Int. J. Communications, Network and System Sciences, 2015, 8, 161-180
Published Online May 2015 in SciRes. http://www.scirp.org/journal/ijcns
http://dx.doi.org/10.4236/ijcns.2015.85018

Predictive Prefetching for Parallel Hybrid Storage Systems

Maen M. Al Assaf

King Abdullah II School for Information Technology, The University of Jordan, Amman, Jordan
Email: m_alassaf@ju.edu.jo

Received 17 April 2015; accepted 11 May 2015; published 14 May 2015

Copyright © 2015 by author and Scientific Research Publishing Inc.
This work is licensed under the Creative Commons Attribution International License (CC BY).
http://creativecommons.org/licenses/by/4.0/
Open Access

Abstract

In this paper, we present a predictive prefetching mechanism that is based on probability graph approach to perform prefetching between different levels in a parallel hybrid storage system. The fundamental concept of our approach is to invoke parallel hybrid storage system's parallelism and prefetch data among multiple storage levels (e.g. solid state disks, and hard disk drives) in parallel with the application's on-demand I/O reading requests. In this study, we show that a predictive prefetching across multiple storage levels is an efficient technique for placing near future needed data blocks in the uppermost levels near the application. Our PPHSS approach extends previous ideas of predictive prefetching in two ways: (1) our approach reduces applications' execution elapsed time by keeping data blocks that are predicted to be accessed in the near future cached in the uppermost level; (2) we propose a parallel data fetching scheme in which multiple fetching mechanisms (*i.e.* predictive prefetching and application's on-demand data requests) can work in parallel; where the first one fetches data blocks among the different levels of the hybrid storage systems (*i.e.* low-level (slow) to high-level (fast) storage devices) and the other one fetches the data from the storage system to the application. Our PPHSS strategy integrated with the predictive prefetching mechanism significantly reduces overall I/O access time in a hybrid storage system. Finally, we developed a simulator to evaluate the performance of the proposed predictive prefetching scheme in the context of hybrid storage systems. Our results show that our PPHSS can improve system performance by 4% across real-world I/O traces without the need of using large size caches.

Keywords

Predictive Prefetching, Probability Graph, Parallel Storage Systems, Hybrid Storage System

1. Introduction

In data-intensive computing systems, researchers have proposed a wide variety of prefetching techniques to preload data from disks prior to the data accesses in order to solve the I/O bottleneck problem (see [1] [2]). Existing prefetching techniques can be categorized into two camps—predictive and informed. Predictive prefetching approaches predict the future I/O access patterns based on historical I/O accesses of applications [3], whereas informed prefetching techniques make preloading decisions based on applications' future access hints [4]. In this study, we focus on predictive prefetching schemes and investigate performance impact of predictive prefetching on hybrid storage systems. We predictively prefetch the data from lower levels (slow) to the uppermost one (fast) in hybrid storage system's levels to improve application's performance and reduce its execution elapsed time.

1.1. Motivations

Since there exists a rapidly increasing performance gap between processors and I/O subsystems, disk performance becomes a serious bottleneck for large-scale computing systems supporting data-intensive applications [5]. Recent studies show that prefetching can bridge the performance gap between the CPU and I/O; for example, a predictive prefetching algorithm that is based on probability graph approach proposed by Griffioen and Appleton aims to build a history based on the application's on-demand I/O reading requests in order to predict the future and prefetch the data to improve performance of I/O-intensive applications [3]. An informed prefetching approach proposed by Patterson et al. takes the advantage of hints provided by the application to improve its performance by applying cost-benefit analysis to allocate buffers for both prefetching and caching and to prefetch the data [4].

In [1] [6] [7], we proposed several informed prefetching mechanisms for hybrid storage systems. Our observations show that there exist some problems with informed prefetching like: application must provide hints, there should be a lead time to ensure file is prefetched, and informed prefetching is not good for multiple executables. On the other hand, predictive approach does not require applications to provide hints and can handle the other problems.

Predictive prefetching should make accurate decisions in order to improve the application' performance and to reduces its execution time.

The following key factors motivate us to investigate predictive prefetching in hybrid storage systems:

1) The growing needs of hybrid storage systems,

2) Predictive prefetching improves hybrid storage system performance.

3) It does not rely on hints offered by applications.

4) Predictive prefetching is good for I/O intensive applications.

5) It is also good for multiple executables.

6) The possibility of multiple data fetching mechanisms working in parallel in a hybrid storage system, and

7) Prefetching utilizes parallel storage system bandwidth.

Hybrid storage systems are important in data centers supporting service-based applications such as scientific computing and multimedia streaming. Hybrid storage systems consist of storage devices with various performance; upper levels show better performance in term of speed. So, popular data are cached in an upper-level storage while massive amounts of unpopular data are placed in lower-level storage servers. Existing studies (see, for example, [8] and [9]) suggest using new prefetching and caching techniques are needed to improve the I/O performance of hybrid storage systems.

In addition, predictive prefetching techniques can improve performance of parallel disks through eliminating I/O stalls. When parallel disks are extended into hybrid storage systems [10] [11], a hierarchy of multiple storage devices increase data access latency if the data are residing in a lower level of the systems. To shorten long data transfer latency, popular data or near future accessed data may be stored in the uppermost level of the storage systems. Predictive prefetching from a lower level to an upper one will do such contribution.

Also, predictive prefetching does not rely on hints offered by applications. It can build history data structures by recording the application' on-demand I/O reading requests. It is also good for I/O intensive applications because it does not require hints from the application so it can prefetch data even if the application is not running.

In addition, predictive prefetching is good for multiple executables because it relies on building a history based on the applications' on-demand requests thus detect access patterns that span multiple applications executed repeatedly rather than taking hints from the running applications.

In a parallel hybrid storage systems, multiple fetching mechanisms can independently fetch data blocks from lower-levels to upper-levels storage devices. These mechanisms can work in parallel, because multiple data fetching operations can be simultaneously processed by the upper-level and lower-level data fetching mechanisms. Such storage parallelisms make it possible to implement a prefetching mechanisms among the different levels of the hybrid storage system that can work in parallel with the application's on-demand I/O reading requests. Multiple fetching mechanisms working in parallel increases the parallel system's utilization.

1.2. Contributions

The following list summarizes the major research contributions made in this paper:

-To reduce I/O delays and application's execution elapsed time in a hybrid storage system, we propose new predictive prefetching approaches to work in parallel with the application on-demand I/O reading requests. It implements the predictive prefetching algorithm proposed by Griffioen and Appleton [3] in a hybrid storage system of two levels (*i.e.* SSD and HDD). The predictive prefetching algorithm developed in this study is called PPHSS. We show that our prefetching works in parallel with the application's on-demand requests.

-We develop a simulated hybrid storage system, in which the PPHSS prefetching technique is implemented. Simulation results show that our prefetching mechanism from lower levels to upper ones in a hybrid storage systems reduces applications' stall and execution elapsed times.

1.3. Roadmap

The rest of the paper explains and justifies a predictive prefetching scheme in hybrid storage system. Section 2 reviews the related work. We outline the PPHSS architecture in Section 3. Section 4 describes the design and implementation issues of the PPHSS prefetching mechanism. Section 5 presents our simulation framework and results. Finally, Section 6 provides conclusions and *directions* for future studies.

2. Literature Review

Previous researchers have suggested that predictive prefetching mechanisms improve I/O performance. To our best knowledge, however, ours is the first study to focus on predictive prefetching in hybrid storage systems, the first to construct a multiple data fetching mechanisms in a hybrid storage system, and the first to offer a systematic performance evaluation of predictive prefetching in hybrid storage systems.

2.1. Hybrid Storage Systems

A hybrid storage system is a heterogeneous hierarchy of storage devices that differ in their specifications: hardware, speed, size, and others [12]. Hybrid storage systems provide cost-efficient solutions for large-scale data centers. In the same time, they do not affect I/O response times. The I/O performance of a hybrid storage system depends on the data placement of the system. Ideally, a high-level storage device should store the popular data that are frequently accessed and data that are likely to be access in the near future.

Modern hybrid storage systems include storage devices like main memory, solid state disks, hard disks, and magnetic tape subsystems (see, for example, **Figure 1**).

Multi-level hybrid storage systems and caches have many advantages, for example, a multi-level cache system contains a hierarchy of several cache levels [13]. Previous studies show that there is more benefit when having a multi-level cache [14] [15].

Similar to multi-level caches, hybrid storage systems [16] [17] first check for the data in the upper-level storage devices. If the data is not found in the upper level storage, the next storage level is checked. This process is repeated from the upper to the lower levels until the required data is retrieved.

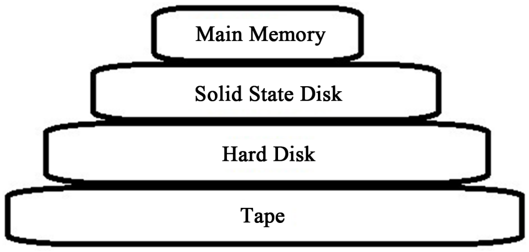

Figure 1. Hybrid storage system that consists of different storage devices with various speed performance. e.g. main memory, solid state disk, hard disk, and a tape.

Research in multi-level hybrid storage systems is not limited to data retrieval time efficiency, it also covers power consumption performance issues. A recent study conducted by us (see: [6]) provided an energy aware informed prefetching mechanism for hybrid storage systems based on Patterson *et al*. [4] solution. We investigated how informed prefetching can make hard disk drives of a parallel hybrid storage system to hibernate as long as possible in order to save power. Another recent study conducted by us (see: [18]) provides a thermal modeling of hybrid storage clusters that consist of two layers (HHDs and SSDs). The model aims to minimize the negative thermal impacts of hybrid storage clusters and to estimate the hybrid storage system's cooling cost.

2.2. Predictive Prefetching

Predictive Prefetching (a.k.a., automatic prefetching) relies on past I/O accesses to predict future access and to prefetch them [3] [19]. Griffioen and Appleton developed a model based on probability graph approach that is used to record the application's past access patterns in order to predict future access probabilities [3]. In their model, probability graph data structure involves using directed weighted graphs to estimate access probabilities [3].

The Markov predictor is another model used by existing predictive prefetching algorithms [20]-[26]. It predicts future data accesses by applying partial match to find recurring sequences of I/O events. The Markov prediction algorithms used to be widely applied to do prefetching for web applications because of the hypertextual nature of the web accesses over the Internet.

More details about predictive prefetching approach will be discussed in latter sections.

2.3. Informed Prefetching

Patterson *et al*. [4] [27]-[30] conducted an informed prefetching algorithms that invokes storage parallelisms and take advantage of the application' disclosed I/O access hints to eliminate I/O stalls by doing aggressive prefetching [4] [28] [30].

Other studies undertaken by Huizinga *et al*. [31] and Chen *et al*. [32].

In parallel storage systems, informed prefetching aims to leverage parallel I/O to improve prefetching performance [33] [34].

Parallel informed prefetching eliminates I/O stalls by prefetching hinted data in the cache before it is actually requested by the applications. To improve cache usage for both prefetching and caching, Patterson et al. proposed a cost-benefit model, that performs informed prefetching as well as it balances cache/buffer space that is shared between the LRU (least-recently-used) cache and the prefetching buffer [4]. The model makes compromise between the benefit of using more buffers for prefetching and the cost of ejecting a LRU block or a prefetched data block.

Several studies investigated various ways of collecting information to offer accurate access hints for informed prefetching mechanisms. Accurate hints are important to make informed prefetching efficient (See: [35] [36]).

2.4. Prefetching in Hybrid Storage Systems

Prefetching in hybrid storage systems is technically challenging. This is because aggressive prefetching is important to reduce I/O latency [37], but on the other hand, overaggressive prefetching may waste I/O bandwidth by transferring useless data from lower-level to upper-level storage devices and pollute its space.

Previous studies provided empirical evidences that show how a hybrid storage system can extend the benefits of a single-level cache by augmenting storage systems with multi-level caches [14]-[18]. Nijim proposed a hybrid prefetching algorithm that can speculatively do prefetching in a hybrid storage system from tapes to hard drives and preload data from hard drives to solid state disks [11] [38]. Their study provided experimental results that show how one can leverage prefetching techniques to enhance I/O performance of hybrid storage systems.

Another study conducted by Zhang *et al*. shows that it is inappropriate to use prefetching algorithms designed for single-level storage systems in hybrid systems [9]. Rather than proposing a new prefetching algorithm for hybrid storage systems, Zhang *et al*. developed their PFC algorithm [9] which forms a hierarchy-aware optimization scheme. They show that their PFC coordinator is applicable to any existing prefetching algorithm because it aims to coordinate prefetching aggressiveness across the cache different levels.

Our solution differs from the existing prefetching and caching techniques in the sense that they make predictive prefetching among hybrid storage system levels.

3. System Design

Compared with existing prefetching schemes, PPHSS supports predictive prefetching mechanism based on probability graph approach across multiple storage devices. Before presenting the PPHSS implementation details, we first outline a high-level overview of PPHSS's hardware and software architecture.

3.1. The Design of Hardware and Software Architectures

The system consists of an application (user) that contentiously issues on-demand I/O reading requests for data blocks that are stored in a parallel hybrid storage system. The hierarchy from top to bottom consists of an array of two levels of storage devices (Solid State Drive (SSD) and Hard Disk Drive (HDD)) as shown in **Figure 2**. As we descend in the hierarchy, both disk read latency and capacity increase. In this research, we will not implement a higher level cache between the application and the hybrid storage system. Our motivation behind that is to test our prefetching mechanism performance alone far away from a higher level caching effect.

In the uppermost level of the hybrid storage system (*i.e.* SDD), there exists a reserved portion (cache) that uses least-recently-used policy (LRU) to cache the application's on-demand I/O reading request and the pre-fetched data from the lowest level (*i.e.* HDD). As we will discuss later, our system will use a fixed size for the data blocks stored in the hybrid storage system. For this reason, the cache size unit is measured based on number of cache blocks. Each cache block has the same size of a data block. Since the data blocks are stripped in the hybrid storage system array, each cache block is also stripped in the SSD level of the disk array in the form of equal size chunks; so each cache block chunk can buffer a prefetched chunk of a particular stripped data block in the HDD level. **Figure 3** shows an example of a 3 disks hybrid storage system array where the SDD cache size equals to 5 cache blocks. Each cache block is stripped equally between the 3 disks. The total size of each stripped block (e.g. 111) equals the data block fixed size. For example, if the data block size equals to 10 MB, then the total cache size will be equal to 50 MB and the chunk size equals to 10/3 MB.

Figure 4 shows PPHSS's software architecture, in which applications issues its on-demand I/O reading requests. PPHSS receives the application' requests in 3 software modules: disk manager, SSD cache manager, and

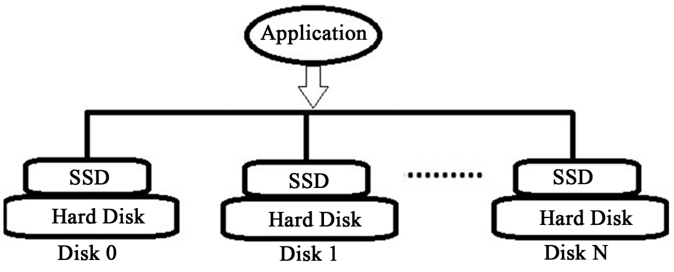

Figure 2. PPHSS's hardware architecture consists of an array of hybrid disks.

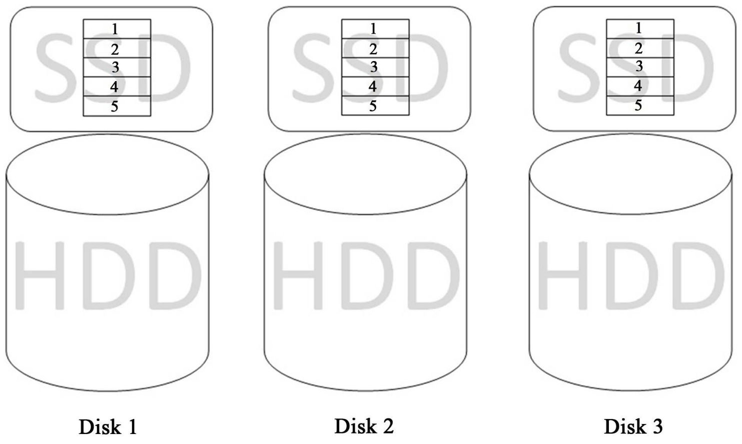

Figure 3. 3 disks hybrid storage system array where the SDD cache size equals to 5 blocks.

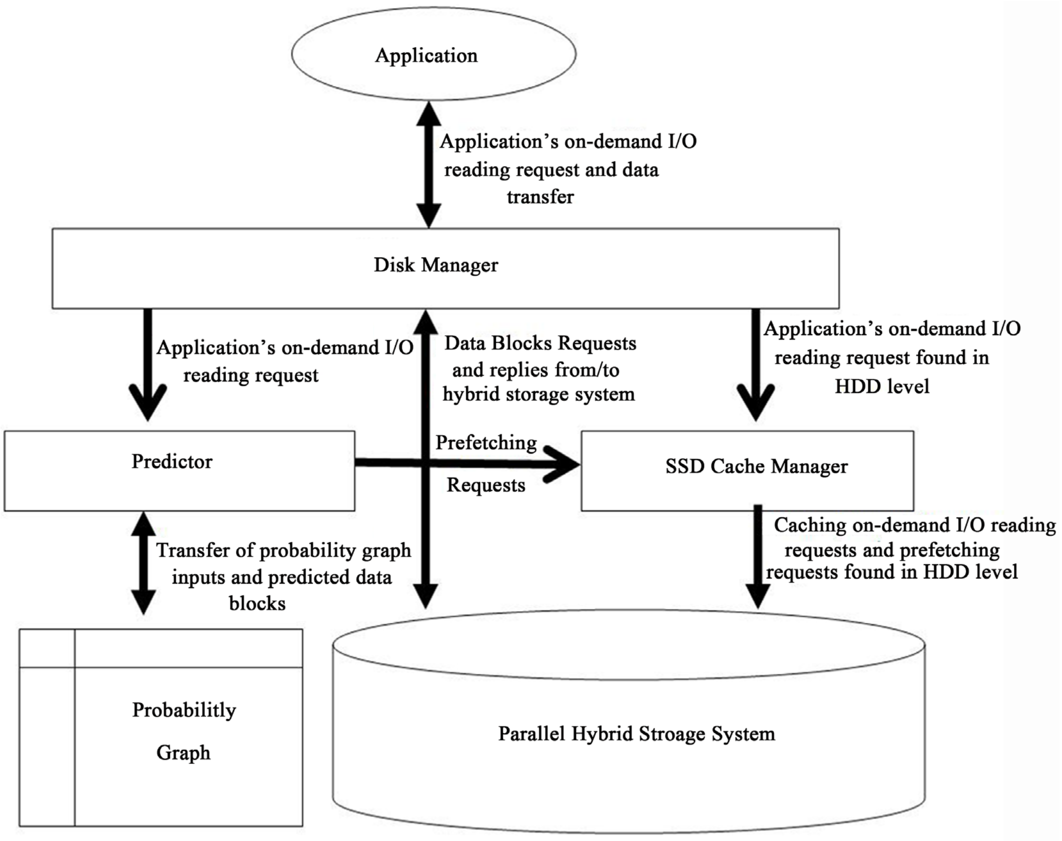
Application
Application's on-demand I/O reading request and data transfer
Disk Manager
Application's on-demand I/O reading request
Data Blocks Requests and replies from/to hybrid storage system
Application's on-demand I/O reading request found in HDD level
Predictor
Prefetching
Requests
SSD Cache Manager
Transfer of probability graph inputs and predicted data blocks
Caching on-demand I/O reading requests and prefetching requests found in HDD level
Probabilitly Graph
Parallel Hybrid Stroage System

Figure 4. Detailed design of PPHSS software architecture for a hybrid storage system. PPHSS implements 3 software modules: disk manager, cache manager, and predictor. PPHSS receives application's on-demand I/O reading requests and provides the application with requested data, perform SSD cache management and prefetching.

predictor. The disk manager searches for the for the application' request in the hybrid storage system starting from the uppermost level. The cache manager controls the cache implemented in the SSD level and applies LRU policy to buffer the application's on demand requests and the prefetched data from the HDD level. The predictor implements the predictive prefetching algorithm approach—(see [3] for details)—that is based on probability graph approach and issues its prefetching request. The prefetching requests go directly to cache manager to read the data from the HDD and cache them in the SDD cache. In case the particular data block is already in the SDD, its prefetching request is discarded. Later, we will show that there is a maximum number of prefetching requests that can be issued at once due to the limited available I/O bandwidth.

3.2. System Assumptions

As we did in [1] [6] [7], we assume a conservative assumption that all data blocks are initially placed in the HDDs thanks to its large capacity compared to SDDs. It is noteworthy that I/O performance of hybrid storage systems can be improved if data blocks are initially placed in SSDs rather than HDDs.

In a hybrid storage system, a small portion (cache) of SSD space is reserved for retaining copies of the prefetched data blocks as well as the application's on-demand I/O reading requests that are found in the HDD. Instead of migrating data blocks from HDD level to SSD cache, PPHSS keeps original copies at the HDD level while fetching duplicated copies to SSD cache.

3.3. Data Initial Placement

As we did in [1] [6] [7], let us consider a simple hybrid storage system, where there are two-level disk array composed of an SSD (top) and an HDD (bottom). All of the data blocks can initially be distributed between the two levels. In our simulation studies, we initially place all data blocks in the HDDs (bottom level). This initial data placement is reasonable because of the following five reasons.

-Normally, HDDs have larger storage capacity than SSDs and; therefore, the probability of finding data in the HDDs is greater than that of finding it in the SSDs.

-Research of prefetching in hybrid storage systems (see previous studies conducted by Nijim and Zhang *et al*.) assume that the bottom level contains less important data [9] [11]. Consequently, prefetching techniques move data likely to be accessed in the future to the uppermost level storage devices. Thus, the probability of finding data blocks in the HDDs is increased.

-The worse case scenario is that all data blocks are initially allocated in the lowest level (e.g., HDDs in our study). If some data were initially allocated in the uppermost level (e.g., SSDs) of the system, the number of prefetching requests would be reduced.

-Having data partially or totally allocated in SSDs does not properly show the performance improvement offered by our solutions. For this reason, we consider a conservative case-the worst-case scenario—in which all the data blocks are initially allocated to the lowest level.

-If a predicted data block is already available in an SSD level, the corresponding prefetching request will be discarded.

3.4. A Prefetching for Data Transfers

In this study, we build a predictive prefetching mechanism to fetch data from HDDs and stores the data to a cache in SSDs in parallel with the application's on-demand I/O reading requests. In our design, the lowest-level storage devices (*i.e.* HDD) always contains original copies of the prefetched data blocks. Our motivations behind this design are:

-Keeping original copies in lower-level storage devices can save storage space in higher-level storage devices, which are more expensive than the lower-level counterparts. As we will see latter, the reserved portion in the uppermost level (*i.e.* SDD) will not consume its storage space.

-In case we have to migrate entire data blocks from HDDs to SSDs, we need to keep moving the data back and forth among multiple-level storage, which consumes the I/O bandwidth.

-Keeping the original copies at the lowest level improves the overall I/O performance of a hybrid storage system as it saves the I/O bandwidth.

3.5. Block Size

As we did in [1] [6] [7], using SSDs and HDDs installed in the servers at our laboratory, we observed that an SSD does not provide better performance than an HDD for small data blocks; SSDs are faster than HDDs when the block size is at least 10 MB.

In addition, our observation found that many file systems pack data into large blocks to improve I/O performance. For example, the data block size in HDFS (Hadoop Distributed File System) is 64 MB [39]. In reference [40], HDFS block size is increased to improve system performance. Hadoop achieves (HAR) tool used to pack several small files into a large one [41], and can be used to create large data blocks out of small ones.

3.6. The Use of LASR Traces

As we did in [1] [6] [7], we are going to develop trace-driven simulator to evaluate performance of our PPHSS system under I/O-intensive workload. Trace used in our experiments represents applications that have small CPU processing time between each two subsequent I/O reading requests. In particular, we will ignore the CPU processing time due to its insignificance. All data blocks in the tested traces have identical block size. More specifically, we use the machine 01 trace (called LASR1) [42] in our simulation studies.

We assume that all the requested data blocks have the same blocks size. We also assume that the application issues each consequent on-demand I/O reading request immediately after the current one is completely read from the level of its placement in the hybrid storage system. This setting allows us to evaluate I/O-intensive cases.

4. The PPHSS Algorithm

Our PHSS makes use of the predictive prefetching algorithm—(see [3] for details)—that is based on probability graph approach to reduce applications' I/O stalls and hence, reducing execution elapsed time. Since the uppermost level of a hybrid storage system is more efficient in terms of speed, our goal is to make the application finds more of its on-demand I/O reading requests in the uppermost level. The probability graph predictive approach records the applications' accesses and builds a history record in order to do predictive prefetching. In this section, we implement the probability graph predictive prefetching approach in a parallel hybrid storage system to perform prefetching among the multiple storage levels in parallel with the application's on demand I/O reading requests.

Our empirical experiments indicate that parallel storage systems may have I/O congestion. Evidence shows that there is a maximum number of read requests being concurrently processed in a parallel storage system. In case the application issues only a single on-demand I/O reading request at a time, unused I/O bandwidth can be allocated to lower-level prefetching mechanisms to bring the predicted data blocks from lower-level to upper-level storage in parallel manner. This makes the application more likely to find its future on-demand I/O reading requests available in the upper-level.

This section presents an algorithm that guides us in implementing the PPHSS mechanism for hybrid storage systems.

4.1. Definitions

PPHSS handles the predictive prefetching process that is based on probability graph approach between SSDs and HDDs (see **Figure 4**). When an application starts its execution, it begins to issue its on-demand I/O reading requests to the hybrid storage system. An application' request is satisfied first from the SSD level. In case, it is not found there, it is satisfied from the HDD one. The application' on-demand I/O reading requests are also directed to the predictive prefetching algorithm module to build a probability graph and to do predictive prefetching from the HDD level to a reserved portion (cache) in the SDD level. The cache used to buffer the prefetching process from the HDD to the SDD as well as the application' on-demand I/O reading requests that were missed in the SDD level. Since the application issues only a single on-demand I/O reading request, the reaming non-utilized portion of the I/O bandwidth is used for the predictive prefetching process. Since we are using fixed size data blocks, each on-demand I/O reading request or each prefetching request is performed for a single data block.

Let Max_BW be the maximum number of read requests that may take place concurrently in the parallel storage system. Since the application only issues a single on-demand I/O reading request at a time, the reaming value of Max_BW is available for predictive prefetching (*i.e.* the predictive prefetching can issue up to (Max_BW-1) prefetching requests from the HDD to the SDD at a time). Our empirical results show that each disk in an array of parallel hybrid storage system can handle a single I/O reading request without causing any congestion. For this reason, we consider Max_BW equals to the number of hybrid disks in the parallel hybrid storage system. The size of the cache that is reserved for predictive prefetching in the uppermost level (*i.e.*, SSDs) is represented by (CacheSize). Since we are using fixed size data blocks, we consider CacheSize value in terms of number of blocks instead of any other storage capacity units. Each cache block is stripped among the SSD level array disks in form of equal size chunks; so each cache block chunk can store a prefetched chunk of a particular stripped data block in the HDD level. In case the CacheSize value is large enough in respect to the total SDD level's aggregated capacity in the hybrid storage system array, predictive prefetching may pollute the SDD level. However, predictive prefething is designed to reduce application' I/O read time when using small size caches [3].

LetsT_cpu represent CPU processing time on each data block. This value may be added to the total time between each two consequent on-demand I/O reading requests. Since our system is oriented for I/O intensive applications, we will ignore the processing time and set it to zero. For this reason, our prefetching process is synchronized with the application's on-demand I/O reading requests. In addition, as our hybrid storage system consists of HDD and SSD levels, let T_hdd be the latency for the application to read a single data block from the HDD, and T_ss be the latency for the application to read a single data block from the SSD. T_ss should be less than T_hdd. Let T_hdd-ss be the total disks latencies to read a single data block from the HDD and to write it to the SDD.

4.2. The Probability Graph Predictive Prefetching Approach

Our PPHSS makes use of Griffioen and Appleton predictive prefetching algorithm—(see [3] for details)—that is based on probability graph approach to reduce applications' stalls and execution elapsed time. Probability graph approach is used to record the application's past access patterns in order to predict future access probabilities. In their model, probability graph data structure involves using directed weighted graph that consists of nodes and edges to estimate access probabilities. Each data block stored in the storage system has a node in the probability graph. Directed weighted edges that make connections among nodes are used to predict the probability that a particular set of data blocks will be accessed in the near future if a particular data block is currently accessed. **Figure 5** shows a typical probability graph data structure.

The predictive algorithm contentiously keeps building the probability graph and predicting future accessed data blocks in parallel with the application's on-demand I/O reading requests. Lookahead period is an important metric that is used to build the probability graph. It determines the relationship between each of the application's consequent accesses. For example, in case the application accessed A,B, and C data blocks and the lookahead period value was equal to 1, then B is probable to be accessed in case A was accessed and C is probable to be accessed in case B was accessed. So, the weight of each of the edges between A to B and B to C are incremented by one. In case that the lookahead period value was equal to 2, then both B and C are probable to be accessed in case A was accessed and C is probable to be accessed in case B was accessed. So, the weight of each of the edges between A to B, A to C, and B to C are incremented by one.

Based on the minimum chance value, the algorithm determines what data blocks to prefetch in case a particular data block was accessed. Edges weights are considered in this mathematics. For example, if the minimum chance equals to 0.5, and assume the following sample of a probability graph: an edge from A to B that weights 1 and another one from A to C that weights 2. So, the total of edges weights equal to 3. In case A was accessed, the algorithm should determine what of B and C should be prefetched by examining the percentage of the edges weights going from A to each of B and C to the total weights of the edges going out from A. The percentage should be at least equals to the minimum chance value. In this case, C will be prefetched because 2/3 exceeds the minimum chance value. Whereas B achieves 1/3 which is less than the minimum chance value.

Both values of lookahead period and minimum chance determine the prefetching aggressiveness. An increased lookahead period value and a decreased minimum chance percentage increase the prefetching aggressiveness. Aggressive prefetching may decrease prefetching accuracy.

4.3. PPHSS Implementation

Every time the application issues an on-demand I/O reading request, the predictive prefetching module continues to build its probability graph and predicts a set of consequent future data blocks that the application may request in the near future. After prediction, it issues prefetching requests (I/O reading requests) for the predicted data blocks in order to be fetched from the HDD to the cache in the SDD. The prefetching process brings a copy of the predicted data blocks that are in HDD to the cache in SDD. In case a data block is already placed in the SDD, no action is taken. For those data blocks in the HDD, it sends a copy of each prefetched data block from the HDD to the SDD cache instead of moving the whole data block; which may pollute the SSD storage capacity. As we mentioned, the number of prefetched data blocks that are predicted by the predictive prefetching algorithm

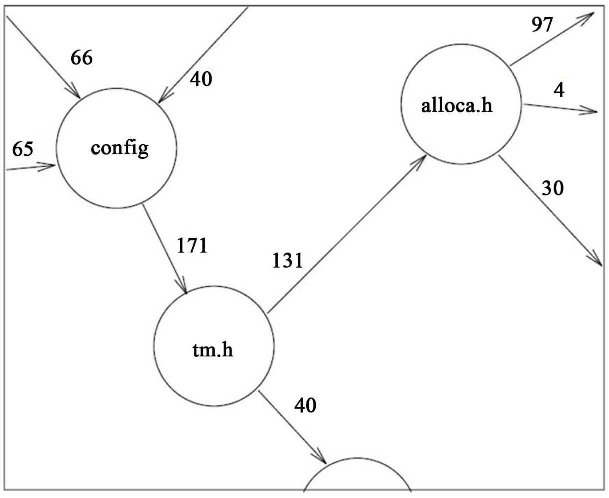

Figure 5. Probability graph data structure that consists of nodes and weighted edges.

depends on the values of minimum chance and the lookahead period. Both of them determine the aggressiveness of the predictive prefetching process. In all cases, our system will allow only the first (Max_BW-1) of the predicted data blocks to be prefetched in each prefetching round. Later, we will show that the number of predicted data blocks to be prefetched will not exceed that amount. Initially, most of the application' on-demand I/O reading requests are found in the the lowest level (*i.e.*, HDD) until the prefetched data blocks arrive in the uppermost one's cache (*i.e.*, SSD). At this point, the application may find each of its requests in the SDD in case the prediction process was accurate and the requested data block was prefetched. In case the application does not find its request in the SDD, the application's request will be satisfied from the HDD. As we will discuss later, both of cache size and prefetching aggressiveness boost the prefetching efficiency. For SSD cache, it caches only the prefetched and the on-demand requested data blocks found in the HDD. It uses least-recently-used (LRU) policy to drop LRU data blocks in order to open the room for new data blocks to be prefetched and cached in the SSD. In case the LRU SSD cache evicts a particular data block that was modified by the application through an I/O writing request, the new copy will overwrite the old one in the HDD.

The application may spend T_cpu CPU processing time on each data block. This may be added to the time between each two consequent on-demand I/O reading requests. Since our system is oriented from I/O intensive applications, we will ignore the processing time as assign it to zero. The following algorithm describes PPHSS algorithm work.

```
Algorithm 1: PPHSS
Input parameters: Max_BW, CacheSize, Minimum Chance, lookahead period.
while Application's on-demand I/O reading request do
        if bandwidth shortage then
                drop the last current prefetching request
        end if
        if prefetched data block is altered by write I/O request then
                discard the prefetched data block from the SSD cache
        end if
provide the application with it's request
if on-demand I/O reading request is found in the HDD then
        update SSD LRU cache
end if
update the probability graph
issue prefetching requests from HDD to SDD cache only for the predicted data blocks that are in HDD and
apply LRU policy
if The number of prefetching requests exceeds Max_BW-1 then
        drop the extra requests.
end while
```

Algorithm 1. The PPHSS algorithm: While the application is doing on-demand I/O reading requests, PPHSS takes each request in the disk manager module. In case the requested data block is found in HDD, PPHSS disk manager provides a copy to the application, invokes the cache manager to cache a copy in the SSD cache and to apply LRU policy. In case the requested data block is found in the SSD, disk manger provides a copy to the application without invoking the cache manager. In all cases, the predictor takes the application's request from the disk manager and invokes the predictive prefetching algorithm to issue prefetching requests for the predicted data blocks that are already placed in the HDD. In coordination with the cache manager, a copy of the prefetched data blocks are cached in the SSD cache. Cache manager helps the predictor to cache the prefetched data in the SDD cache and to apply LRU policy. In case the application requests a data block that is currently in the prefetching process (not fully arrived to the SDD), the request will be satisfied from the HDD. In the event of I/O bandwidth shortage, PPHSS decreases the number of current prefetching requests by one request by dropping the last one. If a data block is currently in prefetching processes from the HDD to the SDD and encountered any write I/O request by the application before its complete arrival to the SSD cache, the prefetched copy will be discarded to preserve consistency. If the block is already cached in the SSD, the prefetching request will be discarded. In case the LRU SSD cache evicts a particular data block that was modified by the application through an I/O writing request, the new copy will overwrite the old one in the HDD. In all cases, PPHSS will not allow the number of prefetching requests to exceed (Max_BW-1).

5. Performance Evaluation

In this section, we are going to show simulation results for our PPHSS system. Before that, we are going to validate our PPHSS system parameters that will be used in the simulation process using data collected from real-world storage systems.

5.1. System Parameters' Validations

In this subsection, we validate system parameters of our simulator using data collected from real-world storage systems.

According to our research lab test-bed, there exists a blocks size where SSD's read performance is better than that of HDD. We also validate SSD and HDD disk reading latencies (*i.e.*, T_ss, T_hdd) for a single data block. We also validate T_hdd-ss latency which is the time spent in reading a single data block from an HDD and write the it back to the SSD. Concerning the time needed for the application to process a single data block (*i.e.*, (T_cpu), we are going to set it to zero in order to make our application very I/O intensive and aggressive in the process of issuing on-demand I/O reading requests.

The following list summarizes the validated system parameters:

-Data block size.

-T_cpu: CPU time to process a single data block.

-T_hdd-ss: Time to fetch a single data block from HDD and to store the block in SSD.

-T_hdd: Time for the application to fetch a single data block from the HDD.

- ss: Time for the application to fetch a single data block from the SSD.

5.1.1. System Setup

All the system parameters used in our simulator are validated by the testbed in our laboratory at Auburn University. The following are storage devices tested in our laboratory:

-Memory: Samsung 3 GB RAM Main Memory.

-HDD: Western Digital 500 GB SATA 16 MB Cache WD5000AAKS.

-SSD: Intel 2 Gb/s SATA SSD 80 G sV 1 A.

5.1.2. Data Block and Cache Block Size

According to our research lab test-bed, our preliminary results based on our storage devices indicate that an SSD is guaranteed to exhibit better performance in term of speed than HDD when the data block size at least 10 MB size for a local system.

In order to support our argument, we run a comparison between the highest read latency of SSD with the lowest of HDD, observing that for small data blocks (e.g., 1 MB), SSD does not show a better performance than HDD. When data block size reaches 5 MB, SSD begins to improve performance over HDD. We choose to use 10 MB as the next point where SSD shows noticeably better performance than HDD. **Figure 6** validates this argument. For this reason, in the subsequent subsections, we consider the data block size equals to 10 MB. As we mentioned previously, cache size is determined in terms of number of cache blocks where each cache block size is the same to a data block one. So, we will consider a cache block size equals to 10 MB.

5.1.3. Modeling T_cpu

Since prefetching research aims to improve I/O performance of I/O-intensive applications, we intend to use small T_cpu value to make our benchmarks I/O intensive and aggressive in performing on-demand I/O read request. Also, the worst case scenario is when there is no processing time between the consequent on-demand I/O reading request. In other words, prefetching should be efficient to satisfy the application' demands. So in our model, T_cpu is set to zero.

5.1.4. Parameters Validation

In this section, we will validate the important system parameters used in simulation studies presented in the subsequent subsections. These system parameters are I/O disk access latencies T_hdd-ss, T_hdd, and T_ss when block size is 10MB.

Figures 7-9 show that T_hdd-ss approximately equals to 0.122 seconds, T_hdd is about 0.12 seconds, and

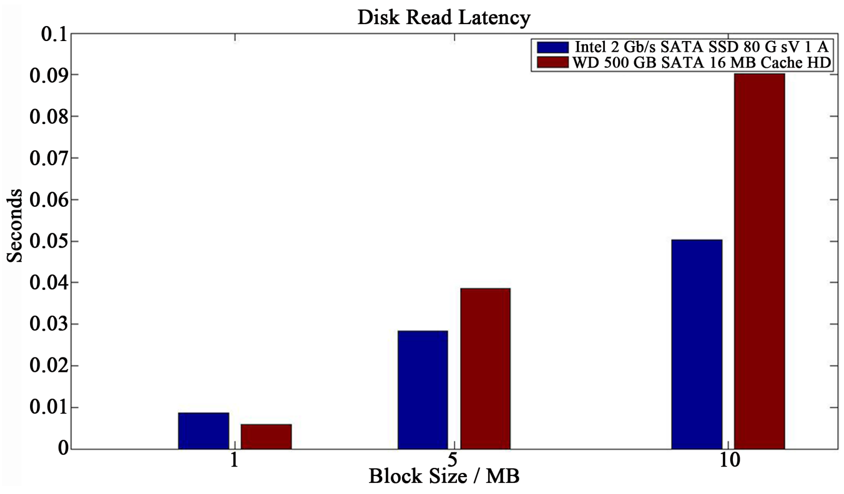

Figure 6. Read Latency of HDDs and SDDs. When block size is 10 MB, SSD has better read performance than HDD.

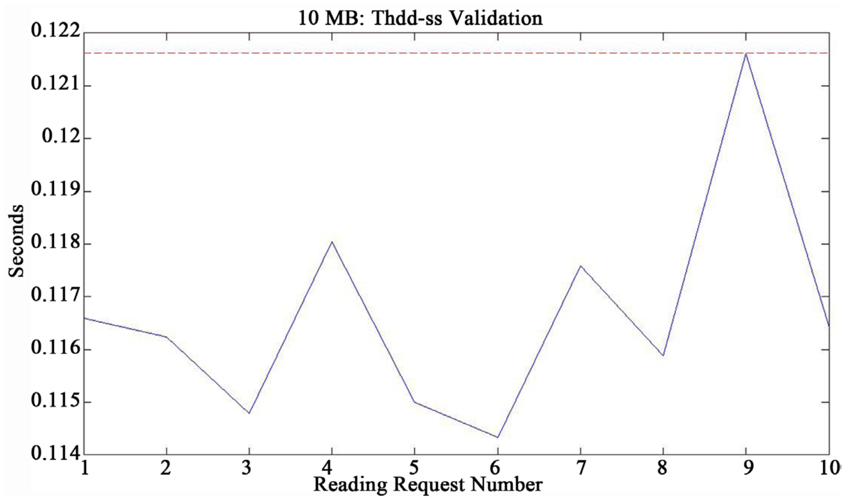

Figure 7. 10 MB: Estimated T_hdd-ss is 0.122 seconds.

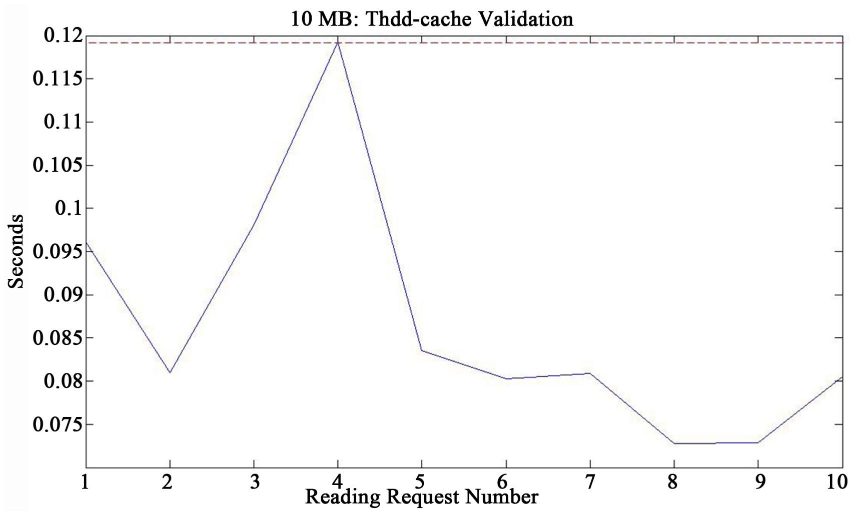

Figure 8. 10 MB: Estimated T_hdd is 0.12 seconds.

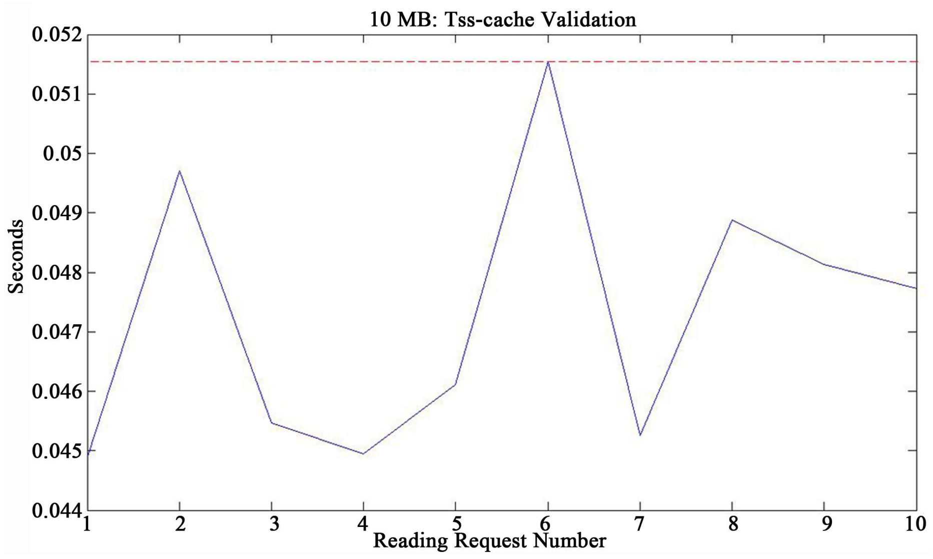

Figure 9. 10 MB: Estimated T_ss is 0.052 seconds.

T_ss is about 0.052 seconds when data block size is 10 MB. We considered the worst case scenario by taking the highest recorded value of latencies.

5.2. The PPHSS Simulation

We implement the PPHSS technique in a trace-driven simulator written in C++. The performance metric evaluated in the simulated two-level (*i.e.*, SSD and HDD) hybrid storage system (see **Figure 2**) is the elapsed time under various SSD cache size, minimum chance, and lookahead period values. A decreased elapsed time shows system performance improvement. For the size of the simulated disk array (*i.e.*, Max_BW value), our empirical results shows that in case we set the SSD cache size to 5 cache blocks (*i.e.* 50 MB cache size), it will be able to improve the system performance even if it might be considered a little size. In case the cache is the stripped in the SSD level, one cache block of size 10 MB in each SDD will not consume its capacity. So, we will consider Max_BW a constant value that equals to 5. As we increase Max_BW value (*i.e.* increase the SSD level array size), SSD level will be more able to host a larger size cache without being polluted.

In this simulation, we first test the system performance without deploying PPHSS. Then, we enable our PPHSS solution and test its performance. When PPHSS is enabled, first we test the effect of increasing the cache size on the system performance. Later, we test the effect of a variety degrees of prefetching aggressiveness on the system performance when using different cache size values.

Data blocks are initially placed in HDDs level. In this simulation, we will use the system parameters that we validated in the previous subsection. Recall that the block size equals to 10MB. The PPHSS mechanism coordinates three modules: the first provides the application with its requested data blocks. The second makes future predictions and fetches predicted data blocks from HDDs to the cache in the SSDs. The third one manages the LRU policy in the SSD cache.

In our experiments, we use a real-world trace-LASR1 from the LASR trace suite [42]. The trace consists of 11686 I/O read system calls. Each I/O request is accessing a data block from the two-level storage system. As a conservative assumption, the average I/O arrival interval between each two consequent requests is the disk read latency for the application' on-demand I/O reading request. In case the current request is found in HDD or in SDD, the subsequent request will be issued after T_hdd or after T_ss respectively. This represents the worst case scenario because if the interval is set to a larger value than T_hdd or T_ss, PPHSS will have more time for prefetching and can achieve even better I/O improvement.

5.2.1. System Performance without Using PPHSS

In case of not implementing PPHSS, the application's on-demand I/O reading requests will be found in the HDD level. This goes to our assumption in section 3 of data initial placement. When using LASR1 trace, the total application's elapsed time without implementing PPHSS will equal to **1401.12 seconds**.

5.2.2. System Performance Evaluation When Using Different Cache Size Values

We simulated our application when using different cache size values that range from 1 to 100 cache blocks. It is a fact that the system performance improves when increasing the cache size regardless the degree of prefetching aggressiveness. In this test, we used a moderate degree of prefetching aggressiveness by setting the lookahead period to 1 and the minimum chance to 0.5. In the next section, we will simulate our system performance when using a variety of lookahead period and minimum chance values. Figures 10-14 show our simulation results when using different cache size values. As the cache size increases, the application's elapsed time decreases. An important observation shows that our PPHSS solution that is based on predictive prefetching approach in a hybrid storage system provides its best performance improvement when using small caches.

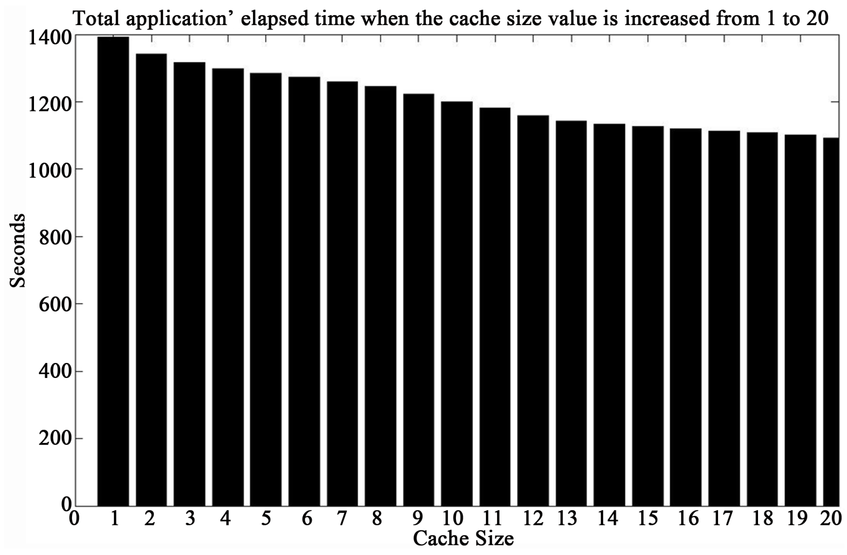

Figure 10. Execution elapsed time of PPHSS in seconds when the cache size value is increased from 1 to 20. Max_BW is set to 5. Lookahead period is set to 1. Minimum chance is set to 0.5.

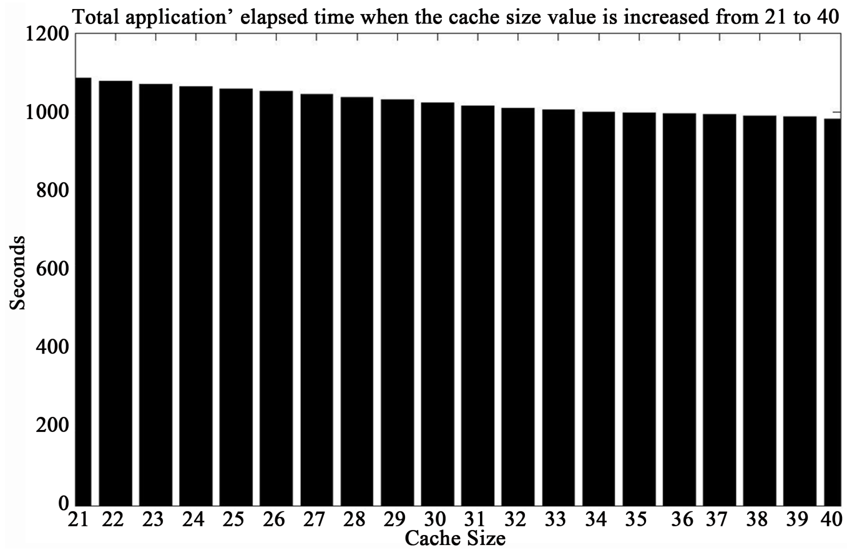

Figure 11. Execution elapsed time of PPHSS in seconds when the cache size value is increased from 21 to 40. Max_BW is set to 5. Lookahead period is set to 1. Minimum chance is set to 0.5.

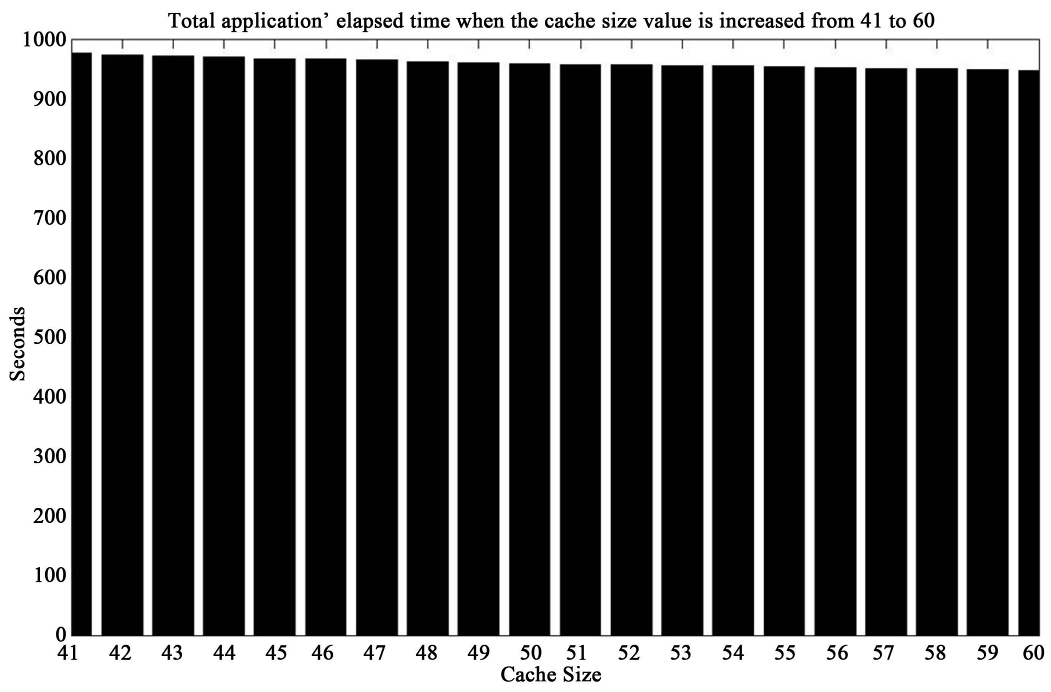

Figure 12. Execution elapsed time of PPHSS in seconds when the cache size value is increased from 41 to 60. Max_BW is set to 5. Lookahead period is set to 1. Minimum chance is set to 0.5.

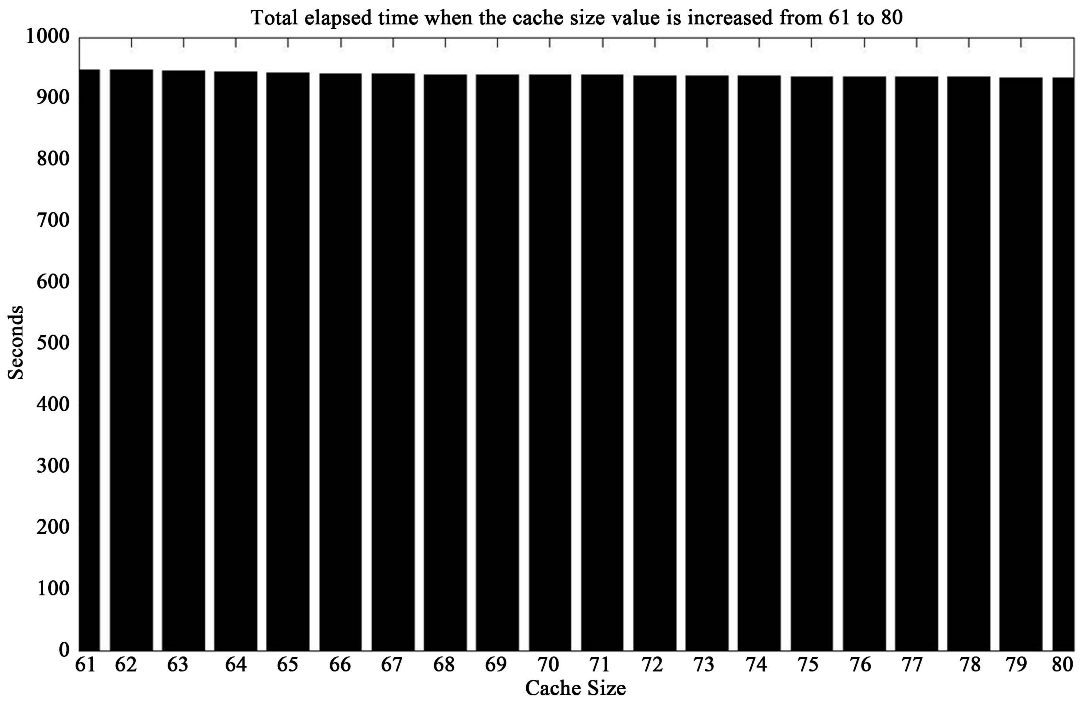

Figure 13. Execution elapsed time of PPHSS in seconds when the cache size value is increased from 61 to 80. Max_BW is set to 5. Lookahead period is set to 1. Minimum chance is set to 0.5.

Figures 10 shows a better performance optimization that the others. In addition, larger caches pollute the SSD level and consume its capacity; but still can improve the system performance. This creates a trade-off between choosing a cache size from small to large.

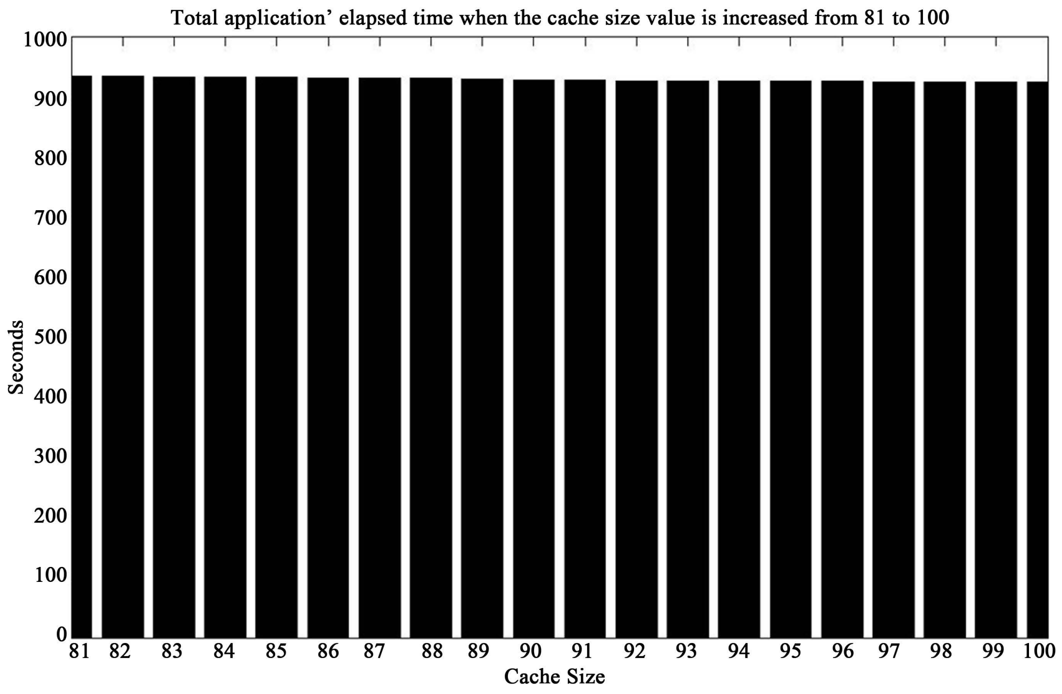

Figure 14. Execution elapsed time of PPHSS in seconds when the cache size value is increased from 81 to 100. Max_BW is set to 5. Lookahead period is set to 1. Minimum chance is set to 0.5.

5.2.3. System Performance Evaluation under Different Degrees of Prefetching Aggressiveness

As mentioned in Section 4, prefetching aggressiveness depends on lookahead period and minimum chance values. More aggressive prefetching can be produced by increasing the lookahead period value and decreasing the minimum chance one. In this section, we evaluate the effect of PPHSS aggressive prefetching on system performance. We will illustrate system performance in terms of application' execution elapsed time with various values of lookahead period values that range from 1 to 3 and different minimum chance values that range from 0.1 to 0.9 when using different chance size values that range from 1 to 100. Max_BW is also set to 5.

Table 1 shows the application's elapsed time when using a cache size that equals to 1. Since the cache size is pretty small, it is noticeable that PPHSS' severe aggressive prefetching (*i.e.* lookahead period = 1, 2, 3 and minimum chance = 0.1) will rapidly replace data blocks in the cache and prefetch unnecessary data. So, PPHSS will not provide a significant system performance improvement. When the prefetching becomes slightly less aggressive, (*i.e.* lookahead period = 2, 3 and minimum chance = 0.2), PPHSS provides its best performance improvement. This is because the cache will become able to keep the frequently accessed data blocks and the prefetching process will provide more accurate decisions. When the lookahead period equals to 2 or 3 and the minimum chance increases to values greater than 0.2, PPHSS performance improvement will gradually decrease. This is because prefetching becomes less aggressive due to the increased minimum chance value, but in the same time, prefetching decisions tend to be less accurate due to the increased lookahead value. So, PPHSS will not be able to maintain in the cache the valuable data blocks that will be repentantly accessed in the near future. When using lookahead period value equals to 2, PPHSS provides a better performance than the case when using a lookahead period value equals to 3. This goes to the enhanced prefetching accuracy provided by the case in which lookahead period value equals to 2. When lookahead period equals to 1, PPHSS shows more sustainable and gradual system performance improvement as the minimum chance value increases. This is because prefetching becomes less aggressive, so the cache can maintain the valuable data blocks for a longer period of time. In the same time, prefetching decisions tend to be more accurate due to the decreased lookahead period value.

When cache size increases to 2 (see: **Table 2**), previous case observations in which cache size was equal to 1 apply. But in this case, cache size becomes slightly larger. So, it can accommodate more cached data blocks even with aggressive prefetching. When minimum chance equals to 0.1, the case in which lookahead period equals to 2 provides the best performance improvement due to prefetching accuracy that is achieved by this case.

The remaining cache size cases (3, 5, 7, 10, 20, 50, 100) (see: **Tables 3-9**) show similar trend in system performance improvement. Since cache size value becomes more significant, the cache becomes more able to accommodate more data blocks. Hence, a performance improvement will be recorded even if non-accurate prefetches were contentiously prefetched. In those cases, aggressive prefetching provides a better system performance because it will result in more cache hits. When the prefetching aggressiveness decreases, prefetching will not be able to bring as much data blocks as possible.

Table 1. Execution elapsed time of PPHSS in seconds when cache size value is equals to 1 with various values of lookahead period that range from 1 to 3 and different minimum chance values that range from 0.1 to 0.9.

Lookahead/MinChance	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
1	1399.42	1398.33	1395.95	1394.73	1392.76	1389.7	1384.8	1374.19	1370.86
2	1399.69	1349.24	1353.72	1358.35	1361.41	1392.55	1392.42	1395.34	1395.68
3	1399.56	1352.09	1371.88	1392.01	1391.46	1392.69	1393.1	1395.54	1396.22

Table 2. Execution elapsed time of PPHSS in seconds when cache size value is equals to 2 with various values of lookahead period that range from 1 to 3 and different minimum chance values that range from 0.1 to 0.9.

Lookahead/MinChance	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
1	1376.71	1373.51	1355.63	1342.98	1342.23	1347.2	1344.61	1338.22	1336.72
2	1317.75	1303.47	1311.22	1316.53	1330.54	1390.17	1390.65	1393.91	1394.18
3	1349.37	1256.35	1280.22	1385.75	1386.64	1389.9	1390.92	1394.8	1395.48

Table 3. Execution elapsed time of PPHSS in seconds when cache size value is equals to 3 with various values of lookahead period that range from 1 to 3 and different minimum chance values that range from 0.1 to 0.9.

Lookahead/MinChance	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
1	1347.81	1351.21	1302.18	1309.93	1316.73	1325.23	1325.1	1320.74	1321.97
2	1249.34	1262.13	1284.57	1296.4	1316.05	1388.06	1389.08	1393.37	1386.5
3	1106.27	1099.88	1185.76	1381.6	1383.64	1387.52	1390.17	1384.87	1385.82

Table 4. Execution elapsed time of PPHSS in seconds when cache size value is equals to 5 with various values of lookahead period that range from 1 to 3 and different minimum chance values that range from 0.1 to 0.9.

Lookahead/MinChance	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
1	1288.24	1263.9	1260.9	1277.77	1283.89	1300	1304.83	1302.32	1305.04
2	1181	1229.08	1261.79	1275.8	1290.48	1380.99	1383.3	1379.97	1379.9
3	979.656	1075.4	1163.46	1371.13	1378.07	1380.72	1380.45	1380.18	1384.46

Table 5. Execution elapsed time of PPHSS in seconds when cache size value is equals to 7 with various values of lookahead period that range from 1 to 3 and different minimum chance values that range from 0.1 to 0.9.

Lookahead/MinChance	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
1	1213.98	1208.68	1208.68	1252.61	1259.07	1275.05	1279.13	1277.63	1268.32
2	1151.22	1212.49	1242.95	1254.38	1268.52	1378.68	1379.22	1378.07	1375.42
3	951.3	1060.17	1137.76	1364.13	1372.36	1378.14	1378.48	1376.57	1377.39

Table 6. Execution elapsed time of PPHSS in seconds when cache size value is equals to 10 with various values of lookahead period that range from 1 to 3 and different minimum chance values that range from 0.1 to 0.9.

Lookahead/MinChance	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
1	1157.48	1159.79	1182.09	1195.42	1199.3	1210.92	1217.38	1211.67	1220.85
2	1126.4	1190.25	1211.4	1225.07	1238.4	1369.09	1375.14	1370.38	1370.04
3	929.744	1036.71	1101.1	1356.78	1363.52	1371.61	1370.45	1370.38	1374.26

Table 7. Execution elapsed time of PPHSS in seconds when cache size value is equals to 20 with various values of lookahead period that range from 1 to 3 and different minimum chance values that range from 0.1 to 0.9.

Lookahead/MinChance	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
1	1015.42	1021.75	1064.38	1084.1	1091.18	1108.72	1116.27	1126.81	1143.74
2	1069.76	1137.21	1156.12	1174.34	1154.76	1347.06	1356.92	1357.12	1359.1
3	884.932	987.816	1054.73	1309.32	1328.29	1354.54	1357.67	1358.82	1359.03

Table 8. Execution elapsed time of PPHSS in seconds when cache size value is equals to 50 with various values of lookahead period that range from 1 to 3 and different minimum chance values that range from 0.1 to 0.9.

Lookahead/MinChance	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
1	877.452	890.1	929.064	944.84	958.168	987.476	1002.78	1018.55	1053.98
2	1018.08	1078.6	1081.45	1098.52	1075.6	1341.76	1351.62	1350.26	1351.96
3	837.876	923.624	973.264	1276.41	1279.26	1340.26	1352.36	1353.11	1353.11

Table 9. Execution elapsed time of PPHSS in seconds when cache size value is equals to 100 with various values of lookahead period that range from 1 to 3 and different minimum chance values that range from 0.1 to 0.9.

Lookahead/MinChance	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
1	824.344	845.9	882.96	907.848	923.216	936.68	948.308	965.988	1007.2
2	994.548	1051.53	1052.96	1060.24	1019.44	1298.3	1295.72	1284.23	1287.36
3	812.308	892.004	935.32	1222.76	1264.3	1285.93	1293.07	1287.02	1287.02

6. Conclusion & Future Work

In this paper, we presented a predictive prefetching technique—PPHSS—for hybrid storage systems. PPHSS makes use of probability graph predictive prefetching approach to fetch predicted data blocks from lower-level to upper-level storage devices in a parallel hybrid storage system. PPHSS aims to boost I/O bandwidth utilization that is under-utilized by the application's on-demand I/O reading requests. Compared with the existing predictive prefetching techniques, our PPHSS approach has the following salient features. First, PPHSS reduces I/O stalls by keeping data blocks that are predicted to be accessed in the near future cached in the uppermost level of the hybrid storage system (*i.e.* solid state disks). Second, a predictive prefetching process in PPHSS woks in parallel with the application on-demand I/O reading requests.

We implemented a simulator for our predictive prefetching schemes. We validated our simulation parameters using a real world computing cluster. Our results show that our PPHSS improves system performance by 4% when using a small cache size and I/O-intensive workload conditions.

The experimental results of this study are very encouraging. We intend to extend the PPHSS scheme to work in a distributed storage systems, where data sets are stored across multiple hybrid storage nodes connected by networks. We plan to conduct more sophisticated experiments to evaluate the performance of our predicative prefetching process under a wide range of I/O workloads and benchmarks.

References

[1] Al Assaf, M.M. Informed Prefetching in Distributed Multi-Level Storage Systems. http://hdl.handle.net/10415/2935

[2] Yang, C.K., Mitra, T. and Chiueh, T. (2002) A Decoupled Architecture for Application-Specific File Prefetching. *Proceedings of the Freenix Track*: 2002 *USENIX Annual Conference*, Monterey, 10-15 June 2002, 157-170.

[3] Griffioen, J. and Appleton, R. (1994) Reducing File System Latency Using a Predictive Approach. *Proceedings of the* 1994 *USENIX Annual Technical Conference*, Boston, 6-10 June 1994.

[4] Patterson Hugo, R., Gibson, G., Stodolsky, D. and Zelenka, J. (1995) Informed Prefetching and Caching. *Proceedings of the* 15*th ACM Symposium on Operating System Principles*, Colorado, 3-6 December 1995, 79-95.

[5] Chen, Y., Byna, S., Sun, X., Thakur, R. and Gropp, W. (2008) Hiding I/O Latency with Pre-Execution Prefetching for Parallel Applications. *Proceedings of the* 2008 *ACM/IEEE Conference on Supercomputing*, Austin, 15-21 November 2008, 1-10.

[6] Al Assaf, M.M., Jiang, X.F., Riduan Abid, M. and Qin, X. (2013) Eco-Storage: A Hybrid Storage System with Energy-Efficient Informed Prefetching. *Journal of Signal Processing Systems*, **72**, 165-180. http://dx.doi.org/10.1007/s11265-013-0784-9

[7] Al Assaf, M.M., Qin, X., Jiang, X., Zhang, J. and Alghamdi, M. (2012) A Pipelining Approach to Informed Prefetching in Distributed Multi-Level Storage Systems. *Proceedings of the* 11*th IEEE International Symposium on Network Computing and Applications* (*IEEE NCA*'12), Cambridge, 23-25 August 2012, 87-95.

[8] Song, J. and Zhang, X. (2004) ULC: A File Block Placement and Replacement Protocol to Effectively Exploit Hierarchical Locality in Multilevel Buffer Caches. *Proceedings of the* 24*th International Conference on Distributed Computer Systems*, Tokyo, 23-26 March 2004, 168-177.

[9] Zhang, Z., Lee, K., Ma, X. and Zhou, Y. (2008) PFC: Transparent Optimization of Existing Prefetching Strategies for Multi-Level Storage Systems. *Proceedings of the* 28*th International Conference on Distributed Computing System*, Beijing, 17-20 June 2008, 740-751. http://dx.doi.org/10.1109/icdcs.2008.89

[10] Thomasian, A. (2006) Multi-Level RAID for Very Large Disk Arrays. *ACM SIGMETRICS Performance Evaluation Review*, **33**, 17-22. http://dx.doi.org/10.1145/1138085.1138091

[11] Nijim, M. (2010) Modelling Speculative Prefetching for Hybrid Storage Systems. *Proceedings of the* 2010 *IEEE* 5*th International Conference on Networking*, *Architecture*, *and Storage*, Macau, 15-17 July 2010, 143-151.

[12] Kaneko, T. (1974) Optimal Task Switching Policy for a Multilevel Storage System. *IBM Journal of Research and Development*, **18**, 310-315. http://dx.doi.org/10.1147/rd.184.0310

[13] Rivera, G. and Tseng, C.-W. (1999) Locality Optimizations for Multi-Level Caches. *Proceedings of the* 1999 *ACM/IEEE Conference on Supercomputing* (*CDROM*), Portland, 14-19 November 1999, Article No. 2. http://dx.doi.org/10.1145/331532.331534

[14] Przybylski, S., Horowitz, M. and Hennessy, J. (1988) Performance Tradeoffs in Cache Design. *Proceedings of the* 15*th Annual International Symposium on Computer Architecture*, Honolulu, 30 May-2 June 1988, 290-298. http://dx.doi.org/10.1109/isca.1988.5239

[15] Przybylski, S., Horowitz, M. and Hennessy, J. (1989) Characteristics of Performance-Optimal Multi-Level Cache Hierarchies. *Proceedings of* 16*th Annual International Symposium on Computer Architecture*, 28 May-1 June 1989, 114-121. http://dx.doi.org/10.1109/ISCA.1989.714545

[16] Jason, F. (2002) Multi-Level Memory Prefetching for Media and Stream Processing. *Proceedings of the IEEE International Conference on Multimedia and Expo*, St. Louis, 26-29 August 2002, 101-104.

[17] Kang, J. and Sung, W. (2001) A Multi-Level Block Priority Based Instruction Caching Scheme for Multimedia Processors. *Proceedings of the* 24*th International Conference on Distributed Computer Systems*, Antwerp, 125-132.

[18] Jiang, X.F., Al Assaf, M.M., Zhang, J., Alghamdi, M.I., Ruan, X.J., Muzaffar, T. and Qin, X. (2013) Thermal Modeling of Hybrid Storage Clusters. *Journal of Signal Processing Systems*, **72**, 181-193. http://dx.doi.org/10.1007/s11265-013-0787-6

[19] Lewis, J., Alghamdi, M.I., Assaf, M.A., Ruan, X.-J., Ding, Z.-Y. and Qin, X. (2010) An Automatic Prefetching and Caching System. *Proceedings of the* 29*th International Performance Computing and Communications Conference* (*IPCCC*), Albuquerque, 9-11 December 2010, 180-187. http://dx.doi.org/10.1109/PCCC.2010.5682310

[20] Vellanki, V. and Chervenak, A.L. (1999) A Cost-Benefit Scheme for High Performance Predictive Prefetching. *Proceedings of the ACM/IEEE SC*99 *Conference on Supercomputing*, Portland, 14-19 November 1999.

[21] Chen, Y., Byna, S. and Sun, X. (2007) Data Access History Cache and Associated Data Prefetching Mechanisms. *Proceedings of the AMC/IEEE Conference on Supercomputing*, Reno, 10-16 November 2007, 1-12. http://dx.doi.org/10.1145/1362622.1362651

[22] Wang, J.Y.Q., Ong, J.S., Coady, Y. and Feeley, M.J. (2000) Using Idle Workstations to Implement Predictive Prefetching. *Proceedings of the* 9*th IEEE International Symposium on High Performance Distributed Computing*, Pittsburgh, 1-4 August 2000, 87-94.

[23] Domenech, J., Sahuquillo, J., Gil, J.A. and Pont, A. (2006) The Impact of the Web Prefetching Architecture on the Limits of Reducing User's Perceived Latency. *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence*, Hong Kong, 18-22 December 2006, 740-744.

[24] Jeon, J., Lee, G., Cho, H. and Ahn, B. (2003) A Prefetching Web Caching Method Using Adaptive Search Patterns. *Proceedings of the* 2003 *IEEE Pacific Rim Conference on Communications*, *Computers*, *and Signal Processing*, Victoria, 28-30 August 2003, 37-40.

[25] James, O. and Daniel, A.R. (2002) Markov Model Prediction of I/O Requests for Scientific Applications. *Proceedings of the* 16*th International Conference on Supercomputing*, New York, 22-26 June 2002, 147-155. http://dx.doi.org/10.1145/514191.514214

[26] Nanopoulos, A., Katsaros, D. and Manolopoulos, Y. (2003) A Data Mining Algorithm for Generalized Web Prefetching. *IEEE Transactions on Knowledge and Data Engineering*, **15**. http://dx.doi.org/10.1109/TKDE.2003.1232270

[27] Hugo Patterson, R., Gibson, G.A. and Satyanarayanan, M. (1993) A Status Report on Research in Transparent Informed Prefetching. *ACM SIGOPS Operating Systems Review*, **27**, 21-34. http://dx.doi.org/10.1145/155848.155855

[28] Tomkins, A., Hugo Patterson, R. and Gibson, G. (1997) Informed Multi-Process Prefetching and Caching. *Proceedings of the* 1997 *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Seattle, 15-18 June 1997, 100-114. http://dx.doi.org/10.1145/258612.258680

[29] Hugo Patterson, R., Gibson, G.A. and Satyanarayanan, M. (1992) Using Transparent Informed Prefetching (TIP) to Reduce File Read Latency. *Proceedings of the Conference on Mass Storage Systems and Technologies*, Greenbelt, 22-24 September 1992, 329-342.

[30] Hugo Patterson, R. and Gibson, G.A. (1994) Exposing I/O Concurrency with Informed Prefetching. *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, Austin, 28-30 September 1994, 7-16. http://dx.doi.org/10.1109/PDIS.1994.331737

[31] Huizinga, D.M. and Desai, S. (2000) Implementation of Informed Prefetching and Caching in Linux. *Proceedings of the International Conference on Information Technology*, Las Vegas, 27-29 March 2000, 443-448.

[32] Chen, Y., Byna, S., Sun, X., Thakur, R. and Gropp, W. (2008) Exploring Parallel I/O Concurrency with Speculative Prefetching. *Proceedings of the* 2008 37*th International Conference on Parallel Processing*, Portland, 8-12 September 2008, 422-429. http://dx.doi.org/10.1109/icpp.2008.54

[33] Kimbrel, T., Cao, P., Felten, E., Karlin, A. and Li, K. (1996) Integrated Parallel Prefetching and Caching. *Proceedings of the* 1996 *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Philadelphia, 23-26 May 1996, 262-263. http://dx.doi.org/10.1145/233013.233052

[34] Ganger, G.R., Worthington, B.L., Hou, R.Y. and Patt, Y.N. (1994) Disk Arrays: High-Performance, High-Reliability Storage Subsystems. *Computer*, **27**, 30-36. http://dx.doi.org/10.1109/2.268882

[35] Chang, F. and Gibson, G.A. (1999) Automatic I/O Hint Generation through Speculative Execution. *Proceedings of the* 3*rd Symposium on Operating Systems Design and Implementation*, New Orleans, 22-25 February 1999, 1-14.

[36] Byna, S., Chen, Y., Sun, X.-H., Thakur, R. and Gropp, W. (2008) Parallel I/O Prefetching Using MPI File Caching and I/O Signatures. *Proceedings of the* 2008 *ACM/IEEE Conference on Supercomputing*, Austin, 15-21 November 2008.

[37] Li, C.P., Shen, K. and Papathanasiou, A.E. (2007) Competitive Prefetching for Concurrent Sequential I/O. *Proceedings of the* 2*nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, Lisbon, 21-23 March 2007, 189-202. http://dx.doi.org/10.1145/1272996.1273017

[38] Nijim, M., Zong, Z., Qin, X. and Nijim, Y. (2010) Multi-Layer Prefetching for Hybrid Storage Systems: Algorithms, Models, and Evaluations. *Proceedings of the* 2010 39*th International Conference on Parallel Processing Workshops*, San Diego, 13-16 September 2010, 44-49. http://dx.doi.org/10.1109/ICPPW.2010.18

[39] Borthakur, D. (2008) HDFS Architecture, the Apache Software Foundation. http://hadoop.apache.org/docs/

[40] Shafer, J., Rixner, S. and Cox, A.L. (2010) The Hadoop Distributed Filesystem: Balancing Portability and Performance. *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software* (*ISPASS*), White Plains, 28-30 March 2010, 122-133. http://dx.doi.org/10.1109/ISPASS.2010.5452045

[41] Hadoop Archive Guide. http://hadoop.apache.org/docs/

[42] Lasr Trace Machine 01. http://iotta.snia.org/