* (the size of read request in pages) whenever there is a hit on the *last* page of a read set (pages read in the same I/O) which is also not marked *old*.

Adapting the trigger distance (g): The main idea is to increment g if on the completion of a prefetch we find that a read is already waiting for the first page within the prefetch set (`readWaiting()`). If g was larger, the prefetch would have been issued earlier, reducing or eliminating the stall time for the read. Thus, we increment g in Line 47. However, we also need to decrement g when p itself is being decremented (Line 57). This keeps $g = r \cdot t(p)$ as per Lemmas III.1, III.2.

Whenever we adapt the value of p and g we store the updated values in the last page that has been read into the cache for the sequence. This is located by the `lastInSequence(x)` method in Lines 11-19. The method simply returns the last page of the set that x belongs to, or if the next set of pages has also been prefetched,

DATA STRUCTURE AND FUNCTIONS:

```
struct page {
    off_t addr;    // addr of the page
    off_t Laddr;  // addr of the last page in read set
    bool accessed; // whether page has been accessed
    bool tag;     // if page hit will initiate prefetch
    bool old;     // if page was given another chance
    short int p;  // prefetch degree
    short int g;  // trigger distance
}

lookup(off_t addr)
1:  if (page addr is present in cache)
2:      return page
3:  endif
4:  return null

#define prev(x) lookup(x→addr - 1)
#define last(x) lookup(x→Laddr)
#define isLast(x) (x→addr == x→Laddr)

createPages(page[] s, page ** last, page ** prev)
5:  *prev = prev(first page in s)
6:  *last = the last page in s
7:  foreach x in s; do
8:      create page x in cache
9:      x→Laddr = (*last)→addr
10: done

lastInSequence(page * x)
11: l1 = last(x)
12: if (!l1)
13:     return null
14: endif
15: if (!lookup(x→Laddr + 1))
16:     return l1
17: elsif (l2 = lookup(x→Laddr + l1→p))
18:     return l2
19: endif
```

Fig. 3. AMP: Data structures and support functions

it returns the last page of the next set. The logic is to keep the adapted values of p and g in the page that is most recently added to the cache and is thus most likely to stay in the cache for the longest time. In the unlikely case where the adapted values of p and g are forgotten because of page evictions, we set p to the current prefetch size, and set g to half of p .

Since AMP adapts to discover the optimal values of p and g , it incurs a minor cost of adaptation which quickly becomes negligible and allows it achieve near optimal performance.

In the eviction part of the algorithm (Lines 50-59), pages that are *old* or *accessed* are evicted from the LRU end. Finally, we always make sure that $p \geq g + 1$ (Lines 44, 57) and $p \leq p_{threshold}$ (a reasonable 256 pages for all algorithms).

ON HIT OR MISS:

A read request of *readsize* pages is processed one page at a time as follows:

```
On read miss on page x
20: read page x along with
21:   (a) remaining pages in read request
22:   (b) if (prev(x))
23:       prev(x)→p pages beyond the read request

On read hit on page x
24: if (x→accessed)
25:     moveToMRUPosition(x)
26: endif
27: if (x→tag)
28:     prefetch [x→Laddr + 1, x→Laddr + last(x)→p]
29:     x→tag = 0
30: endif
31: if (isLast(x) && !x→old)
32:     if (y = lastInSequence(x))
33:         y→p = y→p + readsize
34:     endif
35: endif

On reading page x (after hit or miss)
36: x→accessed = 1
```

ON DISK READ COMPLETION:

```
When a read completes for a set (s) of pages
37: createPages(s, &last, &prev)
38: last→p = (prev ? prev→p : 0) + readsize
39: if (last→p ≥ APT)
40:     last→g = APT/2
41:     (lookup(last→addr - APT/2))→tag = 1
42: endif
```

```
When prefetch completes for a set (s) of pages
43: createPages(s, &last, &prev)
44: last→p = max(prev→p, last→g + 1)
45: last→g = prev→g
46: if (readWaiting())
47:     last→g = last→g + readsize
48: endif
49: (lookup(last→addr - prev→g))→tag = 1
```

EVICTON ALGORITHM:

```
When page x reaches the LRU end
50: if (x→old || x→accessed)
51:     evict page x from cache
52: else
53:     x→old = 1
54:     moveToMRUPosition(x)
55:     if (y = lastInSequence(x))
56:         y→p = y→p - 1
57:         y→g = min(y→g - 1, y→p - 1)
58:     endif
59: endif
```

Fig. 4. AMP: Main algorithm

IV. WORKLOADS

A. Which workloads to use

The study of caching algorithms using traces is popular as it simplifies the experimental setup while allowing to simulate a real-life workload scenario. Another approach is to use synthetic workload generators which are flexible and can simulate a large number of scenarios for which it is not practical to obtain real-life traces. When the algorithms being tested involve prefetching, using traces is not a good choice. Firstly, the timing of the I/Os is crucial in the context of prefetching algorithms. A read can be a miss or hit depending on the amount of time that passed between consecutive requests. This is not the case with pure demand-paging algorithms where we will get the same hit ratio independent of the timing between the read requests. To ensure that we are faithful to the timing information in the traces we need to run the trace preferably on the same hardware that generated it in the first place. For example, it is impossible to run a trace from a data server with hundreds of disks on a setup with only a few disks. This requires us to either simulate the original hardware on which the trace was collected, or scale the speed of the trace based on the disparity of the two systems. When using older traces, we may also need to factor in the improvement of disk access times. Therefore, using traces for comparison of prefetching algorithms is extremely difficult and an approximation at best. We favor using versatile workload generators which can simulate both simple workloads and complex workloads like OLTP and Video-on-Demand.

B. Sequential Streams

This workload comprises of a continuous series of read requests on consecutive pages with a specified time (*thinktime*) between the requests. Each request is for the specified number of pages (*readsize*). We examine both single stream and multiple stream cases.

C. Short Sequences

Each stream of this kind generates read requests for consecutive pages for the specified sequence length. Once the sequence length is read, the stream randomly selects a new location to start reading another short sequence with the specified *thinktime* and *readsize*.

D. SPC1-Read workload

SPC-1 ([36], [17]) is a widely used commercial benchmark provided by the Storage Performance Council. It uses a sophisticated workload that simulates business critical environments like OLTP systems, database systems and mail server applications. We use a prototype implementation of the SPC-1 benchmark called SPC1-Read, that matches the specifications of the read

component of the SPC1 workload([37]), which comprises uniform random (10%), hierarchical reuse random (65%), and incremental sequential (25%) access patterns. The *thinktime* and *readsize* of the constituent workloads are continuously varied according to the probability distribution prescribed by the benchmark. The impact that the write component has on concurrent reads depends on the choice of the size of the write cache, the order of destages as well as the timing of the destages([38]). It serves us well to ignore the write component of the workload so as to clearly appreciate the relative performance of the prefetching algorithms without diluting the results with writes.

E. SPC2-VOD workload

The SPC-2 benchmark is designed to demonstrate the performance of a storage subsystem when running business critical applications that require the large-scale, sequential movement of data. It is comprised of tests that simulate applications characterized by large I/Os. One such test which is purely reads (and hence of interest to us), is the *Video on Demand* test which simulates individualized video entertainment provided to a community of subscribers, by drawing from a digital film library [39]. The workload creates the specified number of sequential streams that read 256 KB on each I/O with a thinktime of 333.3 ms.

V. EXPERIMENTAL SET-UP

A. The Basic Hardware Set-up

We use an IBM xSeries 345 machine equipped with two Intel Xeon 2 GHz processors, 4 GB DDR, and six 10K RPM SCSI disks (IBM, 06P5759, U160) of 36.4 GB each. A Linux kernel (version 2.6.11) runs on this machine hosting all our workload generators and cache simulation framework. We employ five SCSI disks for the purposes of our experiments, and the remaining one for the operating system, our software, and workloads.

B. Software Setup

We implemented a framework, called *WorkGen* as shown in Figure 5, which allows us to benchmark various algorithms across a large range of cache sizes, varying page sizes, and consumption rates. *WorkGen* is divided into three layers. The first layer consists of the workload modules, which can simulate any number of parallel streams consisting of sequential, short sequential, SPC1-Read, or SPC2-VOD workloads. The second layer consists of the prefetching modules, which allows us to select any of the prefetching algorithms that we implemented. Each algorithm module has access to the LRU data structure as well as the `read()`, and `prefetch()` methods used to perform demand reads and prefetches respectively. The third layer consists of the disk backend which is the target for all the I/Os.

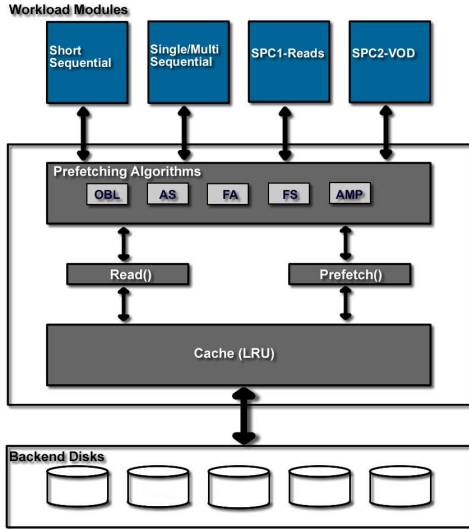


Fig. 5. WorkGen Architecture

C. The Competitors

We implemented prefetching algorithms that represent the FS, FA, and AS classes.

Within the FS and FA classes we pick members with small, medium and large p . In Figure 6, we observe that for the FA algorithms there is no optimal fixed value for g that works for all workloads. We have chosen g to be half of p as that works best for the widest variety of workloads. For AS algorithms, we chose two popular variants, which adapt p linearly (AS_{Linear}) and exponentially (AS_{Exp}). We also compare with OBL and the case with no prefetching.

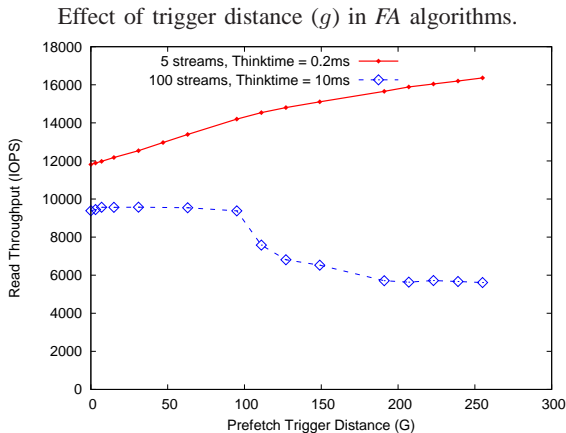


Fig. 6. On x-axis we vary the trigger distance (g) in an FA algorithm with $p = 256$. On the y-axis we show the throughput when using a 120 MB cache. When using five fast streams we get higher throughput with higher values of g , whereas, with a hundred slower streams, a smaller g performs better.

D. Measuring success

The ultimate goal of any cache management algorithm is to improve the shape of the throughput-response time curve for the system by lowering the response times and increasing the throughput across all workloads. Most caching research has focused on minimizing miss ratios (or maximizing hit ratios) which at best is a good heuristic for improving performance of a system. To be fair it is not just the miss ratio but also the average cost of misses that impacts the aggregate response time. For example, an aggressive prefetching algorithm can potentially reduce the miss ratio but suffer a severe increase in the average cost of misses as it overloads the disks. In fact, with prefetching, the concept of a read miss itself is nebulous because a read that happens after a prefetch request for the page has been issued and before the prefetch actually completes is somewhere between a hit and a miss, but technically neither. Even in the absence of prefetching, some disks might be less busy than others leading to smaller miss penalties on those disks. Even on a single disk reading from an area that is not visited often by the disk head tends to be more expensive. In short, it is prudent to measure performance in terms of aggregate read response times and throughput whenever possible.

Another quantity which is useful is the *stall time*. It is the total time for which application had to wait because the requested data was not present in the cache. This is very closely related to the aggregate throughput as a lower stall time results in correspondingly higher throughput. We however choose to report in terms of throughput as it is more immediately relevant to performance.

VI. RESULTS

A. Single Sequential Stream

Our goal is to create an intimate understanding of the behavior of various sequential prefetching algorithms. We implemented 9 prefetching algorithms and compared them with AMP. In Figure 7, we examine the actual throughput achieved as a function of the requested rate by a single sequential stream when assisted by various prefetching algorithms.

As expected, we observe the lowest throughput for no prefetch. The One Block Lookahead (OBL) algorithm performs about the same because we prefetch only one page and the request size is 2 pages. A two or more block lookahead algorithm would have worked better by reducing the amount of misses. This is precisely the intent of the Fixed Synchronous (FS) class of algorithms which have a fixed degree of prefetch (p).

FA algorithms are superior to the FS algorithms because they can start a prefetch on a hit thereby avoiding more misses than the FS algorithms. Both the

Single Sequential Stream with varying request rates

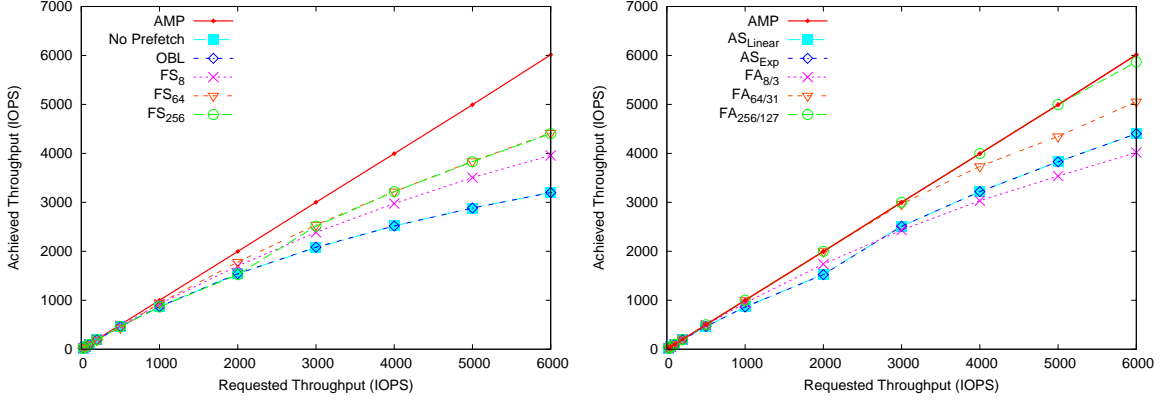


Fig. 7. We show the achieved throughput as a function of the requested throughput for 8 KB reads using a 100 MB cache and one SCSI disk. For clarity, we split the comparison with AMP into two panels. AMP keeps up with the requested throughput and outperforms all other algorithms.

Multiple Sequential Streams with varying cache sizes

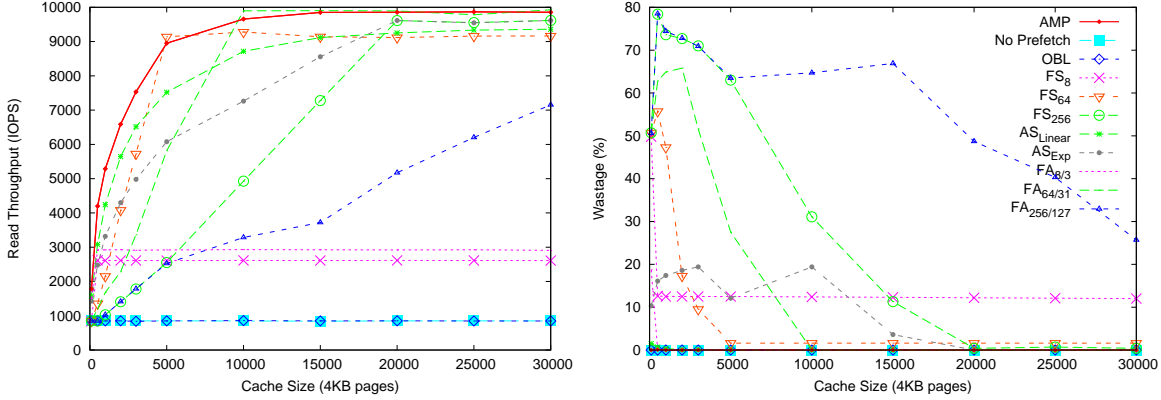


Fig. 8. On the left panel, we show the average achieved throughput over a period of two minutes as a function of the cache size for 100 concurrent streams reading from five SCSI disks with thinktime = 10 ms and readsize = 8 KB. On the right panel, we show the corresponding wastage percentage (unaccessed pages evicted / total pages evicted * 100%).

FS and FA algorithm results seem to indicate that the larger the p , the better the performance. This is true for a single sequential stream where the cache is abundantly available. In multiple stream experiments which compete for cache space, we will better appreciate the need for a careful choice of p .

We also measure the performance of the adaptive synchronous algorithms. AS_{Linear} and AS_{Exp} increase p as the detected sequence becomes longer. For all the adaptive algorithms in this paper we uniformly use a maximum degree of prefetch of 256 for a fair comparison. Both AS variants perform roughly the same as they adapt and reach this maximum value of p during the first few seconds on the experiment.

AMP being able to adapt not only p but also g convincingly outperforms the FA algorithms by 2-50%, the AS algorithms by 37%, the FS algorithms by 37-

52% and no prefetching and OBL by 88%. AMP closely followed by $FA_{256/127}$ was able to satisfy the request rate throughout the single stream experiment.

B. Multiple Sequential Streams: varying cache size

It is extremely important to examine the common case where a limited cache is used by prefetching algorithms to cater to multiple sequential streams. In Figure 8, we depict the aggregate throughput achieved by a hundred parallel sequential streams with a think-time of 10 ms. The key observation is that different algorithms are suitable for different cache sizes, while AMP is universally the best, closely enveloping all the other plots.

Out of the FS algorithms, FS_8 is the best at very low cache sizes, while FS_{64} is much better at the higher cache sizes. In FA algorithms, $FA_{8/3}$ is again the best

Varying number of Sequential Streams

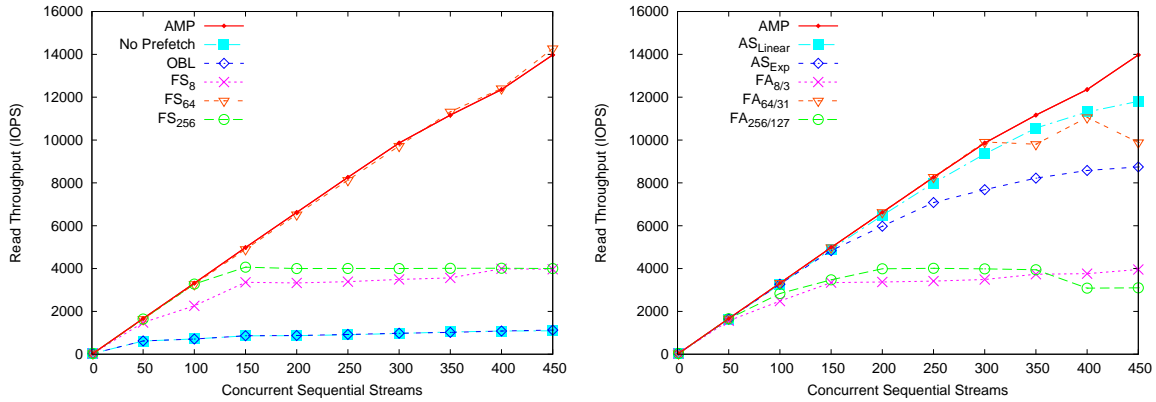


Fig. 9. We show the average achieved throughput over a period of two minutes as a function of the number of concurrent sequential streams reading from five SCSI disks with thinktime = 30 ms, readsize = 8 KB, and a 100 MB cache.

at low cache sizes, while $FA_{256/127}$ is much superior at the higher cache sizes. As a rule of thumb, a lower prefetch degree performs better at lower cache sizes as high values of p create prefetch wastage in smaller caches. This is in contrast to the single stream case, where a higher p was always a better choice. This leads us to the adaptive synchronous algorithms. AS_{Exp} increases p exponentially, reaching higher values of p quickly creating significant prefetch wastage for lower cache sizes. AS_{Linear} , being slower in its adaptation, usually performs better than AS_{Exp} at lower cache sizes while the reverse is true at higher cache sizes. The only algorithm that truly adapts the value of p so as to minimize prefetch wastage is AMP. As a rough measure of the overall performance of each algorithm, we compute the average throughput across all cache sizes in Figure 8. We find that AMP outperforms the FA algorithms by 29-172%, the AS algorithms by 12-24%, the FS algorithms by 21-210% and no prefetching and OBL by a factor of 8. This is also a testimonial to the fact that AMP algorithm closely follows the optimality criteria derived in Section III-B.

In the right panel of Figure 8 we plot wastage, which is the percentage of evicted pages that were evicted before they could be accessed. We observe that the prefetching algorithms that have a high p or aggressively increase p suffer from the most prefetch wastage, and that wastage is larger for smaller cache sizes. The prefetch wastage in the case of AMP is always less than 0.1%, while on an average, FA, FS, and AS algorithms waste up to 60%, 41%, and 11% respectively.

C. Multiple Sequential Streams: varying number of streams

In Figure 9, we study the aggregate streaming throughput of various prefetching algorithms when we

increase the number of concurrent sequential streams while keeping the cache size constant. We observe that most algorithms saturate at some throughput beyond which increasing the number of streams does not improve the aggregate throughput. Algorithms that issue fewer but larger disk reads and at the same time waste little generally do better. We observe that no prefetching and OBL have the lowest throughput as they have a large number of small read requests. Somewhat better are the $FA_{8/3}$ and FS_8 algorithms as they create fewer read requests than OBL. Interestingly, $FA_{256/127}$ and FS_{256} algorithms also have similarly low performance in spite of large prefetch degree. This is because the large prefetch degree leads to significant prefetch wastage. The FS_{64} and $FA_{64/31}$ perform the best in their respective classes as they strike a balance and have large reads but do not waste as much. The AS_{Linear} and AS_{Exp} are generally good performers because they adapt the degree the prefetch. However, since these algorithms lack the ability to detect and avoid wastage, the more aggressive AS_{Exp} fares worse than its linear counterpart. AMP being an asynchronous adaptive algorithm discovers the right prefetch degree for each stream thus avoiding wastage and achieving the best possible performance.

At the maximum number of streams, AMP outperforms the FA algorithms by 41-350%, the AS algorithms by 18-60%, the FS algorithms from nearly equal to 252% and no prefetching and OBL by a factor of 12.

D. SPC1-Read workload

We study the impact of the various prefetching algorithms on the performance of the cache when subjected to the read component of the SPC1 benchmark workload. In Figures 10, 11 we show the aggregate response time as a function of the obtained throughput. Lower plots indicate better performance as at the same

SPC1-Read in a small cache scenario

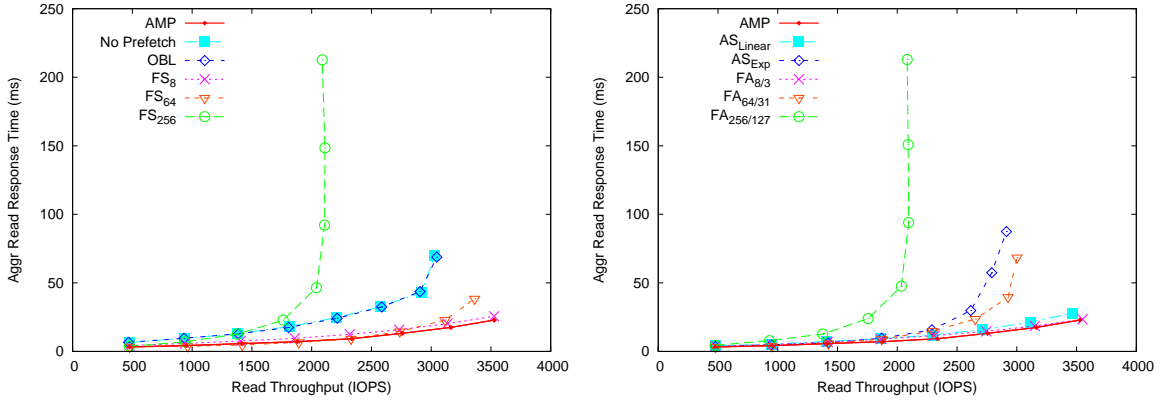


Fig. 10. We measure the aggregate read response time as a function of the achieved throughput when running the read portion of the SPC1 benchmark in a small cache scenario: 120 MB cache, backend=3.5 GB. The above graph also shows an example where no prefetching performs better than aggressive prefetching algorithms like FS_{256} , $FA_{256/127}$ and AS_{Exp} . This underscores the importance of wisely adapting p according to the workload.

SPC1-Read in a large cache scenario

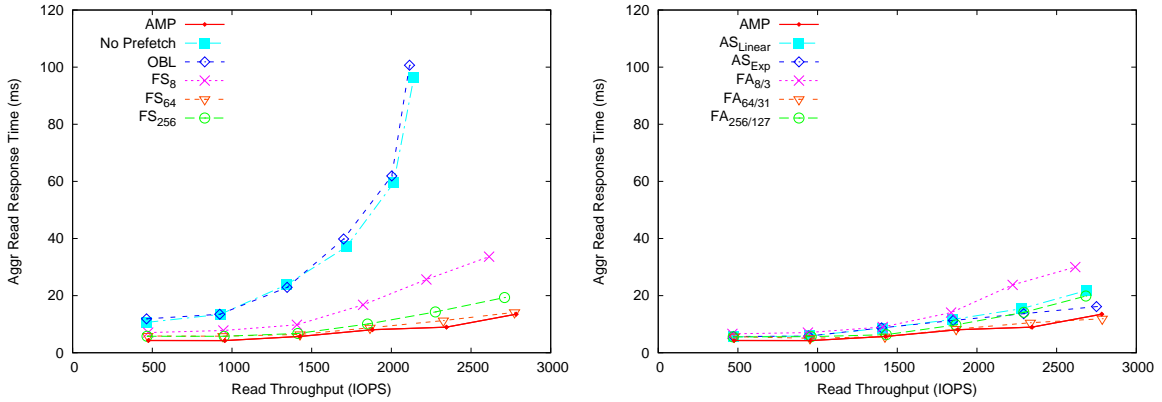


Fig. 11. We measure the aggregate read response time as a function of the achieved throughput when running the read portion of the SPC1 benchmark in a large cache scenario: 2 GB cache, backend = 35 GB

response time a higher throughput can be achieved. Clearly, AMP is consistently the best performing algorithm in both the small cache and large cache scenarios. No other algorithm can perform well in both scenarios. When the cache is large, the algorithms with a higher prefetch degree seem to do better, while in the small cache scenario, where the cache is precious, the algorithms that are conservative in their prefetch degree tend to perform much better as they incur lesser prefetch wastage. AMP is able to quickly adapt to different workloads and cache sizes, hence performing the best among all the algorithms.

E. Short Sequences

In Section III-B, we derived the optimality criteria for sequential prefetching in the steady state. We have not discussed the behavior of the sequential prefetching algorithms when the average length of sequences is

rather short. Apart from providing close to optimal performance for long streams, AMP achieves the best overall performance for short streams as well. Figure 12 shows the throughput of various algorithms as the length of sequences go from 1 (effectively random) to 8192 read I/Os. The AS algorithms along with FS_8 and $FA_{8/3}$ perform well for short sequence lengths as they have a smaller p and suffer from less prefetch wastage. As the sequence lengths are increased, the $FA_{256/127}$ becomes a strong contender. The fact that AMP starts off with a small p and adapts to make it larger if necessary makes it perform reasonably well for short sequences. As the length of sequences becomes larger, the adaptive power of AMP allows it discover the right combination of p and g . AMP is therefore not only provably optimal for steady sequential streams but also has the best overall performance for short sequences as well.

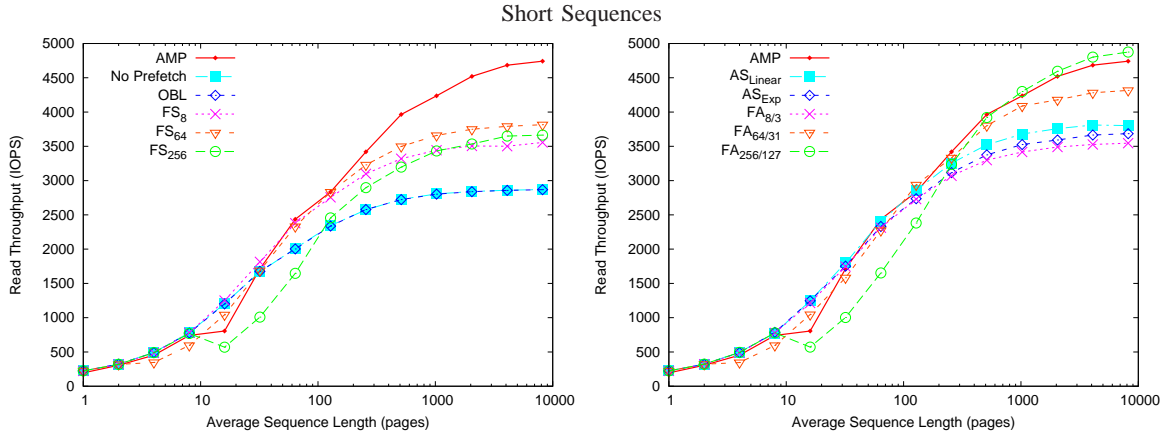


Fig. 12. We show the achieved throughput as a function of the average sequence length ranging from 1 I/O (essentially random) to 8192 I/Os. We use a single stream with thinktime = 0.2 ms and a 100 MB cache.

SPC-2 Video-on-Demand with varying number of streams

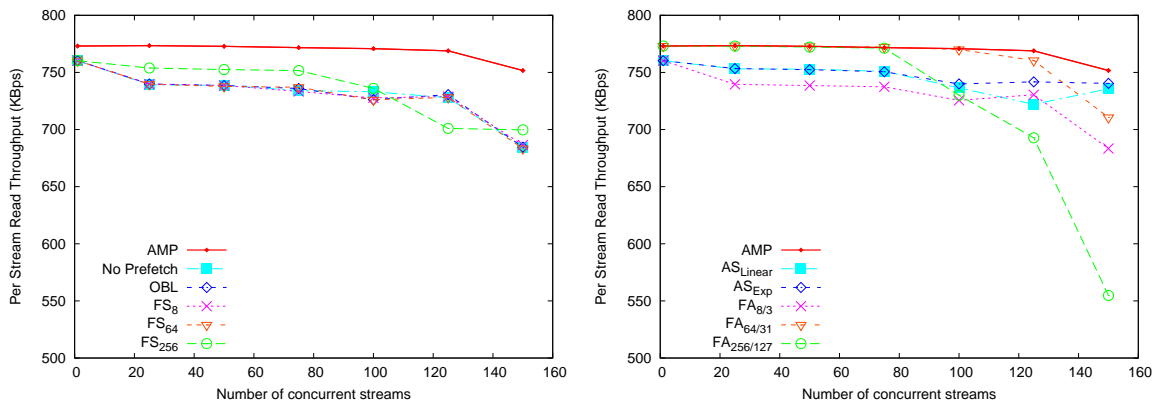


Fig. 13. We measure the achieved throughput per stream as the number of concurrent video-on-demand streams increases using five SCSI disks and a cache of 100 MB. The SPC2-VOD workload uses read size of 256 KB, thinktime = 333.3 ms.

F. SPC2-VOD workload

In Figure 13 we measure the performance for video-on-demand workloads. The goal is to provide each sequential stream its requisite bandwidth (768 KBps in the case of SPC2-VOD workload) for the maximum number of streams. We can easily observe that AMP is able to entertain the most number of concurrent streams (up to 125) at the desired bandwidth. $FA_{64/31}$ starts failing at about 100 streams and $FA_{256/127}$ fails after 75 streams because of more severe prefetch wastage. None of the other algorithms can match the demanded rate as they incur expensive read misses which stall the client and lower the throughput.

VII. CONCLUSIONS

Sequential prefetching is the most widely used prefetching technique in storage subsystems. We have argued the need for an algorithm that can adapt both the prefetch degree and the trigger distance on a per

stream basis in response to evolving workloads. We have provided a theoretical analysis and proved the sufficient conditions for optimal online cache management for steady-state sequential streams. We also presented a novel, simple, adaptive and practical implementation called AMP. We have demonstrated through a series of wide ranging experiments including realistic benchmarks, that AMP provides the highest possible aggregate throughput when a cache is shared among multiple sequential streams. Even in scenarios where the sequential streams are not steady, comprise of short sequences, or are intermixed with random workloads (as in SPC1-Read), we demonstrated that AMP convincingly outperforms all competing algorithms by wasting the least amount of cache while providing the best overall throughput.

We anticipate AMP to be widely applicable not only in storage subsystems, but in any system that services sequential workload.

REFERENCES

- [1] A. Rogers and K. Li, "Software support for speculative loads," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, vol. 27(9), (New York, NY), pp. 38–50, ACM Press, 1992.
- [2] K. K. David Callahan and A. Porterfield, "Software prefetching," in *ACM SIGARCH Computer Architecture News*, vol. 19(2), (New York, NY), pp. 40–52, ACM Press, 1991.
- [3] C. Metcalf, "Data prefetching: a cost/performance analysis," 1993.
- [4] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," in *SOSP*, pp. 79–95, 1995.
- [5] T. Mowry and A. Gupta, "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 12, no. 2, pp. 87–106, 1991.
- [6] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum, "Compiler-directed data prefetching in multiprocessors with memory hierarchies," in *Proceedings 1990 International Conference on Supercomputing, ACM SIGARCH Computer Architecture News*, vol. 18(3), pp. 354–368, 1990.
- [7] T.-F. Chen and J.-L. Baer, "Reducing memory latency via non-blocking and prefetching caches," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, vol. 27(9), (New York, NY), pp. 51–61, ACM Press, 1992.
- [8] C.-K. Luk and T. C. Mowry, "Compiler-based prefetching for recursive data structures," in *Architectural Support for Programming Languages and Operating Systems*, pp. 222–233, 1996.
- [9] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," *ACM SIGPLAN Notices*, vol. 33, no. 11, pp. 115–126, 1998.
- [10] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger, "SPAD: Software prefetching in pointer- and call-intensive environments," in *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 231–236, 1995.
- [11] T. M. Kroeger, D. D. E. Long, and J. C. Mogul, "Exploring the bounds of web latency reduction from caching and prefetching," in *USENIX Symposium on Internet Technologies and Systems*, 1997.
- [12] K. M. Curewitz, P. Krishnan, and J. S. Vitter, "Practical prefetching via data compression," pp. 257–266, 1993.
- [13] J. Griffioen and R. Appleton, "Reducing file system latency using a predictive approach," in *USENIX Summer*, pp. 197–207, 1994.
- [14] D. Kotz and C. S. Ellis, "Practical prefetching techniques for parallel file systems," in *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pp. 182–189, IEEE Computer Society Press, 1991.
- [15] K. S. Grimsrud, J. K. Archibald, and B. E. Nelson, "Multiple prefetch adaptive disk caching," *IEEE Transactions on Knowledge and Data Engineering*, vol. 5, no. 1, pp. 88–103, 1993.
- [16] S. Harizopoulos, C. Harizakis, and P. Triantafillou, "Hierarchical caching and prefetching for high performance continuous media servers with smart disks," *IEEE Concurrency*, vol. 8, no. 3, pp. 16–22, 2000.
- [17] B. S. Gill and D. S. Modha, "SARC: Sequential prefetching in adaptive replacement cache," in *Proceedings of the USENIX 2005 Annual Technical Conference*, pp. 293–308, 2005.
- [18] A. J. Smith, "Cache memories," *ACM Computing Surveys*, vol. 14, no. 3, pp. 473–530, 1982.
- [19] F. Dahlgren, M. Dubois, and P. Stenström, "Fixed and adaptive sequential prefetching in shared memory multiprocessors," in *ICPP*, pp. 56–63, 1993.
- [20] M. K. Tcheun, H. Yoon, and S. R. Maeng, "An adaptive sequential prefetching scheme in shared-memory multiprocessors," in *ICPP*, pp. 306–313, 1997.
- [21] T. Cortes and J. Labarta, "Linear aggressive prefetching: A way to increase the performance of cooperative caches," in *Proceedings of the Joint International Parallel Processing Symposium and IEEE Symposium on Parallel and Distributed Processing*, (San Juan, Puerto Rico), pp. 45–54, 1999.
- [22] J. W. C. Fu and J. H. Patel, "Data prefetching in multiprocessor vector cache memories," in *Proceedings of the 18th annual international symposium on computer architecture*, (Toronto, Ontario, Canada), pp. 54–63, 1991.
- [23] R. L. Lee, P.-C. Yew, and D. H. Lawrie, "Data prefetching in shared memory multiprocessors," in *ICPP*, pp. 28–31, 1987.
- [24] T.-F. Chen and J.-L. Baer, "Effective hardware based data prefetching for high-performance processors," *IEEE Trans. Computers*, vol. 44, no. 5, pp. 609–623, 1995.
- [25] F. Dahlgren and P. Stenström, "Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 4, pp. 385–398, 1996.
- [26] K. S. Grimsrud, J. K. Archibald, and B. E. Nelson, "Multiple prefetch adaptive disk caching," *IEEE Trans. Knowl. Data Eng.*, vol. 5, no. 1, pp. 88–103, 1993.
- [27] D. Joseph and D. Grunwald, "Prefetching using markov predictors," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 121–133, 1999.
- [28] J. S. Vitter and P. Krishnan, "Optimal prefetching via data compression," *Journal of the ACM*, vol. 43, no. 5, pp. 771–793, 1996.
- [29] H. Lei and D. Duchamp, "An analytical approach to file prefetching," in *1997 USENIX Annual Technical Conference*, (Anaheim, California, USA), 1997.
- [30] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, "A study of integrated prefetching and caching strategies," in *Measurement and Modeling of Computer Systems*, pp. 188–197, 1995.
- [31] M. Kallahalla and P. J. Varman, "Pc-opt: Optimal offline prefetching and caching for parallel i/o systems," *IEEE Trans. Computers*, vol. 51, no. 11, pp. 1333–1344, 2002.
- [32] T. Kimbrel and A. R. Karlin, "Near-optimal parallel prefetching and caching," in *IEEE Symposium on Foundations of Computer Science*, pp. 540–549, 1996.
- [33] T. Kimbrel, A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. Felten, G. Gibson, A. R. Karlin, and K. Li, "A trace-driven comparison of algorithms for parallel prefetching and caching," in *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pp. 19–34, USENIX Association, 1996.
- [34] P. Reungsang, S. K. Park, S.-W. Jeong, H.-L. Roh, and G. Lee, "Reducing cache pollution of prefetching in a small data cache," in *ICCD*, pp. 530–533, 2001.
- [35] P. Jain, S. Devadas, and L. Rudolph, "Controlling cache pollution in prefetching with software-assisted cache replacement," Tech. Rep. CSG-462, M.I.T., 2001.
- [36] B. McNutt and S. Johnson, "A standard test of I/O cache," in *Proc. Comput. Measurements Group's 2001 Int. Conf.*, 2001.
- [37] Storage Performance Council, "SPC Benchmark-1: Specification, version 1.10.1," September 2006.
- [38] B. S. Gill and D. S. Modha, "WOW: Wide ordering of writes - combining spatial and temporal locality in non-volatile caches," in *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*, pp. 129–142, 2005.
- [39] Storage Performance Council, "SPC Benchmark-2: Specification, version 1.2," September 2006.