HFetch: Hierarchical Data Prefetching for Scientific Workflows in Multi-Tiered Storage Environments

Hariharan Devarajan
Department of Computer Science
Illinois Institute of Technology
hdevarajan@hawk.iit.edu

Anthony Kougkas
Department of Computer Science
Illinois Institute of Technology
akougkas@iit.edu

Xian-He Sun
Department of Computer Science
Illinois Institute of Technology
sun@iit.edu

***Abstract*—In the era of data-intensive computing, accessing data with a high-throughput and low-latency is more imperative than ever. Data prefetching is a well-known technique for hiding read latency. However, existing solutions do not consider the new deep memory and storage hierarchy and also suffer from under-utilization of prefetching resources and unnecessary evictions. Additionally, existing approaches implement a client-pull model where understanding the application's I/O behavior drives prefetching decisions. Moving towards exascale, where machines run multiple applications concurrently by accessing files in a workflow, a more data-centric approach can resolve challenges such as cache pollution and redundancy. In this study, we present HFetch, a truly hierarchical data prefetcher that adopts a server-push approach to data prefetching. We demonstrate the benefits of such an approach. Results show 10-35% performance gains over existing prefetchers and over 50% when compared to systems with no prefetching.**

***Index Terms*—hierarchical, multi-tiered, prefetching, middleware, server-push, data-centric**

I. Introduction

Data-intensive computing offers unprecedented opportunities for scientific discovery, high-fidelity insights, and data-driven decision making with timely data access being a driving factor of the overall execution time [1]. The computational efficiency of modern applications is closely related to the ability of the storage systems to push data to the compute units as the performance of the latter has progressed significantly faster than disk capabilities [2], [3]. Modern applications spend significant amounts of time in reading data; in some cases up to 80% of the overall execution time [4]. As we move towards exascale, this trend is expected to exacerbate further the I/O bottleneck (i.e., I/O wall problem [5], [6]). While modern storage systems are adapting quickly to the challenges of today's fast-paced computation environment, they still struggle to address the demand for low-latency data access.

To address the gap between the data consumption from the compute and data supply from the storage system, recent research has proposed solutions broadly categorized in two relatively orthogonal directions: a) new hardware devices that offer lower access latency and higher throughput, and, b) middle-ware software that sits between applications and storage and is responsible to mask the access latency. Many of the leadership computing facilities have already deployed fast node-local NVMe devices and/or shared specialized buffering nodes [7] (i.e., burst buffers) creating a new multi-tiered storage environment, called deep memory and storage hierarchy (DMSH) [8]. Many supercomputing facilities have widely deployed specialized buffering solutions such as Cray's Datawarp [9] and DDN's IME [10]. However, traditional file systems are not equipped to handle this new hierarchy and users are left to manually manage the layers of the hierarchy [11]. To handle data movement through the hierarchy, some software platforms such as Data Elevator [12], Univistor [13], and Hermes [8] have been developed. All the above systems are designed to optimize only write-heavy workloads via data buffering. Therefore, the read operation optimizations, that leverage the DMSH and elevate the caching effect, have to be further explored.

Typically, all read operations are going through a caching layer the operating system implements [14]. Read-caches are limited in capacity and suffer from cache pollution, cache redundancy, and unnecessary evictions of cached data in a multi-process environment. Further, a read-cache lifetime is coupled with the scope of the application. To optimize the caching layer effectiveness, data locality (both spatial and temporal) needs to improve. There are several software techniques that aim to optimize read caching, each with their advantages and disadvantages. Data concurrency, via replication [15], boosts the availability of data for reads (i.e., spatial data locality in cache), with more copies to read from, at the cost of additional data movements and loss of storage capacity. Data locality, via careful data partitioning [16], can boost read performance by reducing interference at the cost of data transformations and recompilation. I/O reordering [17] techniques re-arrange I/O calls to be executed earlier in the code, but require source code access and compilation which may not always be possible. Data staging [18] is another technique that pre-loads data in a set of staging resources before the application starts at the cost of additional dedicated hardware (i.e., allocated staging memory or special staging nodes). Lastly, data prefetching [19] reads data ahead of a read operation to hide the gap between I/O and compute. The effectiveness of data prefetching depends upon the ability to recognize data access patterns and to timely identify the data which should be prefetched. Therefore, both timeliness and accuracy are critical in the perceived performance of a data prefetcher. The latter three optimizations aim to boost the temporal data locality in the read-cache.

In this study, several significant challenges came to light

1530-2075/20/$31.00 ©2020 IEEE
DOI 10.1109/IPDPS47924.2020.00017

when optimizing read operations by using the existing methods. *Firstly*, a truly hierarchical data prefetching in a multi-tiered storage environment is either partially or not supported by existing solutions. Prefetching algorithms aim to reduce cache misses [20] by correctly predicting the next access and proactively fetching the appropriate data. Hence, all prefetching solutions have to answer two main questions [21]: a) when to prefetch data, and b) what data to prefetch. Prefetching the wrong data or the right data at a wrong time not only does not help but actually hurts the overall performance [22]. Additionally, the presence of multiple tiers of the storage hierarchy raises a third question: *where to prefetch data*? Modern architectures, in extreme scale computing, suggest a decrease in the amount of RAM per core. To avoid overwhelming the precious main memory, data staging and prefetching, that are memory-based, need to evolve and include all available layers of the deep memory and storage hierarchy. *Secondly*, existing prefetching solutions rely on identifying application's data access patterns (e.g., via tracing, source code analysis, prediction models, or machine learning) which poses several issues: a) runtime changes of access patterns due to dynamic inputs or linked libraries, b) unwanted data evictions and prefetching cache pollution when one application's prefetched data collide with another's, c) prefetching cache redundancy when different applications prefetch the same data, and, d) resource contention when multiple applications prefetch data in an uncoordinated fashion. To alleviate these issues, an application-agnostic approach should be employed. *Thirdly*, prefetching data in a smaller granularity than the entire file is unavailable or, at best, limited. Finer granularity can lead to better prefetching resource utilization and, therefore, higher performance. *Lastly*, several existing solutions require increased user involvement by passing hints, which is undesirable since it assumes that users know the behavior of their application. As the complexity of computation workflows increases, identifying data access patterns becomes infeasible.

To address the above challenges, we present HFetch, a new hierarchical data prefetcher that supports multi-tiered storage environments. HFetch is primarily a data-centric prefetching decision engine that utilizes system-generated events, while leveraging the presence of multiple tiers of storage, to perform timely hierarchical data placement. HFetch can obtain a global system-wide view of how data is accessed, regardless of which application or process is performing the access, by monitoring the file system and collecting statistics for each data segment. Based on a global scoring function that ranks the importance or urgency of the targeted data, it makes intelligent decisions as to when, what, and where to prefetch data. We build upon the observation that scientific workloads demonstrate a WORM data access model (i.e., write-once-read-many) [4], which is also true for BigData applications [23]. We also target modern scientific workflows that span across multiple applications in a pipeline of data processing. In a multi-process, multi-application workflow, data might be read multiple times by many processes (or across applications) which might create issues for prefetching cache management. Cache pollution, cache redundancy, and unnecessary data evictions leading to increased miss ratios are the norm, and not the exception, especially in extremely large scale workloads. HFetch addresses these issues by maintaining global file heatmaps that represent how a file is accessed across processes or applications. It uses those heatmaps to express the placement of data in a hierarchical system. For instance, hotter data are fetched in more capable tiers (e.g., DRAM) and colder data in lower ones (e.g., burst buffers). The contributions of this work are: a) Demonstrating the importance of incorporating multiple tiers of storage when performing data prefetching optimizations (Subsection III-B). b) Providing evidence that a server-push model can achieve better data-prefetching performance by leveraging a global view of how data is accessed across multiple applications. (Subsection III-C). c) Showcasing that a data-centric prefetching approach solves several issues caused by a growing set of application-specific optimizations (Subsection III-D).

II. Background and Motivation

A. Multi-tiered Storage Systems

I/O systems are going through an extensive transformation by adding multiple levels of memory and storage in a hierarchy [24]. For example, Cori system at the National Energy Research Scientific Computing Center (NERSC) uses CRAYs Datawarp technology [9]. Each tier of the hierarchy, being an independent system, requires different expertise to manage [11]. This significantly increases the complexity of the data movement among the layers. The problem of dealing with the complexity is left to the users. While some solutions [9], [10], [25], [12], [13], [8] handle data movement through multiple tiers for storage, they mostly focus on write operations. However, the main limitation that these systems demonstrate is the lack of efficient support for read data access optimizations such as native hierarchical prefetching [11]. For instance, in DataWarp, reading data relies on a *stage-in* function, provided to the users, loading the entire dataset from the parallel file system (PFS) onto the buffers. This increases the end-to-end time and assumes that the dataset can fit into the buffers, an unrealistic assumption in data-intensive computing. Similarly, in data staging services staged-in data are assumed to fit in memory (e.g., RAM) of the staging servers, which is again unrealistic. Recently proposed work [26], [4] aims to tackle this issue. They rely on bringing the correct data into the faster layer (e.g., burst buffer or RAM) by utilizing a prefetching prediction engine. In essence, these solutions still don't answer *where* to prefetch (i.e., various placements of prefetched data into the DMSH), disregarding the pipelining opportunities from a lower to a higher layer of the hierarchy. Therefore, a truly hierarchical data prefetching solution, that aims to accelerate reading data by utilizing multiple tiers of the storage hierarchy, is still required.

B. Accelerating Read Access Time

Hardware prefetchers [27], [28], [29] move data through the main memory into the CPU caches to increase the hit ratio thereby increasing data locality. The granularity of a hardware

prefetcher is a cache-line, and the trigger is executed per-core. Locality-aware prefetching (i.e., read-ahead approach) is a common implementation where once a memory page is accessed, the prefetcher brings the next page into the caches (temporal and spatial locality). The ability to detect strided patterns is also present in most modern CPU architectures [30], [31]. However, if the application demonstrates irregular patterns, then the miss ratio is high, and applications experience performance degradation due to the contention in memory bus between the normal memory access and the prefetcher. Lastly, a memory page is a well defined prefetching unit while the same cannot be said for I/O where file operations will be variable-sized. Software-based solutions leverage information collected from the application to perform data prefetching and can be broadly categorized into:

1) Offline Data Prefetchers: This category of prefetchers involve a pre-processing step, where an analysis of the application determines the data access patterns, and devise a prefetching plan. There are several different ways to perform this pre-execution analysis and several ways to devise a prefetching plan. In trace-driven [32] prefetching, the application runs once to collect execution and I/O traces. These are then analyzed to generate prefetching instructions. This method offers high accuracy in prefetching but requires significant user-involvement and poses large offline costs. More importantly, a trace-driven approach suffers from the fact that an application's I/O behavior is subject to change at runtime since applications may include third party libraries resulting in a mismatch between application's I/O calls and what the servers experience [33], [3]. Similar to trace-driven approach, a history-based [22] prefetcher stores the seen accesses in a previous run of an application into a database, and, thus, access patterns are known when the same application executes again in the future. While this method decreases the level of user involvement and the cost of trace analysis, it assumes that the application's behavior remains stable between executions which is unrealistic since applications demonstrate different access patterns when run with different inputs [34]. Compiler-based prefetching utilizes the source code structure and modifies it to add prefetching instructions either by I/O re-ordering [35] or by hint-generation [21] to provide the information to the prefetcher about when and what to prefetch. The code is then re-compiled and executed with the extra prefetching instructions. This approach avoids the increased offline costs, since it does not require any execution of the application, but suffers from miscalculations of placing the prefetching calls, as code-flows are often dynamic in nature [36]. Lastly, data staging [18] leads to high hit ratios, but it assumes that the working set can fit in the staging resources capacity.

2) Online Data Prefetchers: This category of prefetchers trade accuracy for a "learn as you go" model. The application's access patterns are learned as the execution proceeds, avoiding any pre-processing steps. Statistical methods such as hidden Markov models (HMM) [37] and ARIMA models [38] require a large number of observations to accomplish model convergence. Once the model has converged, it can predict the next access and trigger prefetching. However, they often focus exclusively on either spatial or temporal I/O behaviors and need long execution time or several runs to achieve accurate predictions. A grammar-based model [39], [4] relies on the fact that I/O behaviors are relatively deterministic (inherent from the code structure) and predicts when and what future I/O operations will occur. However, this method demands repetitive workloads and does not work well for irregular access patterns. *Lastly*, machine learning approaches [40], [26] have been recently proposed where a model learns the data access pattern and uses it to drive the prefetching plan. All online approaches share the fact that they do not rely on a priori knowledge of the application's data access patterns or user inputs and hints. The problem is that they require a warm-up period at the beginning of the execution as they build their models, which can result in added overheads and low performance.

The common theme for all existing approaches is that they implement a client-pull model. Prefetching is driven by the applications' data access patterns. In a multi-tenant environment, application-specific optimizations will not perform due to a lack of global coordination. Prefetching cache space will be limited and shared, leading to cache pollution, cache redundancy, and unwanted evictions. Application-bound prefetching resources will be competing with one another, leading to loss of performance due to interference. We need to address the challenges of read-optimizations from a data-centric view. A server-push model can obtain a global view and apply optimizations on the most valuable pieces of data. None of the existing data prefetching solutions fully utilize the hierarchical environment which can elevate the reduction of RAM per core. The hardware is there, but we need to design software to drive performance by masking access latency behind the DMSH. An application-centric prefetching approach can lead to a loss in performance and under-utilization of the prefetching resources [41].

III. Hierarchical Data Prefetching

In this work, we propose a system-wide, data-centric, server-push prefetching solution that aims to identify how files are accessed, regardless of which process or application does the data access, and utilize this information to pre-load the data needed into the deep memory and storage hierarchy. We designed and implemented a new hierarchical data prefetcher, called HFetch [42], that optimizes read operations by leveraging two main observations: a) the presence of multi-tiered storage suggests a feasible solution to the shrinking DRAM size per-core; a prefetching solution that utilizes the storage hierarchy to fetch data in a pipeline fashion is needed, and, b) identifying an application's data access pattern will not suffice in the age of data-intensive computing; a global view of how files are accessed across a workflow is needed to place prefetched data at the right tier of the hierarchy.

A. HFetch Overview

The main idea behind HFetch is to fetch portions of a file and place them in a tier of the hierarchy based on access

frequency, recency, and the relationship between segments (i.e., file segment sequencing). In other words, instead of guessing what an application will access next, HFetch collects access statistics of file regions (which we call file segments) from the file systems themselves and pro-actively loads them in the hierarchy based on a segment score that reflects the urgency to access the chosen segment. This score basically incorporates the frequency with which the segment is accessed across processes or applications thereby creating a file access heatmap. The file heatmap is then used to naturally match it to a hierarchical environment. Segment movement between tiers of the hierarchy is also based on how recently a segment was accessed. In effect, HFetch answers the three prefetching questions (what to prefetch, when to prefetch, and where to place prefetched data) indirectly by naturally mapping the spectrum of segment frequency and recency to the appropriate tier leveraging the hardware capabilities of each tier. We have designed HFetch with the following goals in mind:

1) Hierarchy-aware: the prefetcher should be able to fetch and place data in all available layers of a multi-tiered environment. Effectively, this should relieve the pressure of in-memory prefetching by overflowing data to lower tiers, or automating data movement between tiers thereby presenting the hierarchy as one big prefetching cache.

2) Application-agnostic: the prefetcher should be able to fetch the right data in a server-push model regardless of who needs the data. In fact, the prefetching flow should be de-coupled from the running applications and must maintain a global view of what data need to be accessed next.

HFetch aims to optimize complex scientific workflows where a collection of data producers (i.e., simulations, static data sources, etc.) send data down a pipeline and a collection of consumers (i.e., analytics, visualization) process the data multiple times. HFetch's design fits naturally in such environment with hierarchical data prefetching boosting read operations across all data consumers.

Figure 1 shows the architecture of HFetch. HFetch follows a client-server model. Each compute node is equipped with an HFetch server. Each application dynamically links to the HFetch library with an *Agent* (depicted as the H). Upon application initialization (e.g., $MPI\_Init()$), client connection to the server core are established. Our proposed architecture incorporates modern extreme scale system designs with a local NVMe drive, a shared collection of burst buffer nodes, and a remote parallel file system. A connection to all available hardware tiers is also established by the HFetch server (i.e., mounting points, prefetching memory allocations, and burst buffer leases). Essentially, the flow of operations in HFetch is as follows. Each tier independently pushes its I/O events into a queue that resides in HFetch Server memory. A hardware monitor collects events (i.e., consumes the queue) and passes them to the file segment auditor who calculates statistics for each file segment. An engine periodically devises a data placement plan in the hierarchy using each segment's attributes and pushes it to the I/O clients to be executed while updating the segment mapping (i.e., segment-to-tier location) in the

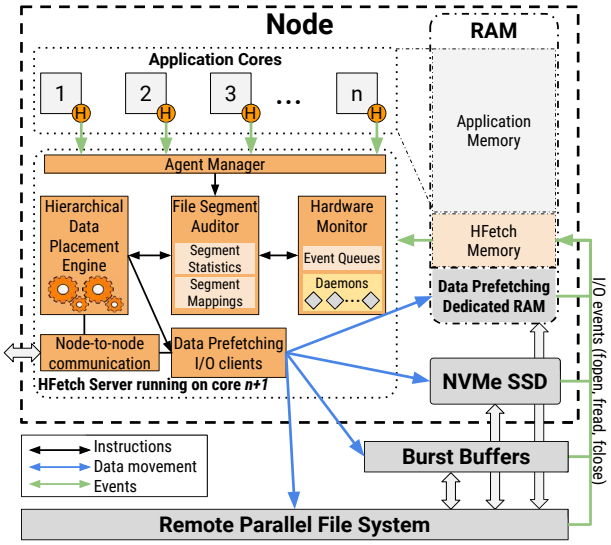


Fig. 1. HFetch overview.

auditor. HFetch server consists of the following components:

1) Hardware Monitor: Its main role is to monitor all available hardware tiers. During initialization of the HFetch server, the hardware monitor is tasked to discover all available tiers of the hierarchy (using a user-defined configuration) and to keep track of each tier's events. The events are generated by the system and are pushed to an in-memory event queue which is served by a pool of daemon threads. In HFetch context, events are either file accesses or tier remaining capacity. All collected events are then passed on to the file segment auditor. In the face of update events, HFetch invalidates the previously prefetched data enforcing data consistency.

2) File Segment Auditor: Its main role is to calculate file segment statistics. Specifically, for each file segment (i.e., a region of file) the auditor calculates its access frequency, when was it last accessed, and which segment access preceded it. Using this information, the auditor can construct a score for each file segment that reflects how hot the segment is in the prefetching context. A hot segment is one that is accessed many times in a recent time window. The sequencing of segments also provides a logical map of which segments are connected to one another. Lastly, the auditor maintains the file segment mappings (i.e., where in the hierarchy each segment has been prefetched) acting as a type of internal metadata manager. Segment statistics and mappings are both maintained in a distributed hashmap we have developed [43]. The details of this distributed hashmap are out of the scope of this paper and therefore skipped. This hashmap provides a uniform and fast $O(1)$ insertion and querying capability, support for concurrent access, fault tolerance in case of power-downs, and low latency. HFetch has the ability to maintain a global view of how each file is accessed, while avoiding a global synchronization barrier, by using this distributed hashmap. For example, based on the starting offset and the length of a read request, the auditor will atomically update one or more targeted segments'

score in the map. This update will be visible across all nodes. Removing the distributed hashmap from HFetch's design will result in increased latencies since for each read request the auditor would need to propagate the update of segment statistics across the cluster, a prohibitively expensive operation.

3) Hierarchical Data Placement Engine: Its main role is to devise a file segment placement plan. The prefetching of file segments and their placement in DMSH are triggered by the change of segment scores and not by the application's accesses, realizing HFetch's data-centric approach. Effectively, the engine maps file segments to the layers of the hierarchy based on their score spectrum. Once a placement plan has been devised, the engine passes it to the I/O clients to execute the actual fetching across and between the tiers. Additionally, the engine utilizes the node-to-node communicator to include remote tiers in its data placement plan (i.e., tiers can be local or remote based on the performance of each tier).

4) Agent Manager: Its main role is to initialize the client connections through a $PMPI$ wrapper. During this connection, the agents link application with HFetch server for sharing prefetch data. Each application process is attached to an HFetch agent who talks to the agent manager to acquire the location of the prefetched file segments for each read request. Also, the agent manager is responsible to collect from the agents the beginning and the end of a prefetching epoch enclosed between a file open and file close calls and pass it to the auditor who marks the appropriate file segments that are targeted for prefetching. The Agent is able to intercept POSIX, MPI-IO, and HDF5 open-close calls.

5) Data Prefetching I/O Clients: For each available hardware tier there is a thread that is responsible to perform the I/O calls to and from source tier and destination tier. These I/O clients can also participate in collective I/O operations across multiple nodes using inter-node MPI communicators (e.g., $MPI\_Comm\_join()$).

6) Node-to-node Communicator: Its main role is to allow node-to-node communications regarding both metadata calls (e.g., segment locations, mappings, etc.,) and data movement (e.g., fetch data from another node). It leverages the existence of RDMA or RoCE capabilities of the machine, if present, to minimize overheads. In our prototype we use Mellanox's `libibverbs` OFED drivers.

B. System Generated Prefetching Events

Existing prefetching solutions rely on the application to trigger prefetching events which may result in untimely data prefetching leading to cache pollution and/or redundancy, as discussed in Section II. Modern complex workflows operate on a collection of data that are processed by multiple analysis applications. HFetch leverages this observation and triggers data prefetching based on system-generated events. A data-centric system design should be intelligent to understand how files are accessed and make prefetching decisions triggered by the system itself. However, this is challenging due to the increased complexity of the system software itself. In an ideal system, the kernel would offer the ability to capture file events (i.e., accesses to a file) and pass the collected information to other components of the system software, such as the prefetcher. Alternatively, this functionality could be offloaded to the file system software which could maintain file access statistics. However, this implies specificity of the file system implementation and portability across systems is questionable.

To overcome this challenge, HFetch utilizes an existing Linux kernel subsystem, called $inotify$ [44] (i.e., inode notify), to capture file system events. The $inotify$ tool is implemented at the VFS layer and, hence, all file system calls will be intercepted without a need to create file system specific implementations (e.g., XFS, EXT4, FUSE, etc). To utilize this subsystem, *watches* on the interested directories or files are installed. HFetch monitors the files opened and closed by only those applications which link to it. Once the watches are placed, any event on the file such as read or write is captured by this subsystem. Consistency between updates from external applications (i.e., not linked to HFetch) is therefore maintained by invalidating prefetched data once the write event is created. Moreover, this subsystem collects a counter of file accesses on installed *watched* directories or files. However, this information is not enough to build a profile of how a file is accessed. Therefore, we implemented a lightweight library that intercepts the Linux $inotify$ API and enhances the events generated with extra information about observed file accesses. The original events created by $inotify$ include the type of event (e.g., open, read, write, close) and the filename that the event refers to. We have additionally added the location of a read operation (i.e., offset), the length of the read operation (i.e., request size), and lastly a timestamp. Using the above information HFetch is able to build a *file heatmap* which reflects the pattern with which the file is accessed regardless of who is performing the access. The hotter the region of a file in the heatmap the more important that region is for data access optimization. Those file heatmaps can be stored alongside the raw files (i.e., similar to enriched metafiles) to facilitate future operations on the same file (i.e., maintain historical accesses). Note that this step is optional and not necessary for HFetch to perform data prefetching in contrast to history-based prefetchers. In a sense, the precious commodity within HFetch are the files themselves and the system aims to derive a prefetching scheme that optimizes all accesses to the specific file. Our enhanced $inotify$ library can be dynamically preloaded easily by using an environment variable that points inotify calls to our implementation.

Figure 2 demonstrates an example of how data-centric prefetching works in HFetch. As it can seen, there are two applications issuing read requests for file $f1$ with different HFetch agents, one per application, talking to the HFetch server. We define a prefetching *epoch* as the time that a file is open (i.e., between fopen-fclose) for reading. A file is targeted for prefetching only during an epoch. Upon an fopen call with the appropriate read flags, the HFetch agent will send a $start\_epoch()$ call to the server who will install an $inotify\_add\_watch()$ for access. If an fopen() does not include read flags, the agent will ignore it. Also, if multiple

Client space					Kernel space	HFetch Server space					
Time	Applications		HFetch Agents		inotify_handle_event	Hardware Monitor	Auditor				Data Placement Engine Tiers: T1<T2<T3<T4
	#1	#2	Agent#1	Agent#2	(push to event queue)		Update Segment Statistics			Calculate Segment Score	
							Frequency	Recency	Sequence		
t0	fopen(f1, READ)	-	start_epoch(f1)	-		inotify_add_watch(f1)	[0,0,0,0]	[0,0,0,0]	null	[0.0,0.0,0.0,0.0]	[T4,T4,T4,T4]
t1	fopen(f2, WRITE)	-	IGNORE	-							
	-	fopen(f1, READ)	-	start_epoch(f1)		IGNORE					
t2	fread(f1,0,1)	-	-	-	f1,offset:0,size:1,t2	collect_event()	[+1,0,0,0]	[+t2,0,0,0]	prev->s0	[1.0,0.0,0.0,0.0]	[T1,T4,T4,T4]
t3	fread(f1,1,1)	fread(f1,0,1)	-	-	[{f1,offset:1,size:1,t3}, {f1,offset:0,size:1,t3}]	collect_event()	[+1,+1,0,0]	[+t3,+t3,0,0]	prev->[s0,s1]	[1.5,1.0,0.0,0.0]	[T1,T2,T4,T4]
t4	fread(f1,0,1)	fread(f1,1,2)	-	-	[{f1,offset:2,size:1,t4}, {f1,offset:1,size:2,t4}]	collect_event()	[+1,+1,+1,0]	[+t4,+t4,+t4,0]	prev->[s0,s1,s2]	[1.5,1.5,1.0,0.0]	[T1,T2,T2,T4]
t5	fread(f1,0,1)	-	-	-	f1,offset:0,size:1,t5	collect_event()	[+1,0,0,0]	[+t5,0,0,0]	prev->s0	[1.2,0.5,0.3,0.0]	[T1,T2,T3,T4]
t6	fclose(f1)	-	end_epoch(f1)	-		IGNORE					
t7	fclose(f2)	fclose(f1)	IGNORE	end_epoch(f1)		inotify_rm_watch(f1)					

Fig. 2. An example of data-centric server-push prefetching in HFetch.

fopen from multiple processes or across applications arrive, only the first will install the watch and the last one will remove it. Once a watch on the opened file has been installed, the kernel (with the help of our modified $inotify()$ interceptor) will start creating events based on file accesses. The events are pushed into an event queue hosted by the hardware monitor who is responsible to collect the events and pass them to the file segment auditor. Segment statistics are then updated and a segment score is calculated. The hierarchical placement engine will then use this score to place the prefetched segments in the tiers of the hierarchy.

C. File Segment Scoring

A *file segment* is defined as a file region enclosed by start and end offsets. The segment size is dynamic based on how the file is being read. A file segment is the prefetching unit within HFetch, which means all prefetching operations are expressed by loading one or more segments. Its dynamic nature provides HFetch a better opportunity to decompose read accesses in finer granularity and better utilize the available prefetching cache, especially in a hierarchical environment where the prefetching cache can span multiple tiers. Each incoming read request may correspond to one or more segments. For example, assume the segment size is 1MB and there is an fread() operation starting at offset 0 with 3MB size, then HFetch will prefetch segments 1, 2, and 3 to optimize this data access. For every segment, HFetch maintains its access frequency within a prefetching epoch, when it was last accessed, as well as which segment preceded it (i.e., segment sequencing). HFetch scores each file segment based on these collected access statistics by the following formula:

$$Score_s = \sum_{i=1}^{k} (\frac{1}{p})^{\frac{1}{n}*(t-t_i)} \quad (1)$$

where s is the segment being scored, k is the number of accesses, t is the current time, t_i is the time of the i^{th} access, and $n \geq 1$ is the count of references to segment s. An intuitive meaning of $\frac{1}{n}$ is that a segment's score is reduced to $\frac{1}{p}$ of the original value after every time step. Finally $p \geq 2$ is a monotonically non-increasing class of functions. Consistently with the principle of temporal locality, $t - t_i$ gives more weight to more recent references with smaller backward distances. This score aims to encapsulate three simple observations about the probability of a segment being accessed in the future. A segment is likely to be accessed in the future again if: a) it is accessed frequently, b) it has been accessed recently, and c) it has multiple references to it.

The file heatmaps are generated by the score of each segment. To minimize overheads, HFetch's auditor maintains segment statistics and file heatmaps in an in-memory map for the duration of an epoch (i.e., while the file remains opened for read). Upon closing the file HFetch has the ability to store the file heatmaps on disk resembling a file access history. When a file gets re-opened, if there is a stored heatmap, HFetch will load it in memory and compare observed accesses with the pre-existing heatmap. New accesses will evolve the heatmap further. Heatmaps get deleted once the workflow ends. We envision HFetch to be able to maintain multiple versions of a file heatmap and select the best fit to the current epoch, but in our prototype implementation we only keep the latest heatmap.

D. Hierarchical Data Placement

HFetch is a truly hierarchical data prefetcher, and thus, the prefetching cache spans across multiple tiers of the deep memory and storage hierarchy. In contrast to existing prefetching solution, HFetch fetches data into multiple tiers using the hierarchical data placement engine. Note that HFetch uses an exclusive cache model where the same data can only be present in one tier in contrast to the inclusive CPU caches. Hardware characteristics such as capacity, latency, and throughput of each tier suggest an environment where a higher tier will be faster but with limited capacity. The main decision a hierarchical prefetcher has to make is *where* to place the prefetched data? Instead of driving this decision by the running applications, HFetch chooses the right data for the right tier based on the segment score. Effectively, it maps the file heatmap to the tiers with hotter segments ending up in higher tiers. Note that a tier can be local or remote based on performance characteristics. Applications can then access the prefetched segments from the tier they were placed in. This approach can lead to better resource utilization, masking access latency behind each tier, and can offer concurrent access with less interference.

Algorithm 1 Data Placement Algorithm

```
1: procedure CALCULATEPLACEMENT(segment, tier)
2:     if segment.score > tier.min_score then
3:         if segment cannot fit in this tier then
4:             tier.min_score ← segment.score
5:             DemoteSegments(segment.score,tier)
6:         if segment.score > tier.max_score then
7:             tier.max_score ← segment.score
8:         place segment in this tier
9:     else
10:        CalculatePlacement(segment, tier.next)
11: procedure DEMOTESEGMENTS(score,tier)
12:    segments ← GetSegments(score, tier)
13:    for each s ∈ segments do
14:        CalculatePlacement(s, tier.next)
```

The data placement engine operates in the background and in a decoupled fashion from the applications. Its main responsibility is to periodically monitor the segment score changes from the auditor and to decide if and what segments should be moved up or down the tiers. All updated scores are pushed by the auditor into a vector which the engine processes. To avoid excessive data movements among the tiers, HFetch uses two user-configurable conditions to trigger the engine: a) a time interval (e.g., every 1 sec), and, b) a number of score changes (e.g., every 100 updated scores). The engine maintains a min and max segment score for each available tier. If an updated segment score violates its current tier placement, then it gets promoted or demoted accordingly. For example, let us assume the minimum segment score placed in RAM is 2.0 and a new segment updated score is 2.2, then the new segment will be brought in RAM, the min tier score will be updated to 2.2, and the previous segment with score 2.0 will be placed to a lower appropriate tier (in NVMe in our example). This approach handles both data placement during prefetching but also automatic evictions since each segment has its natural position in the hierarchy based on its score. Note, if segments have exactly the same score, the default policy in HFetch is to randomly place them in the tiers. HFetch's data placement engine iterates through the vector of update scores. Algorithm 1 has a time complexity of $O(m*n)$ where m is the number of segments updated on that node and n is the number of layers. Note, $n << m$ and m is the number of segments updated on a node between an interval t. This t should be ideally configured close to the average computation time of all the applications in the workflow, to avoid excessive computations. Lastly, it is noteworthy to highlight the globality of the segment scoring system since scores are calculated across all observed accesses and not specifically for a single application's access patterns or a given file. Therefore, its correct placement in the tiers of the hierarchy will optimize accesses across multiple applications and files in a workflow. This also avoids scenarios where a prefetcher of one application conflicts with another application leading to cache pollution and unwanted evictions.

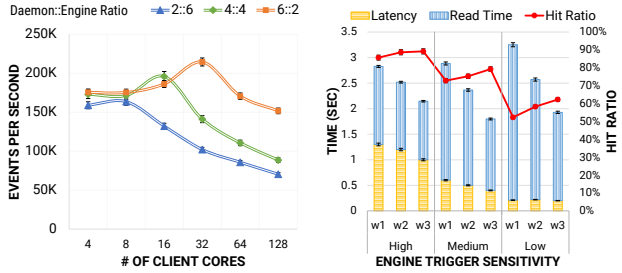
IV. EVALUATION

We evaluate HFetch using synthetic benchmarks that simulate various read patterns. We also perform scaling tests using real scientific workloads that span over a wide range of scientific simulations and data analysis kernels. During each test we collocate the HFetch server with the application cores, as data-centric approach induce almost negligible overheads on the application. All tests were executed five times and we report the average along with the variance. Our evaluation results demonstrate both end-to-end execution time reduction expressed in seconds and miss ratio expressed in %. As our baseline, we use a *No Prefetching* solution based purely on reading from the parallel file system, and we also compare HFetch with state-of-the-art prefetchers *Stacker* and *KnowAc*.

Testbed: All experiments were conducted on the Ares supercomputer at the Illinois Institute of Technology. Each compute node has a dual Intel(R) Xeon Scalable Silver 4114 @ 2.20GHz (i.e., 40 cores per node), 96 GB RAM, 40Gbit Ethernet, and a local 512GB NVMe SSD. Each storage node has a dual AMD Opteron 2384 @ 2.7Ghz, 32GB RAM, 40GBit Ethernet with RoCE, and is also equipped with 2x512GB SSD in RAID, and 2TB HDD. The total experimental cluster consists of 2560 client MPI ranks (i.e., 64 nodes), 4 burst buffer nodes, and 24 storage nodes running an installation of OrangeFS 2.9.8. We use CentOS 7.1 as the operating system, and the MPI version is Mpich 3.2.

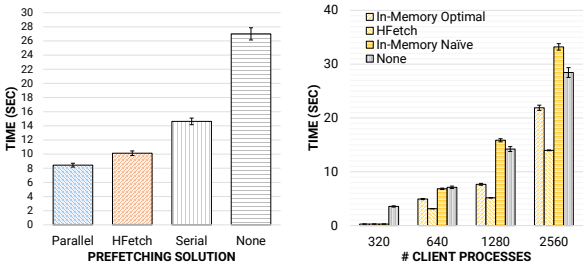
A. HFetch Performance Analysis

*1) **HFetch Internal Components**:* The amount of events produced by the file system impacts the performance of a data-centric approach such as HFetch. To better understand this impact, we evaluate the event consumption ability of HFetch's hardware monitor and file segment auditor by scaling the number of generated events while measuring the consumption rate, reported in events per second. The HFetch server, which has a multi-threaded design, is deployed on a dedicated core with the ability to scale the number of the threads that hardware monitor daemons and the hierarchical data placement engine use. Figure 3(a) demonstrates the results. During this test, each client process issues 100K events and the HFetch server uses 8 threads in total. We scale the number of client cores and we tested three configurations of the server, namely 2 daemon - 6 engine threads, 4 daemon - 4 engine threads, and 6 daemon - 2 engine threads. The intuition behind these configurations is to match the production rate. More available daemons means more throughput on the event queue consumption. The results verify that 6 daemon - 2 engine threads performs better as the number of produced events increases offering more than 200K events per second consumption rate. This implies a granularity of one HFetch server to 32 client cores and can be used as a deployment guideline. A hierarchical data prefetcher has to also decide in which tier of an available hierarchy should it bring data in. This process involves a typical trade-off between the cost of finding the optimal placement of data in the hierarchy and the resulting reading I/O time. Deriving an optimal placement is often more expensive but can lead



(a) HFecth server-to-client ratio (b) Placement engine reactiveness

Fig. 3. HFetch server evaluation.



(a) Reducing RAM footprint (b) Extending prefetching cache

Fig. 4. Effect of hierarchical prefetching.

to better read throughput. On the other hand, a sub-optimal placement, typically random or round robin placements, is quicker to calculate but might result in lower read bandwidths. With this observation in mind, we designed HFetch's data placement engine to be tunable by the user and to be able to adapt to commonly running workloads. In HFetch, the engine is responding to the change of segment scores. It does so either by a time window or by the number of updated scores. Therefore, HFetch can be configured to be extremely sensitive (i.e., triggering the engine per score update) or to be relatively slower to react to score changes. We define this sensitivity to score changes as *engine reactiveness*. Figure 3(b) demonstrates three configurations of engine reactiveness and three workloads that consist of alternating computations and I/O bursts. In this test, the engine is triggered as follows: a) *high*, at every segment score update, b) *medium*, every 100 score updates, and c) *low*, every 1024 score updates. Each I/O burst reads 1GB of data in 1MB requests and $w1, w2, w3$ are a data-intensive, a balanced, and a compute-intensive workload respectively. The amount of computation between the I/O bursts gives the prefetcher a better chance to complete the data loading. This can be seen by the results where $w3$ demonstrates the best overall performance across all engine configurations. On the other hand, a highly sensitive engine achieves the best hit ratio of around 88%, but at the cost of increased latency penalties which stem from increased data movement among the tiers, and thus, interference with the read operations. In contrast, low sensitivity of the engine results in low hit ratios but significantly improves the latency observed. Note that lower hit ratios impact the performance more compared to the interference slowdown. The medium sensitivity (the default in HFetch) resulted in the best performance for $w2$ and $w3$, balancing latency penalties and read bandwidth.

*2) **Hierarchy-aware prefetching**:* Architectural trends indicate that the amount of available memory per core is decreasing [45]. One possible solution is to extend the prefetching cache into more tiers of the hierarchy. There are two ways to utilize a hierarchy: a) reduce the DRAM footprint by extending the cache to more tiers, b) increase the total available cache size by expanding it to more tiers. To better understand the impact of the size and physical location of the prefetching cache, we compare a traditional single-tier prefetcher with a hierarchical one and measure the end-to-end time. Figure 4(a) shows the results of the first configuration. In this test, we deployed 2560 MPI processes, each performing sequential reads, for a total of 40 GB in 10 time steps. We evaluate HFetch against a *serial* prefetcher, a *parallel* prefetcher, and a *no-prefetching* approach. Both HFetch and the parallel prefetcher use four threads. The prefetching cache size is 40 GB. In the case of HFetch, this cache spans across three tiers: 5 GB in RAM, 15 GB in NVMe, and 20 GB in burst buffers. The data initially reside in a PFS installation of 24 storage servers. As shown in the figure, the parallel prefetcher is able to overlap reading with the prefetching operations almost perfectly resulting in a 89% hit ratio and the smallest execution time. In contrast, the serial prefetcher can only bring one data piece at a time and its miss ratio is higher since reading from RAM is faster than fetching data from PFS. By utilizing more tiers, HFetch is able to reduce the RAM footprint by 8x while being only 17% slower than the parallel since it directs read operations to slower tiers. It is 44% faster than the serial one. HFetch achieves this performance by pipelining fetching operations from tier to tier while allowing the application cores to read from multiple tiers at the same time increasing access concurrency. It does of course expose increased latencies since lower tiers are slower than RAM. However, trading minor performance losses for a significant RAM footprint reduction is crucial to a lot of modern workloads such as BigData and ML analytics.

In Figure 4(b), we show results when expanding the prefetching cache with more tiers. In this test, we weak scale the I/O operations by scaling the number of client processes. Each process sequentially reads 16MB in 4 time steps which results in 40 GB of total I/O. We compare HFetch with these prefetchers: a) *in-memory optimal*, where each process brings data into its own cache, and b) *in-memory naive*, where each process competes for access to the prefetching cache. The prefetching cache size for both in-memory prefetchers is configured at 5 GB RAM space whereas for HFetch we supplement it with 15 GB NVMe and 20 GB burst buffer space. As can be seen, increasing the cache size, even if it is in lower, less capable tiers, helps reduce miss ratio penalty and ultimately leads to higher read performance. For smaller scale, all solutions can fit all data in RAM and therefore all achieve the same performance. As the scale grows, prefetched data cannot fit in the prefetching cache and therefore applications are forced to go to the PFS. This is obvious for the in-memory naive prefetcher's performance where the prefetcher threads and the application threads both

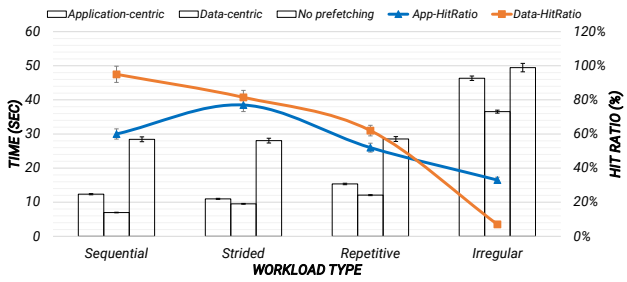


Fig. 5. Application-centric vs. data-centric prefetching.

compete for access to PFS (i.e., observed interference), and thus, enabling prefetching actually hurts the performance when compared to the native PFS performance. In summary, HFetch can expand the cache size by utilizing the additional tiers available and therefore decrease the miss ratio and boost performance by 35% over the optimal in-memory prefetcher and 50% over the no-prefetching baseline.

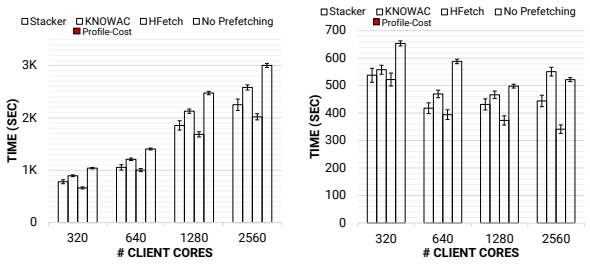
*3) **Data-centric prefetching**:* A system-wide prefetching approach has the advantage of observing how files are accessed across multiple processes or even applications. HFetch's design implements a data-centric logic where access statistics are collected and prefetching decisions are made based on how important a file block or region (i.e., segment in HFetch) is. In contrast, an application-centric prefetcher's main objective is to identify how each application accesses its data and make prefetching decisions accordingly. As discussed in Section II, this approach might create scenarios where the cache can get polluted, or data are fetched twice, or unnecessary evictions of prefetched data occur. To better understand the differences between an application-centric and data-centric prefetching approach and identify which workloads work best in each, we tested HFetch under the following scenario. We have 2560 processes in total organized in four different communicator groups representing different applications resembling a data analysis and visualization pipeline. Each process issues read requests on the same dataset. We tested four commonly-used patterns [45]: sequential, strided, repetitive, and irregular access patterns. The prefetching cache size is configured to fit the total data size of two out of the four applications which means applications compete for access to this cache. For HFetch the prefetching cache is configured to fit one application's load in RAM and one in NVMe. Figure 5 demonstrates the evaluation results. As can be seen, for sequential, strided, and repetitive patterns, HFetch achieves 26% higher performance when compared to an application-centric approach. HFetch is able to capture how data are accessed across applications or files and understand which segments are important to fetch from a global perspective. This results in zero cache evictions and no cache pollution. Both, application-centric and HFetch suffers from irregular patterns, but application-centric suffers more as data-centric would be see the globality of accesses and optimize accordingly.

B. Real Scientific Workflows with HFetch

To evaluate the effectiveness of HFetch's hierarchical-aware and data-centric prefetching approach, we performed scaling tests using two complex multi-application scientific workflows: namely Montage [46] and WRF [47]. Both these real-world workflows showcase the need for a data-centric approach so that multiple applications running together can effectively have data prefetched globally. We compare HFetch with Stacker with KnowAc, one online and one offline prefetcher. Both of those solutions are configured to fetch data from burst buffers to the application's memory.

*1) **Montage**:* This workflow is a collection of MPI programs comprising an astronomical image mosaic engine. Each phase of building the mosaic takes an input from the previous phase and outputs intermediate data to the next one. Specifically, FITS images are initially read by multiple processes in a sequentially order. Then, a subset of them are re-projected into different images. During this stage multiple processes read the same images multiple times but in different time-frames. Once projected images are produced, another multi-processed program runs a `diff` between all the projected images and calculates the least square distance. This phase is executed until the model converges resulting in a random but repetitive read pattern. Finally, a correction is applied on the overlaid images and the final image is created. Hence, Montage's workflow is highly read-intensive and iterative [48] making it an ideal case to perform prefetching effectively [49]. Figure 6(a) shows the results for Montage. During this test, each process does 10 MB of I/O operations in 16 time steps for a total of 400 GB for the largest scale. We weak scaled the execution of Montage by increasing the number of processes from 320 to 2560. Required data are initially staged in the burst buffer nodes. The system is overall configured with prefetching cache organized in 1.5 GB RAM space, 2 GB in local NVMe drives and 400 GB burst buffer allocation. As can be seen, the best read performance is achieved by KnowAc, a history-based prefetcher, since the prefetcher knows exactly what to load next. However, such an approach suffers from prolonged profiling costs. Stacker avoids pre-processing steps and build its models as it goes, but demonstrated a lower hit ratio due to some cache conflicts and unwanted data evictions. HFetch was able to utilize all available tiers and performed the best, offering from 5% to 25% better end-to-end performance when compared to Stacker and 10% to 30% better than KnowAc (i.e., profile-cost plus run time). Note that all solutions scale nicely.

*2) **WRF**:* This workflow is a multi-application mesoscale numerical weather prediction system designed for both atmospheric research and operational forecasting needs. It is an iterative workflow where components of the simulation analyze observed and simulated data many times until the model converges. As the model is simulated, an analysis application produces a visualization of this model. There are three distinct phases: pre-processing, main model, post-processing and visualization. More details on how WRF works can be found in [50]. Figure 6(b) shows the results for



(a) Montage (weak scaling) (b) WRF (strong scaling)

Fig. 6. End-to-end performance of scientific workflows.

WRF. During this test, each process reads 8MB of data in 4 time steps for a total of 80GB across all scales (i.e., strong scale). Input data are assumed to be initially present in the burst buffer nodes. The system is configured with prefetching cache organized in 1.25 GB RAM space, 2 GB in local NVMe drives and 80 GB burst buffer allocation. Results confirm our previous observations with KnowAc having the best read time but additional profiling costs and Stacker demonstrating better end-to-end time over KnowAc. HFetch is able to utilize all tiers and scaled better than all solutions.

V. Related Work

The related work has already been presented in Section II. However, we highlight here the main differences of HFetch from the existing work in the following key areas:

a) HFetch implements an exclusive cache design where prefetched data reside in only one tier. Hardware prefetchers in CPU designs have inclusive cache designs.

b) HFetch does not require any pre-processing steps or offline analyses. Offline prefetchers only work after extensive tracing or pre-execution of the application.

c) HFetch relies on dynamic prefetching granularity to leverage the placement of data in the DMSH by using adaptive segment sizes. Existing solutions opt for a static prefetching unit (i.e., either an entire file or small data variables).

d) HFetch leverages the DMSH transparently whereas most existing prefetchers cannot handle the presence of multiple tiers opting either to bypass them or partially use them as overflowing data buffers.

e) HFetch differentiates itself by proposing a server-push (i.e., data-centric) prefetching approach whereas prefetching has traditionally been triggered by applications and their accesses.

In addition to that relevant research, we identify the following relevant work. Diskseen [19], tracks the locations and access times of disk blocks. Based on analysis of their temporal and spatial relationships, it seeks to improve the sequentiality of disk accesses and overall prefetching performance. However, disk blocks do not carry file semantics and relationships between segments. During HFetch design and development, we drew partial motivation from a cache replacement algorithm presented in [51] where frequency and recency of a memory page can both influence the eviction of the page. HFetch's segment scoring resembles in a sense a similar approach where we target segment with score based on access frequency and recency.

VI. Conclusions and Future Work

While data prefetching solutions is an effective data access optimization that masks the latency by pre-loading data before they are needed, they are designed from an application-centric point of view and fail to recognize global patterns. Additionally, they were designed to operate with prefetching caches in DRAM space which will be an unrealistic approach as we move forward to extreme scale computing environments where each core has access to less and less memory capacity. To overcome these challenges, we designed and presented, HFetch, a hierarchical data prefetcher that implements a data-centric design in this paper. We showcase the benefits of such an approach by evaluating its scalability, effectiveness, and overall performance. Results show, HFetch is a promising solution to a growing problem of extreme scale data access. HFetch achieves 10-35% higher read throughput than prefetching solutions tested, and, over 50% improvement over native storage performance without prefetching. As future steps, we plan to extend the evaluation with more applications, deploy HFetch to larger scales, and enhance its scoring models with machine learning.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant no. OCI-1835764 and CSR-1814872.

References

[1] P. Carns, K. Harms, W. Allcock *et al.*, "Understanding and improving computational science storage access through continuous characterization," *ACM Transactions on Storage (TOS)*, vol. 7, no. 3, p. 8, 2011.

[2] X.-H. Sun, Y. Chen, and M. Wu, "Scalability of heterogeneous computing," in *2005 International Conference on Parallel Processing (ICPP'05)*. USA: IEEE, 2005, pp. 557–564.

[3] J. C. Bennett, H. Abbasi, P.-T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci *et al.*, "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 49.

[4] B. Dong, T. Wang, H. Tang, Q. Koziol *et al.*, "ARCHIE: Data Analysis Acceleration with Array Caching in Hierarchical Storage," in *2018 IEEE International Conference on Big Data (Big Data)*, USA, pp. 211–220.

[5] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *International Conference on High Performance Computing for Computational Science*. USA: Springer, 2010, pp. 1–25.

[6] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Communications of the ACM*, vol. 58, no. 7, pp. 56–68, 2015.

[7] A. Kougkas, M. Dorier, R. Latham, R. Ross, and X.-H. Sun, "Leveraging burst buffer coordination to prevent I/O interference," in *2016 IEEE 12th International Conference on e-Science (e-Science)*. USA: IEEE, 2016.

[8] A. Kougkas, H. Devarajan, and X.-H. Sun, "Hermes: a heterogeneous-aware multi-tiered distributed I/O buffering system," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. USA: ACM, 2018, pp. 219–230.

[9] Cray, "Datawarp documentation," 2016. [Online]. Available: https://pubs.cray.com/browse/datawarp/software

[10] DDN, "IME burst buffers documentation," 2018. [Online]. Available: https://www.ddn.com/products/ime-flash-native-data-cache/

[11] G. K. Lockwood, D. Hazen, Q. Koziol *et al.*, "Storage 2020: A Vision for the Future of HPC Storage," NERSC, Tech. Rep., 2017.

[12] B. Dong, S. Byna, K. Wu, H. Johansen *et al.*, "Data elevator: Low-contention data movement in hierarchical storage system," in *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. Hyderabad, India: IEEE, 2016.

[13] T. Wang, S. Byna, B. Dong, and H. Tang, "UniviStor: Integrated Hierarchical and Distributed Storage for HPC," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. USA: IEEE, 2018.

[14] I. Stefanovici, E. Thereska, G. O'Shea *et al.*, "Software-defined caching: Managing caches in multi-tenant data centers," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. USA: ACM, 2015, pp. 174–181.

[15] I. Gorton and J. Klein, "Distribution, data, deployment: Software architecture convergence in big data," *IEEE Software*, vol. 32, 2015.

[16] K. J. Brown, H. Lee, T. Romp *et al.*, "Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns," in *2016 IEEE International Symposium on Code Generation and Optimization (CGO)*, pp. 194–205.

[17] S. Yang, T. Harter, N. Agrawal *et al.*, "Split-level I/O scheduling," in *Proceedings of the 25th Symposium on Operating Systems Principles*. USA: ACM, 2015.

[18] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng, "Datastager: scalable data staging services for petascale applications," *Cluster Computing*, vol. 13, no. 3, pp. 277–290, 2010.

[19] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, "DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch," in *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*. Santa Clara, CA: USENIX Association, 2007, pp. 20:1–20:14.

[20] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, "A study of integrated prefetching and caching strategies," *ACM SIGMETRICS Performance Evaluation Review*, vol. 23, no. 1, pp. 188–197, 1995.

[21] F. Chang and G. A. Gibson, "Automatic I/O Hint Generation Through Speculative Execution," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI '99. Berkeley, CA, USA: USENIX Association, 1999, pp. 1–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=296806.296807

[22] J. He, X.-H. Sun, and R. Thakur, "Knowac: I/O prefetch via accumulated knowledge," in *2012 IEEE International Conference on Cluster Computing*. USA: IEEE, 2012, pp. 429–437.

[23] B. Saraladevi, N. Pazhaniraja, P. V. Paul, M. S. Basha, and P. Dhavachelvan, "Big Data and Hadoop-A study in security perspective," *Procedia computer science*, vol. 50, pp. 596–601, 2015.

[24] J. Bent, G. Grider, B. Kettering, A. Manzanares, M. McClelland, A. Torres, and A. Torrez, "Storage challenges at Los Alamos National Lab," in *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. Pacific Grove, CA: IEEE, 2012, pp. 1–5.

[25] E. Barton, "DAOS an architecture for extreme scale storage," 2015. [Online]. Available: http://tinyurl.com/y2jqoevt

[26] P. Subedi, P. Davis, S. Duan *et al.*, "Stacker: an autonomic data movement engine for extreme-scale data staging-based in-situ workflows," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. USA: IEEE Press, 2018, p. 73.

[27] D. J., "memory access. White paper, Intel Research Website," 2006. [Online]. Available: https://tinyurl.com/yydpfkt2

[28] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* USA: IEEE, 2003, pp. 129–140.

[29] M. Y. Qadri, N. N. Qadri, M. Fleury, and K. D. McDonald-Maier, "Energy-efficient data prefetch buffering for low-end embedded processors," *Microelectronics Journal*, vol. 62, pp. 57–64, 2017.

[30] Intel, "Intel 64 and IA-32 Architectures Optimization Reference Manual," 2019. [Online]. Available: https://tinyurl.com/lxxw7sn

[31] X.-H. Sun, S. Byna, and Y. Chen, "Server-based data push architecture for multi-processor environments," *Journal of Computer Science and Technology*, vol. 22, no. 5, pp. 641–652, 2007.

[32] G. Cherubini, Y. Kim, M. Lantz, and V. Venkatesan, "Data Prefetching for Large Tiered Storage Systems," in *2017 IEEE International Conference on Data Mining (ICDM)*. New Orleans, USA: IEEE, 2017.

[33] M. C. Wiedemann, J. M. Kunkel, M. Zimmer, T. Ludwig, M. Resch, T. Bönisch, X. Wang, A. Chut, A. Aguilera, W. E. Nagel *et al.*, "Towards i/o analysis of hpc systems and a generic architecture to collect access patterns," *Computer Science-Research and Development*, pp. 1–11.

[34] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, "The cost of doing science on the cloud: the montage example," in *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Ieee, 2008, pp. 1–12.

[35] Y. Joo, S. Park, and H. Bahn, "Exploiting i/o reordering and i/o interleaving to improve application launch performance," *ACM Transactions on Storage (TOS)*, vol. 13, no. 1, p. 8, 2017.

[36] H. Devarajan, A. Kougkas, P. Challa, and X.-H. Sun, "Vidya: Performing Code-Block I/O Characterization for Data Access Optimization," in *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, Dec 2018, pp. 255–264.

[37] V. Thilaganga, M. Karthika, and M. M. Lakshmi, "A Prefetching Technique Using HMM Forward and Backward Chaining for the DFS in Cloud," *Asian Journal of Computer Science and Technology*, vol. 6, no. 2, pp. 23–26, 2017.

[38] N. Tran and D. A. Reed, "Automatic ARIMA time series modeling for adaptive I/O prefetching," *IEEE Transactions on parallel and distributed systems*, vol. 15, no. 4, pp. 362–377, 2004.

[39] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross, "Omnisc'IO: a grammar-based approach to spatial and temporal I/O patterns prediction," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. USA: IEEE, 2014.

[40] G. Daniel, G. Sunyé, and J. Cabot, "PrefetchML: A Framework for Prefetching and Caching Models," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. New York, NY, USA: ACM, 2016, pp. 318–328.

[41] H. Devarajan, A. Kougkas, P. Challa, and X.-H. Sun, "Data Prefetching using System-generated Events: Application-centric vs Data-centric? ," 2019. [Online]. Available: http://www.cs.iit.edu/~scs/assets/files/devarajan2019HFetch_tr.pdf

[42] H. Devarajan, "Hierarchical Data Prefetching software," 2019. [Online]. Available: https://bitbucket.org/scs-io/hfetch

[43] H. Devarajan and C. Hogan, "HCL: Hermes Container Library," 2019. [Online]. Available: https://github.com/HDFGroup/hcl

[44] I. Shields, "Monitor Linux file system events with inotify," 2010. [Online]. Available: https://developer.ibm.com/tutorials/l-inotify/

[45] T. Jin, F. Zhang, Q. Sun, H. Bui, M. Romanus *et al.*, "Exploring data staging across deep memory hierarchies for coupled data intensive simulation workflows," in *2015 IEEE International Parallel and Distributed Processing Symposium*. USA: IEEE, 2015, pp. 1033–1042.

[46] G. Berriman, J. Good, A. Laity, and M. Kong, "The Montage image mosaic service: custom image mosaics on-demand," *Astronomical Data Analysis Software and Systems ASP*, vol. 394, no. 2, 2008.

[47] M. Laboratory, "WRF, Weather Research and Forecasting Model," 2017. [Online]. Available: https://www.mmm.ucar.edu/weather-research-and-forecasting-model

[48] S. Bharathi, A. Chervenak, E. Deelman *et al.*, "Characterization of scientific workflows," in *2008 third workshop on workflows in support of large-scale science*. IEEE, 2008, pp. 1–10.

[49] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham, "Effective stream-based and execution-based data prefetching," in *Proceedings of the 18th annual international conference on Supercomputing*. ACM, 2004, pp. 1–11.

[50] W. C. Skamarock, J. B. Klemp, J. Dudhia, D. O. Gill, D. M. Barker, W. Wang, and J. G. Powers, "A description of the advanced research wrf version 3. ncar technical note-475+ str," 2008.

[51] D. Lee, J. Choi, J.-H. Kim, S. H. Noh *et al.*, "LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE transactions on Computers*, vol. 12, pp. 1352–1361, 2001.