Stacker: An Autonomic Data Movement Engine for Extreme-Scale Data Staging-Based In-Situ Workflows

Pradeep Subedi†, Philip Davis†, Shaohua Duan†, Scott Klasky‡, Hemanth Kolla§, and Manish Parashar†
†{pradeep.subedi, philip.e.davis, shaohua.duan, parashar}@rutgers.edu
†*Rutgers Discovery Informatics Institute, Rutgers University, Piscataway, NJ*
‡klasky@ornl.gov
‡*Oak Ridge National Laboratory, Oak Ridge, TN*
§hnkolla@sandia.gov
§*Sandia National Laboratories, Livermore, CA*

***Abstract*—Data staging and in-situ workflows are being explored extensively as an approach to address data-related costs at very large scales. However, the impact of emerging storage architectures (e.g., deep memory hierarchies and burst buffers) upon data staging solutions remains a challenge. In this paper, we investigate how burst buffers can be effectively used by data staging solutions, for example, as a persistence storage tier of the memory hierarchy. Furthermore, we use machine learning based prefetching techniques to move data between the storage levels in an autonomous manner. We also present Stacker, a prototype of the proposed solutions implemented within the DataSpaces data staging service, and experimentally evaluate its performance and scalability using the S3D combustion workflow on current leadership class platforms. Our experiments demonstrate that Stacker achieves low latency, high volume data-staging with low overheads as compared to in-memory staging services for production scientific workflows.**

***Index Terms*—Extreme Scale Data Staging, Machine Learning, Data Prefetching, High Performance Computing**

I. Introduction

End-to-end scientific workflows include coupled simulations along with data handling/processing services such as analysis and visualization. Running at extreme-scales, these workflows can produce insights into complex phenomenon with increasing fidelity. However, the execution of these simulations on leadership class systems present significant data management challenges due to the increasing volume of data that has to be managed and moved. This data can be in the order of petabytes [3], [24]. For example, the XGC1 gyrokinetic particle-in-cell simulation workflow can generate up to 100PB of data [15], [20] in a single run on Titan, the leadership-class Cray System at ORNL. Moving this data between the simulation and analysis components of the workflow quickly becomes a serious bottleneck. In-situ and in-transit data processing solutions have been proposed to address the data movement and analysis challenges [3], [13], [24]. Many of these approaches use data staging, leveraging memory at staging nodes to hold data as it moves between the components of the workflow [1], [3], [19]. However, emerging architectural trends indicate a decrease in the amount of DRAM memory per core as well as a slower rate of increase in the storage bandwidth as compared to the computational throughput [19], [32]. These trends will severely limit the effectiveness of memory-based staging solutions, which enable in-situ workflow execution and allow applications to tolerate slower disk latencies.

The rate of data generation is just one challenge putting pressure on in-memory staging solutions for in-situ workflows. Even if the simulation application is not generating data at an extremely high rate, coupling or data analysis may require data accumulation over multiple simulation iterations causing the data flowing between the components of the workflow to grow quickly. With limited memory sizes, this intermediate data will have to be offloaded from in-memory staging, possibly to the parallel file system (PFS). However, the increasing latency gap between disks and memory makes this solution undesirable.

Systems at the leadership computing facilities are increasingly augmenting the hardware and I/O middleware layers to better handle applications with bursty I/O by attaching a set of on-node NVMe SSDs (often called burst-buffers) [22]. Emerging NVM technologies have smaller access latencies and better throughput as compared to the PFS, and can therefore be used as an intermediate tier of the memory hierarchy. This tier of on-node NVMe SSDs can enable data staging services to offload data from staging memory [19], [28] without incurring the performance penalty associated with moving it to the PFS. While these burst buffers are designed to act as a write-behind cache for HPC storage systems, integrating them as a part of the memory hierarchy for data staging presents new challenges. Although burst buffers have higher throughput than conventional disks, they have much longer access latencies and a lower throughput as compared to main memory. In an application workflow, if a component of the workflow must access data offloaded to the burst buffer tier it will experience a higher latency, which can significantly impact the performance of the application.

Intelligent data prefetching based on access patterns can hide this disk access and data transfer latency by moving the data to DRAM *before* it is requested [5], [11]. The effectiveness of such prefetching, however, depends upon the

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00 ©2018 IEEE

ability to recognize data access patterns and identify appropriate data to be prefetched. Existing prefetching approaches for coupled scientific HPC applications have been based on user-defined hints [19]. While this works well for simple applications, as application workflows become more complex, this approach quickly becomes infeasible. Furthermore, access patterns can change at runtime, the workflow may include third party libraries where the access pattern is not know to the user, and staging resources may be shared between multiple application workflows [3], [10], [13], [38]. These factors make it likely that user-defined hints will be of little use or even detrimental to performance in complex application workflows, since prefetching the wrong data due to incorrect hints degrades performance due the additional data movement. An alternate prefetching approach is based on spatial and/or temporal locality, analogous the approaches used for CPU caches [26]. While locality-based prefetching can work well for regular accesses within a single application, it is not as effective for more complex access patterns (e.g., variable strided access) or cross-component data access in an application workflow [16], [19], [29].

The goal of this paper is to explore machine learning based approaches to capture the data access patterns between components of staging-based in-situ application workflows, and to use these learned access patterns to move data between the storage layers of the staging service in an autonomous manner. Specifically, in this paper we present the design of Stacker, an autonomous staging-based data management runtime. Rather than modeling individual applications and their access patterns, in Stacker we model the data accesses between multiple components of application workflows. We do this because, in the case of application workflows, access patterns of the different components applications that make up the workflow (and from different workflows sharing the staging resources) can be interleaved at the staging servers, and as a result accurately modeling an individual application's access pattern is not sufficient.

Stacker uses various n-gram [4] models to dynamically manage and optimize data movement across multiple layers of the storage hierarchy. Specifically, incoming read requests to the data staging servers are tracked at runtime to build n-gram models. These models are used to anticipate future requests for prefetching data objects from SSDs to DRAM. This reduces application perceived SSD access overheads and improves the overall data read time. We implemented and deployed Stacker using DataSpaces [13] framework on Calibrun Supercomputer [31] and Titan Cray XK7 system. Since Titan is not equipped with SSDs, we mimicked data read/write behaviors of Intel SSD DC P4600 SSD [7] by introducing appropriate latencies for every read/write requests.

In this paper, we make following contributions:

- We design Stacker, a multi-tiered data staging system, which is based on DataSpaces and utilizes on-node SSD and DRAM to enable high capacity data staging for in-situ workflows, while maintaining low overheads.
- We implement an autonomous data movement runtime in Stacker to enable efficient data movement and placement across the multiple layers of the staging memory/storage hierarchy without a priori knowledge of application data access pattern or user inputs/hints.
- We evaluate Stacker on current leadership computing systems using S3D combustion workflow running on up to ∼32K cores and demonstrate that it can reduce read response time by up to 54% in comparion to no-prefetching.

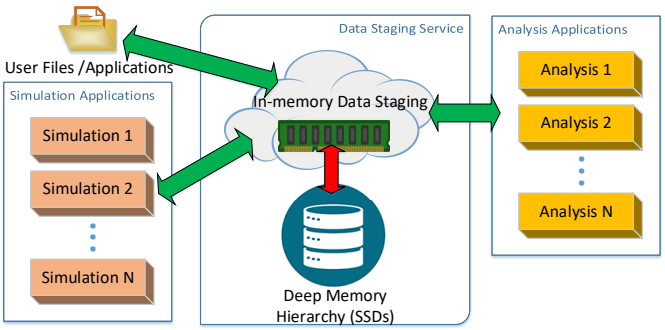


Fig. 1. A typical data staging framework with in-memory data staging. Each of the data staging nodes/servers is equipped with SSDs. Simulations and other applications put/get data to/from staging service. Analysis applications analyze the data and produce intermediate data.

The rest of the paper is organized as follows. Section 2 provides background and a motivating example. Section 3 describes the design and implementation of Stacker. In Section 4, we evaluate Stacker and compare its performance with other staging-based data prefetching approaches. Section 5 includes a discussion of related work and Section 6 presents concluding remarks.

II. Background and Motivation

In this section, we provide a background on data staging and then describe some of the challenges present in scientific HPC workflows. We then investigate why the traditional mechanisms of integrating the deep memory hierarchy are not efficient in supporting HPC workflows with dynamic data access patterns. We then introduce new techniques which alleviate such problems.

A. Data Staging and In-situ Workflows

A sophisticated data staging workflow couples multiple simulation applications with analysis applications to enable efficient data sharing across the workflow. Figure 1 illustrates a coupled simulation workflow, where the simulation applications generate data by ingesting various user input files or data from other simulation applications. Different simulation applications may run at different rates and with different resource requirements. This heterogeneity is evident in the generation of intermediate and output data products as these data become available at different times through the lifetime of the workflow. These output data are stored in

dedicated memory of some intermediary nodes (often called staging nodes/servers.) Multiple analysis applications also connect to these dedicated nodes via the data staging service to consume the data and produce prompt insight.

Data staging techniques leverage dedicated resources in the HPC platform to store and process data as it flows from one component to another. In-memory data staging approaches, such as DataSpaces [13] offer a virtual shared-space abstraction that can be accessed in a transparent manner by various components of in-situ/in-transit workflows. The data movement between the workflow components is usually supported via RDMA-based asynchronous data transport mechanisms. This allows live data to be extracted and analyzed, which makes timely insights from data possible. It should be noted that simulation/analysis applications and staging processes can also be co-located on the same node. Using a staging process allows the coupling to be loose so that coupled applications need not be heavily modified.

While in-memory data staging techniques are very useful for improving workflow performance, they naively store all of the data in the DRAM of the dedicated staging nodes. Increasing data volume and non-uniform data production and consumption in extreme-scale can easily result in situations where data volumes exceed available DRAM resource in the staging area. In such situations, data must be pushed to the lower levels of memory hierarchy, such as on-node SSDs or burst buffers.

B. Deep Memory Hierarchy and Data Prefetching

Integrating various levels of deep memory hierarchy into data staging provides a larger shared-space abstraction, but does not come without caveats. Although pushing data to lower levels frees up DRAM capacity, access to the data being stored at lower levels of deep memory hierarchy incurs extra access latency and lower data throughput. The artifact of longer access latency comes not from the data staging framework design, but rather from the system hardware. In general, SSDs and burst buffers are several orders of magnitude slower than DRAM in terms of access latencies and these technologies do not support RDMA data transfers. Even if RDMA were available for such devices, the access latency variation across different storage devices affects the application perceived latency, because data still needs to be moved from the SSD to the consumer application. Thus, analysis applications will see an increase in data access time when the requested data is stored in SSDs. To alleviate such problems, data must be prefetched from SSDs to DRAM, with the expectation that the prefetched data is highly likely to be accessed in the near future.

There are several different approaches to identifying which data should be prefetched. One of the methods explored in [19] involves a user providing hints about the data access pattern. While providing hints on the likely data access pattern seems feasible for a small application, it can be almost impossible to determine these patterns *a priori* in a more complex workflow, since dynamic data-driven HPC analysis applications evolve based upon earlier simulation/analysis results. To provide hints for a data staging service, a user is required to have a detailed understanding of the underlying simulation and analysis applications to capture the access patterns. It is not reasonable to expect a user of a large scale HPC workflow to understand the minutiae of the (potentially dynamic) data exchange patterns of their workflow well enough to provide generally optimal hints. This makes a solution dependent on user hints undesirable.

A second approach, locality-based prefetching, is widely used in compiler architectures [26]. While locality-based prefetching methods are good for applications that read data on a contiguous tile-by-tile pattern, they suffer from a higher rate of misprediction for applications that perform variable strided data access. The reason for this is that locality-based data prefetches data stored near current requests and if the stride length is larger than prefetch length, there will be a misprediction. These more frequent mispredictions increase the perceived latency of the workflow. Additionally, HPC workflows typically involve multiple processes or applications concurrently reading datasets. When these applications issue requests in parallel for data access, the requests are interleaved and actual data access pattern from the perspective of the storage server will be very different from that of a single application/process. In this case, locality-based prefetching tends to work quite poorly.

As an illustrative example, consider the case in which applications A and B are reading from datasets that consist of variables α and β, respectively, from a staging server. Let us assume that A reads dataset α sequentially and B reads dataset β in the same way. If A and B issue requests such that data staging server sees requests in the pattern of $\{\alpha, \beta, \alpha, \beta ...\}$, then locality-based prefetching will have a very high rate of misprediction. When some data from variable α is requested, locality-based prefetching copies extra data of variable α into memory. When a request for variable β arrives next, the data of β must be copied into memory. Since memory is limited, the prefetched data from α is evicted and the request for β by B is served along with locality-based prefetch of some different portion of variable β. This misprediction, with the subsequent avoidable prefetch and eviction, continues in every subsequent request, which directly affects the workflow's perceived latency. In this example, we want to point out that effective data prefetching techniques should not only capture an individual application's pattern, but should also consider the inter-application data access pattern. In short, a prefetching algorithm should also predict the variable or dataset likely to accessed, in addition to the request offsets or access boundaries for the next request.

The inefficacies of user-based hints and locality-based prefetching warrant an autonomous data movement engine between multiple layers of the memory/storage hierarchy. One of the potential solutions to the problem of

characterizing evolving data patterns is to use an Artificial Neural Network (ANN) [39] to capture the ever-changing data access pattern. While this solution might be able to capture dynamic pattern, most of these methods are designed for offline training. When these networks are trained over offline data sets, it will lead to these algorithms to behave poorly on new, dynamic datasets. Even if these algorithms are ported to perform online training, it requires a prohibitive amount of time to train the network (for example, $\sim 50\%$ of the total execution time is required in [34] to model the I/O performance in Titan Supercomputer). Even if the network is trained, when significant numbers of mispredictions occur, retraining of the network becomes necessary, which again takes significant time. In this paper, to address the above mentioned concerns about autonomic data prefetching while capturing dynamic data access patterns, we present the Stacker staging platform. Stacker uses n-gram models that are updated with every incoming read request to perform data access prediction.

C. N-grams

An n-gram model is a type of probabilistic language model for predicting the next item in a sequence which takes the form of an $(n-1)$-order Markov model [4]. An n-gram is a subsequence of N consecutive tokens in a stream of tokens. N-gram analysis has been heavily used in text mining and natural language processing [4], [9], [36] and also explored in various areas of speech recognition [17] and URL prediction [30]. An n-gram distribution is generated by sliding a fixed-size windows across a stream of tokens and counting the occurrence of each *gram*. This gram frequency is then used to build a probabilistic model and the model is used to perform either prediction or pattern recognition. The choice of the windows size (n) determines the computational complexity and size of the frequency/probability table. A larger value of n increases the size of the frequency table. Rather than using a fixed size n for n-gram or single-order Markov chains, Stacker builds hierarchical n-gram models and uses a cascading approach to expand the search over various length n-gram models for efficient prediction of upcoming requests.

III. Stacker

Motivated by the decrease in the DRAM memory size per core in HPC platforms and increasing availability of on-node SSDs, we extend in-memory data staging solutions to span across DRAM and SSDs/burst buffers. This allows us to dramatically increase the amount of data that can be stored in the staging area. To support autonomous data movement across this multi-tier memory design, we present Stacker, which uses n-gram models to learn and predict the next data access request to enable data prefetching. Specifically, the history of data access requests is stored in an n-gram structure and that information is leveraged for selecting the best candidate for prefetching. Stacker maintains hierarchical storage between DRAM (top-layer) and on-node SSD

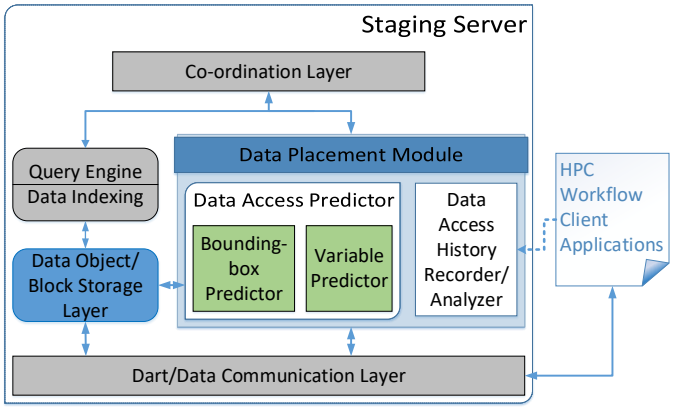


Fig. 2. Architecture of multi-tiered staging framework of Stacker. The Data Placement Module and Data Object/Block Storage layer were implemented on top of DataSpaces framework for Stacker.

(bottom-layer). While Stacker can be extended to include network attached storage, the significant latency associated with such storage devices makes them unsuitable for data staging workflows. If the performance of network storage or PFS was adequate, data staging would not be required. Furthermore, leadership class HPC systems do not generally have hard disks available for use on compute nodes, and as a results disks are not considered as a staging storage resource in this paper. While Stacker assumes that SSD's are sitting at a lower level than DRAM, this doesn't limit the applicability of Stacker to architectures where SSD's are 'in-parallel' rather than 'under' DRAM in the memory hierarchy. In such architecture, Stacker can be used to classify hot and cold data based on the next-predicted sets of access requests and make a dynamic decision on where such objects are to be stored.

Figure 2 presents a schematic overview of Stacker. It is built upon the DataSpaces framework and existing components are leveraged to implement a complete data staging framework spanning across multiple layers of the memory hierarchy. While Stacker is implemented on top of DataSpaces and it uses some of the underlying features of DataSpaces, the key data prefetching technique can be used for other data-staging designs. The key components of Stacker include the *Data Placement Module* and the *Data Object/Block Storage Layer*. These modules cooperate to provide efficient data placement and early data retrieval to facilitate data movement and sharing across HPC workflow components.

A. Data Object/Block Storage Layer

The Data Object/Block Storage Layer implements a shared space abstraction between DRAM and on-node SSDs. It communicates with the Data Placement Module to make appropriate data translation from in-memory objects to file location in persistent storage. Any request going to SSDs needs to have detailed information on the file-name and which directory it is stored. In a complex workflow, the number of data objects can scale substantially. When there are thousands or even millions of files being staged, keeping

TABLE I
METADATA INFORMATION STORED BY THE DATA STAGING SERVER FOR EACH STORED DATA OBJECT

Metadata	Description
DSG_ID	ID of the server storing the object
var_name	Name of the variable in HPC workflow
lb	Starting offset of the object in the global bounding box
ub	Ending offset of the object in the global bounding box
ver	Version number of the object

track of the requisite number of file-names severely increases the metadata complexity and makes the data staging service impossible to scale.

To overcome this problem, Stacker implements a name conversion utility. When an object is stored in data staging service, metadata information about these objects is stored in the data indexing layer. This name conversion module leverages this metadata information to programmatically generate a unique file-name per object. As an example, when an application is sent to the data staging server, the server usually stores following metadata information: DSG_ID, var_name, lb, ub and ver, which are described in Table I. The lb and ub represent the location of the object in the global bounding box or the shared-space abstraction. A bounding box is a geometric descriptor specifying the range of data along each dimension of the application domain. All of this information is concatenated into a string to create an unique file-name for each object. This object is then written as a file to the SSD using POSIX write APIs. When an object stored in SSD is requested, the file-name is functionally computed and appropriate data from that specific file is sent to the requesting client.

In data staging solutions when a data is written in the form of an n-dimensional Cartesian grid by an application, it is converted into 1-dimensional space and stored in memory. When this n-D object representation is offloaded to SSD, the serialized representation is not altered. If only a portion of the n-D object is requested by the client, this portion of n-D object must be reconstructed by reading from various offsets of the 1-D representation. There are two ways the reconstruction can be done: *1)* First copy the whole object into memory and reconstruct the requested portion. *2)* Issue multiple small reads to the SSDs to read only the data necessary for reconstruction of the requested portion. In the second case, multiple read requests are necessary since the transformation from n-D to 1-D representation will alter the data layout and thus a single sequential read request cannot be issued. While Stacker allows choice of which way to reconstruct, decision must be made prior to the deployment of the staging service. We observed that the first method of reconstruction is not scalable. If there are many reader applications requesting portions of large data objects, each of them will copy the large objects into already limited memory. This will eventually cause the applications to fill memory and stall until space becomes available. Consequently, we implement the second option in all of our test cases for serving data stored in SSD to reader applications.

B. Data Placement Module

In a coupled simulation, when a simulation application sends data to the staging service, the simulation application moves forward. Since simulation applications are generally producers of the data, they are rarely affected by choice of the layer of the memory hierarchy for data placement. The reason behind this is that staging service first receives data into memory, reports to the producer that data has been received and then internally schedules the data to be moved to lower levels of memory hierarchy. Since this data movement is decoupled from the data producer applications, these applications are not affected by the introduction of SSDs in the staging service. However, data consumers/readers are directly affected if the data they requested are stored in SSDs because the data needs to be moved to memory before they can be sent to these applications. This is where the efficient prefetching of data plays a vital role.

Since some identifiable, repeated access pattern is a requisite of being able to accurately prefetch, a prefetching algorithm should be capable of identifying the spatial correlation of the data access behavior. In this context, Lofsted et. al. [23] have demonstrated that understanding the read patterns of 2-D and 3-D domain decomposition in HPC workflows helps to increase the I/O performance of the end-to-end application. While this work follows a similar trend in trying to find the read access pattern, we aim to learn these patterns of spatial correlation dynamically.

While sequential locality-based prefetching is default behavior in most of operating systems, it is a special case of spatial locality: strided data access patterns will not be captured by this mechanism. In addition to the identification of spatial attributes, temporal attributes of the application should be identified. Stacker aims to capture both temporal and spatial attributes of the application's data access pattern in order to predict an upcoming sequence of I/O requests. In contrast to predicting only the next set of read offsets, Stacker also predicts next set of variables likely to be accessed. Motivated by the n-gram model's capability of efficient prediction with minimal training, Stacker utilizes Algorithm 1 to leverage hierarchical n-grams stored as hash tables for dynamic prefetching of variables. While the n-gram model being used in Stacker is similar to regular n-grams, Stacker maintains a hierarchical structure of variable length n in n-grams. A n-gram model of higher length gets higher priority over a shorter length n-gram model. This enables Stacker to quickly adapt to dynamic data access patterns and immediately start predicting without a need to wait for predetermined number of requests to occur for building the prediction model.

In Algorithm 1, we show only the prediction of a variable. A very similar algorithm is also utilized for predicting the lower and upper bounds of the predicted variable. Each data staging server maintains a global list of past n requests. When

a new read request arrives, n-grams to uni-grams are created from the sequence of past n requests and the current request. These grams are stored as a hash table in each staging server. As an example, let us suppose the first request is α and next request is β. In this case, a 1-gram is created with α as the current state and β as the successor state with a frequency of 1. When the next request, for γ, arrives, in addition to β as the current state with γ being the successor state in a 1-gram, a 2-gram is also created with $\alpha\beta$ for the current state and γ as successor state with frequency 1. When the same state is repeated from n to 1 grams, frequencies are increased. Thus for any incoming request, we search for its occurrence in the history table and in a decreasing fashion from n-gram to 1-gram. The candidate successor state with highest frequency will be the predicted value. Thus, we grant higher priority to higher grams rather than lower grams. Specifically, if a sequence of $\alpha \rightarrow \beta \rightarrow \gamma \rightarrow \delta$ is found for a current request of γ with past request of $\alpha \rightarrow \beta$, then we predict next request will be δ, even if $\beta \rightarrow \gamma \rightarrow \epsilon$ is present. The reason is that Stacker provides higher priority over the longer sequence of access history.

Once a variable is predicted, we also predict the bounding box of accesses. To facilitate that, we maintain n-grams for each variable where inputs are a sequence of lower and upper bounding box boundaries. We then mark the objects that fall within the predicted bounding box of the predicted variable as targets for prefetching. In one dimension, this is analogous to a standard block-based caching system, where the block granularity is variable and determined by the size of the data objects in staging. In an N-dimensional data domain, defining locality beyond the object granularity is complex. Fetching all neighboring blocks can be very expensive, while choosing a subset of blocks require an arbitrary choice of which dimensions are more important to locality. When Stacker prefetches, it does so at the granularity of the object that was written into the underlying object store. Since this object is a continuous n-dimensional bounding box, Stacker inherently preserves and maximizes the sequential data locality inside the object, while spatial/temporal locality across objects is captured by the n-gram.

When the staging server starts, it starts with 2 extra threads, where each individual thread is assigned the task of either prefetching or evicting data from memory. When a target for prefetching is identified by the staging server, the information is sent to the prefetching thread. This thread communicates with the block storage layer and performs data prefetching asynchronously. Since the prefetching happens in a separate thread, prefetching has a minimal impact on the serving of data, rather than if these tasks were interleaved by a single thread.

Stacker performs a logical partition of the memory for storing prefetched data and other data. When memory starts to become full, some prefetched data is targeted for eviction if the total data size of prefetched objects is larger than the remaining prefetch partition size. Since this is a logical partition, a staging server can use this partition to store data from write requests if necessary. A first-in-first-out (FIFO) queue is maintained for the prefetch partition, so the target for eviction is the oldest data that was prefetched. Since all of these tasks happen asynchronously and in separate threads, all operations of serving the data to the client, prefetch to memory, and eviction from memory can be overlapped. These operations allow Stacker to have a minimal overhead as compared to all in-memory storage while enabling a larger volume of data being stored in the staging area. As mentioned earlier, Stacker also needs to store the records of access history. We store these records in a hash-table as $< string, < string, value >>$ key-value pairs. To analyze the maximum number of key-value pairs in this table, let's assume that we use a max of n grams and there are k reader cores and each reader issues 1 read request in each time-step. In the worst case, there will be $n \times k$ unique key-value pairs in one time-step. To avoid an exponential growth in number of records, we periodically scrub the hash-table to limit its memory size, i.e., we remove the key-value pairs with lowest frequencies after every certain number of time-steps.

Algorithm 1: Hierarchical Prediction Algorithm

```
1  Global: Past sequence of requests [n, n-1, n-2, ...1]
     ;
2  function Predict (A);
   Input  : Requested variable(A)
   Output: B
3  create n to 1 grams from [n-1,n-2,...1,A] requests;
4  for i = n to 1 grams do
5      c=2;
6      search for occurrence of [n-c,n-c-1, n-c-2,...,1,A]
         sequence;
7      if Sequence found then
8          Return the next variable with highest
             occurrence frequency;
9      else
10         c++;
11         i—;
12     end
13 end
14 Return null;
```

IV. Evaluation

In this section, we evaluate Stacker using synthetic codes, which simulate various read patterns. We also perform large scale tests on a real scientific workflow, using the combustion DNS-LES simulation/analysis from the S3D combustion and analysis workflow [6].

Synthetic experiments were performed using the Caliburn Supercomputer at the Rutgers Discovery Informatics Institute [31]. Caliburn consists of 560 nodes, each containing Dual Intel Xeon E5-2695v4, 18-Core processors with 256 GB of

RAM and a 400GB Intel NVMe drive per node. Experiments using the real scientific workflow were performed on Titan Cray XK7 system. It has 18,688 compute nodes and each node is equipped with 16-core AMD 6200 series Opteron processor and 32 GB memory. Since the nodes of the Titan system are not equipped with SSDs, we emulated an INTEL SSD DC P4600 [7] series device by reserving a part of DRAM and introducing artificial delays for read/write requests going to this reserved area. A P4600 SSD has a read latency of $85\mu s$and $15\mu s$ write latency. It can achieve a read throughput of 3200MB/s and a write throughput of 1325MB/s. We used these numbers for the introduction of artificial delays for read/write requests to emulated SSD. All of the tests in subsequent sections are performed 3 times and the average result is reported.

A. Synthetic Experiments

Data read access rate and location of the data impacts the performance of coupled simulation/analysis workflows. To better understand the impact of various data access read patterns upon Stacker, we selected four test cases similar to [19]. In these cases, we assume that scientific applications write/read data to/from a three-dimensional global space, i.e. data domain. The data is assumed to be written over multiple iterations or time-steps, and also read in the similar fashion. We compare our results with three other methods of data staging: In-Memory (data is stored in the memory only), No-Prefetching (no data is prefetched, i.e., every read request goes to SSD), and Locality-Based (prefetching from disk to memory occurs based on the sequential locality of temporal and spatial data attributes.) Since we are evaluating data prefetching techniques and their impact, in order to make a fair comparison we flush all of the data to the SSD prior to issuing read requests. This is to prevent biasing results, as there is no data in memory for the first read request. Data movement between memory and SSD is then performed based on the prefetching technique.

In our synthetic tests, we used two application codes, namely readers and writers. As their names suggest, writers write data to the staging servers and readers read data from the staging servers. This generic end-to-end data movement behavior emulates a coupled scientific simulation, where the writer is producing intermediate simulation data and the reader is performing some analysis on it. The data was organized in a 3-dimensional Cartesian grid format with $X \times Y \times Z$ scale. Table II details the experimental setup and data volume shared via staging servers in our synthetic workflow.

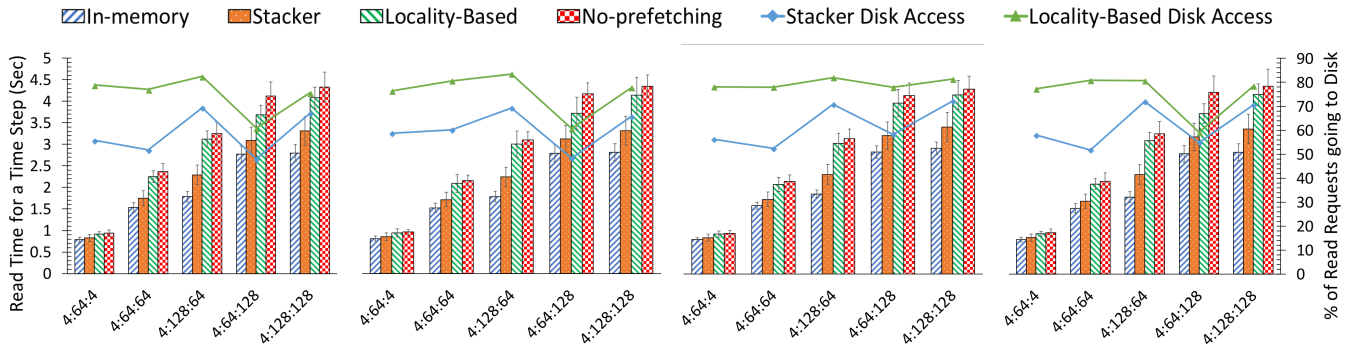
Each writer writes data across two variables, A and B, in our test cases. Half of the readers read data from variable A, while other half read data from variable B. For our tests, we limited the memory partition for prefetching to a size of 500 MB. Once this partition becomes full, all other subsequent prefetches must trigger data eviction from memory to the SSD. In all of the synthetic test cases (Figure 3(a)-3(d)), we limited the number of staging server cores to 4 and observed

TABLE II
EXPERIMENTAL SETUP CONFIGURATIONS OF NO. OF APPLICATION AND STAGING CORES, SIZE OF THE STAGED DATA AND DATA DOMAIN INFORMATION FOR SYNTHETIC TESTS.

Total No. of Cores	260
No. of Staging Cores (Nodes)	4 (1)
No. of Variables	2
No. of Parallel Writer Cores (Nodes)	128 (4)
No. of Parallel Reader Cores (Nodes)	128 (4)
Data Domain for Each Variable	$64 \times 64 \times 32768$
Memory Size Allocated for Prefetching	500 MB/Node
Total Staged Data Size (20 Time-steps)	40 GB
Max Value of n for n-grams	5

the impact of varying readers and writers. Since the staging server is memory and network communication bound, it doesn't require substantial CPU resources. Using 4 staging cores in our experimental-setup means that there are 4 staging process that are accepting read/write requests in parallel. Since prefetching techniques attempt to capture the read pattern, we varied readers from 4 to 128, while writers were kept to either 64 or 128. In all cases, it was observed that In-Memory data staging works best. This observation was expected, since in the In-Memory case, staged data only resides in memory and never incurs the overhead of SSD access. As explained in earlier sections, we are interested in workflows where keeping data in the DRAM is not an option, and this leads us to compare these optimal - but unachievable - results against various prefetching methods such as Stacker, Locality-Based prefetching and No-Prefetching techniques in order to evaluate advantages and overheads associated with integrating deeper memory hierarchy.

Figure 3(a) represents a case where reader applications read the entire data domain of every time-step. This scenario is representative of many analysis codes, e.g. visualizations [24], where all of the data is read for full resolution analysis. The average data read response time is unsuprisingly smallest when data is stored in memory only. In contrast, the read response time was largest for the data stored in the No-Prefetching (i.e. SSD-only) technique due the SSD's large read access latency and slower data throughput bandwidth than DRAM. For the case of 64 writers and 128 readers, we observed that locality-based prefetching can reduce the percentage of total read requests going to SSD by up to 40%, while for other cases of varying readers and writers around 75% of read requests require SSD access. Having higher numbers of readers than writers means that multiple readers are reading data from the same data object (each reader is reading a portion of the large object) and prefetching one object will reduce the data access latency across multiple readers. While a similar pattern is observed for Stacker, up to 51% of total read requests were served from memory instead of SSD. This reduction in SSD accesses has the results of a reduction of up to 29.41% for best case and 23.52% for worst case for read response time in comparison to No-Prefetching, while Locality-Based prefetching reduces the average read response time by up to



(a) Read entire data domain for all time-steps

(b) Read entire data domain over multiple time-steps

(c) Read entire data domain every 10 time-steps

(d) Randomly decide on which time-step to read data.

Fig. 3. Read response time and % of read requests going to disks for varying read patterns in multiple time steps for synthetic applications. The X-axis represents number of application core counts in the server:writer:reader format. Left Y-axis represents the read time while right Y-axis represents percentage of disk access.

11.61% for best case and 3.8% for worst-case scenarios.

Various analyses such as trajectory visualizations and feature tracking read data over multiple time-steps [37]. Figure 3(b) represents this case, where data is read for 20 consecutive time-steps out of every 30 time-steps. In this case, Stacker is able to identify the pattern of changing time-step immediately, while Locality-Based prefetching makes mispredictions on time-step information, and so the wrong data is prefetched. In addition to this, Stacker is also able to capture the interarrival patterns of the read requests from various reader applications, while Locality- Based prefetching only takes into account the current read request and prefetches surrounding data objects. Thus, Stacker is able to reduce the average read response time by up to 27.71%, while Locality-Based reduced it by up to only 10.8%.

In figure 3(c), reader applications read the entire data domain every certain number of time-steps. We use this case as representative of interactive visualizations [27]. Such applications require coarse temporal and spatial data resolutions. Stacker is able to outperform both the Locality-Based and the No-Prefetching techniques in all cases of varying readers and writers. While Stacker reduces read-response time by up to 26.41%, Locality-Based prefetching reduces it by up to only 4.8%. It can be observed that the Locality-Based technique is not able to capture the coarse temporal read requests and so results in the reader applications accessing SSD for around 80% of the total requests.

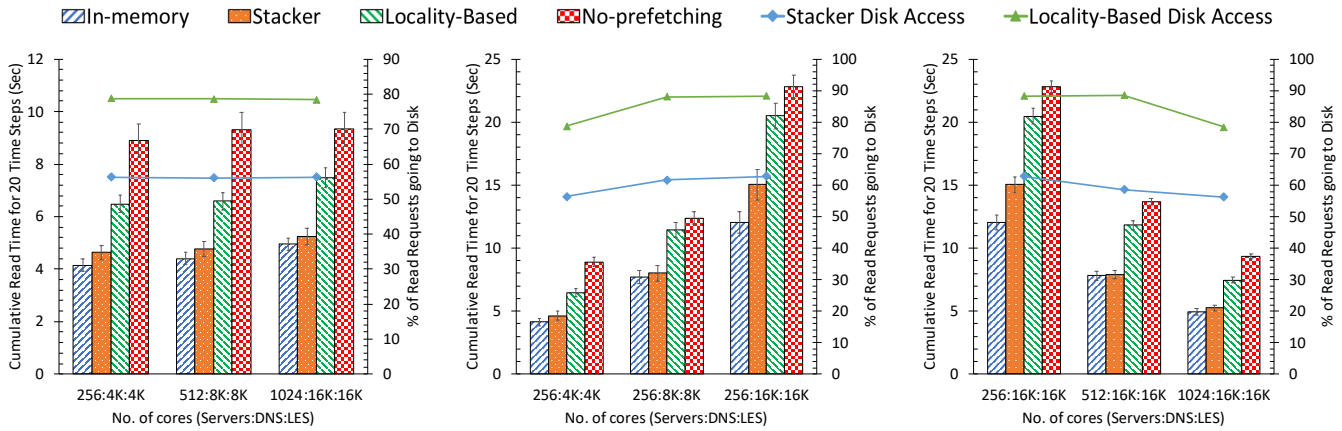
We emulated the case of an irregular temporal read access pattern in figure 3(d) by randomly choosing which time-steps of data to read. Such read access patterns can be observed in data-driven visualizations [18], [33], where applications dynamically decide whether to read a certain set of data or at run-time. Similar to previous observations, the Locality-Based technique has a high degree of misprediction, causing a higher number of SSD accesses than are strictly necessary, while Stacker is able to predict read pattern across various applications in the same time-step. The result is that Stacker outperforms both Locality-Based and No-Prefetching techniques.

In all of the above mentioned test cases, Stacker has the fastest read-response time other than In-Memory data staging. This advantage comes from the fact that, even in the same time-step, Stacker is able to capture the read patterns between variables A and B, i.e., it can predict whether variable A will be accessed after B or vice-versa. This leads Stacker to prefetch the optimal object. Since Locality-Based prefetching is unable to infer this information, it has higher rates of mis-prefetches and suffers from higher data access latency. Stacker has low overheads in terms of read-response time as compared to In-Memory data staging and for these test cases there were $\sim$ 800 total records in the hash-table for each staging server during 20 time-step runs.

B. Real Scientific Workflow

To evaluate the effectiveness of Stacker, we performed large-scale tests of a lifted hydrogen combustion simulation workflow using S3D [6]. We use "direct numerical simulations" (DNS) and "large eddy simulations" (LES) components of the S3D workflow for simulation and analysis purposes, respectively. DNS generates data and is sent to staging servers and LES reads and analyzes the stored data. The experimental configurations of our large-scale tests is listed in Table III. Our evaluations using S3D was performed with 4K, 8K and 16K analysis and simulation cores. The grid domain sizes were chosen such that each core was assigned a spatial sub-domain of size $64 \times 64 \times 64$. The earlier synthetic tests can be considered strong scaling, as the data size was kept the same across different core counts. In contrast, we keep the same data volume per DNS/LES core in following tests to perform weak scaling tests. We measured the cumulative read time for reading data over 20 time-steps.

In Figure 4(a), we show the results of increasing the numbers of data staging servers, while keeping the ratio of DNS/LES cores to servers fixed. It was observed that the percentage of read requests going to disk remains relatively



(a) Constant ratio between servers and DNS/LES cores.

(b) Decreasing ratio between servers and DNS/LES cores.

(c) Increasing ratio between servers and DNS/LES cores

Fig. 4. Cumulative read time for S3D workflow with DNS and LES coupling for varying numbers of staging servers and application core counts.

TABLE III
EXPERIMENTAL SETUP CONFIGURATIONS FOR LARGE-SCALE S3D EXPERIMENTS

Total No. of Cores	33K
No. of Staging Cores (Nodes)	1K (64)
No. of Simulation Cores (Nodes)	16K (1K)
No. of Analysis Cores (Nodes)	16K (1K)
Data Domain	$2048 \times 2048 \times 1024$
Memory Size Allocated for Prefetching	500 MB/Node
Data Size (GB)	640 GB

TABLE IV
TOTAL NUMBER OF PREFETCHES AND EVICTIONS OF 2MB OBJECTS FOR EXPERIMENTAL SETUP IN FIGURE 4(A)

Setup	Total Requests	Stacker		Locality-Based	
		Prefetch	Hit-ratio	Prefetch	Hit-ratio
256:4K:4K	77824	64768	52.50%	82237	20.11%
512:8K:8K	155648	129522	52.79%	164653	20.19%
1024:16K:16K	311296	259016	52.51%	327824	20.51%

constant for both Stacker and Locality-Based. This is expected because total numbers of client applications attached to each staging server remain constant. Since each data server makes prefeteching decisions independent of the other servers, increasing the client applications and servers in the same ratio keeps the cumulative read response time and disk access percentage fairly constant. While No-Prefetching has the worst read performance, Stacker is able to reduce the cumulative read-response time by $\sim$ 49%. In contrast, Locality-Based prefetching reduces the read-response time by $\sim$ 25% only. Table IV shows the total number of read requests issued, total number of prefetches and hit-ratio for the test case in Figure 4(a). We define hit-ratio as the ratio of total read requests served from memory to total prefetches. It can be seen that, while Locality-Based method prefetches a greater number of objects, its hit ratio is relatively low. In addition to prefetching the wrong data, more prefetching leads to more evictions in a limited memory environment, and this degrades the overall performance of the workflow. As a specific example, the Locality-Based method caused 18,273 total evictions from memory to SSD, while Stacker only had 768 evictions for the 256:4K:4K setup of Servers:DNS:LES cores.

To evaluate the impact of an increase in client processes on Stacker, we increased the processes count of the DNS and LES applications from 4K to 16K while keeping the server core count to 256. Since the data-per-process size is kept constant, i.e, weak scaling is done, the cumulative read response time increases as we increase client cores. It can be seen that, in Stacker, $\sim$ 60% of access is served from SSDs, while Locality-Based prefetching causes $\sim$ 80% of accesses to go to SSDs. From this test, we can observe that Stacker is scalable, while Locality-Based prefetching will degrades with increasing scale, due to its inability to capture the pattern between different application clients. As evidenced by the results, Stacker can capture the interleaving pattern of application cores for accessing data, while Locality-Based approach only captures the sequential locality as perceived by the staging server and thus has a relatively small performance improvement *vs* No-Prefetching.

Figure 4(c) illustrates the case of the reduction in the number of clients attached to each staging server. Since we increase the number of servers while keeping DNS/LES cores fixed at 16K, more requests are served in parallel and we see a decrease in the cumulative read response time. Stacker improves the read performance by $\sim$ 40%, while Locality-Based prefetching improves it by $\sim$ 15% on average for varying number of servers. It can be attributed to the fact that Locality-Based has a high number of requests being served through SSDs.

From our synthetic and real scientific workflow simulations, we can infer that Locality-Based prefetching does not have the capability to identify the read access pattern spanning across multiple variables. Even in the case of a single variable being accessed, the interleaving of read requests from multiple applications changes the spatial locality on every read request and makes Locality-Based prefetching perform poorly. On contrary, Stacker is able to tackle these cases easily and provide an improvement of up to 55% read performance in comparison to No-Prefetching.

The hash-table size in Stacker primarily depends upon the length of n in n-gram. Each hash-table in Stacker had a max of around $\sim 1.3K$ records during S3D experimental runs with n set to 5. In summary, Stacker can effectively provide high-performance data staging by effectively prefetching data from high latency storage device to DRAM, with minimal overhead of maintaining hash-table of access history.

V. Related Work

The recent advancements in both in-situ and in-transit paradigms have led towards the development of various data staging solutions. Data staging frameworks, such as DataSpaces [13], ActiveSpaces [12], DataStager [1], provide services to support memory-based data staging solutions for in-situ and in-transit HPC applications. These staging frameworks use RDMA to achieve high data transfer throughput from client applications to the staging servers. Unfortunately, these frameworks utilize DRAM memory for staging all data and processing. In contrast to these frameworks, our solution uses an SSD-based data staging solution, which performs intelligent and autonomous data prefetching to guarantee high storage capacity with low impact on the application performance.

There have been several efforts, such as CloudCache [2] and vCacheShare [25], to integrate flash-based devices as a cache device for single/multiple nodes. There has been increasing interest in using burst buffers in HPC systems for optimizing the I/O path of data-intensive workflows. A storage system for HPC to integrate a tier of solid-state burst buffers into the storage system for absorbing application I/O requests was explored in [22]. TRIO [35] explores on efficiently moving large checkpointing datasets to PFS by utilizing the burst buffers. Data Elevator [14] also utilizes burst buffers as a fast persistent storage layer and asynchronously transfers data to the final destination in the background in order to enable autonomous data movement across storage hierarchy layers. In contrast to these research efforts that optimize the write path from the fast storage layer to the slower storage layer, we strive to enable fast data retrieval from SSDs for timely insight.

The use of Markov models to prefetch data from disks to memory has been studied on Lynx [21]. Curewitz et al. [8] have also studied the concept of data prefetching via data compression using n-th order Markov chains. While Stacker uses the concept of n-grams, which are similar to the n-th order Markov Chains in Lynx [21] and prediction by partial match (PPM) in [8], Lynx and PPM build a transition table using a single n-th order Markov chain. In contrast, Stacker uses a combined approach, where multiple higher-order n-gram models are organized in a stepwise manner. Lynx replaces the default sequential prefetching of the Linux kernel, while Stacker sits at the application level. Stacker prefetches large size objects, while Lynx and PPM prefetch at the granularity of 4KB objects as they predict the next page access. Stacker preserves the inherent sequential locality of page access inside objects, while Lynx and PPM breaks this locality because their predictions occur with page size granularity. In addition, tracking and maintaining information about individual pages at the application level (i.e., in data staging) creates very large amounts of metadata in a memory-constrained setting, which is not feasible. While PPM and Lynx operate at the granularity of files and pages, Stacker predicts at the granularity of object variables and bounding boxes. In summary, Stacker is largely orthogonal to Lynx and PPM.

Jin et al. [19] have explored utilizing both DRAM and SSDs for data staging and aim to reduce the overhead of reading data from the lower level of the memory hierarchy. While the end goal is similar, [19] depends on user-provided hints to identify the data access patterns, which is infeasible for complex data-intensive workflows. The work stores history information of each object being staged in data staging to calculate its likelihood of access on next time-step, which can become a serious bottleneck if there are a large number of small objects in the staging area. On the other hand, Stacker does not depend upon user inputs/hints and only stores sequence of accesses rather than information about individual objects.

VI. Conclusion and Future Work

While data staging frameworks have emerged as effective solutions for data management in in-situ and in-transit workflows, they are designed for storing data in memory only. In this paper, we designed Stacker, which tiers data across SSD and DRAM to enable high capacity data staging. Stacker also implements autonomous data movement across these tiers by predicting the upcoming sequence of data access requests. We implemented Stacker based on the DataSpaces framework and evaluated it on the Caliburn at Rutgers University and Titan Cray XK7 at OLCF. Our experimental results, using both large-scale S3D experiments and synthetic test cases, demonstrate that Stacker efficiently prefetches data from SSDs to memory for a variety of data access patterns. The source code of Stacker is available at https://github.com/pradsubedi/Stacker.git. As future work, we plan to expand Stacker to support multiple prefetching algorithms and dynamically decide which algorithms to use based on workflow requirements. We also plan to use such algorithms for optimizing the write-paths of workflows.

Acknowledgement

We would like to thank all of the reviewers for their valuable feedback and comments. The research presented in this paper is based upon work by the RAPIDS Institute supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program and by the SIRIUS grant (number DE-SC0015160), and by the National Science Foundation (NSF) via grants number IIS 1546145. The research at Rutgers was conducted as part of the Rutgers Discovery Informatics Institute (RDI^2).

REFERENCES

[1] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng. Datastager: scalable data staging services for petascale applications. *Cluster Computing*, 13(3):277–290, 2010.

[2] D. Arteaga, J. Cabrera, J. Xu, S. Sundararaman, and M. Zhao. Cloudcache: On-demand flash cache management for cloud computing. In *FAST*, pages 355–369, 2016.

[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*, page 49. IEEE Computer Society Press, 2012.

[4] P. F. Brown, P. V. deSouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai. Class-based n-gram models of natural language. *Comput. Linguist.*, 18(4):467–479, Dec. 1992.

[5] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems (TOCS)*, 14(4):311–343, 1996.

[6] J. H. Chen, A. Choudhary, B. De Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W.-K. Liao, K.-L. Ma, J. Mellor-Crummey, N. Podhorszki, et al. Terascale direct numerical simulations of turbulent combustion using s3d. *Computational Science & Discovery*, 2(1):015001, 2009.

[7] I. Corporation. Intel ssd dc p4600 series, 2018.

[8] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 257–266, New York, NY, USA, 1993. ACM.

[9] M. Damashek. Gauging similarity with n-grams: Language-independent categorization of text. *Science*, 267(5199):843–848, 1995.

[10] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good. The cost of doing science on the cloud: the montage example. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12. Ieee, 2008.

[11] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang. Diskseen: Exploiting disk layout and access history to enhance i/o prefetch. In *USENIX Annual Technical Conference*, volume 7, pages 261–274, 2007.

[12] C. Docan, M. Parashar, J. Cummings, and S. Klasky. Moving the code to the data-dynamic code deployment using activespaces. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 758–769. IEEE, 2011.

[13] C. Docan, M. Parashar, and S. Klasky. Dataspaces: an interaction and coordination framework for coupled simulation workflows. *Cluster Computing*, 15(2):163–181, Jun 2012.

[14] B. Dong, S. Byna, K. Wu, H. Johansen, J. N. Johnson, N. Keen, et al. Data elevator: Low-contention data movement in hierarchical storage system. In *High Performance Computing (HiPC), 2016 IEEE 23rd International Conference on*, pages 152–161. IEEE, 2016.

[15] E. F. D'Azevedo, J. Lang, P. H. Worley, S. A. Ethier, S.-H. Ku, and C. Chang. Hybrid mpi/openmp/gpu parallelization of xgc1 fusion simulation code. In *Supercomputing Conference 2013*, pages 1441–1441, 2013.

[16] K. S. Grimsrud, J. K. Archibald, and B. E. Nelson. Multiple prefetch adaptive disk caching. *IEEE Transactions on Knowledge and Data Engineering*, 5(1):88–103, 1993.

[17] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, et al. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

[18] M. C. Hao, U. Dayal, D. A. Keim, and T. Schreck. Importance-driven visualization layouts for large time series data. In *Information Visualization, 2005. INFOVIS 2005. IEEE Symposium on*, pages 203–210. IEEE, 2005.

[19] T. Jin, F. Zhang, Q. Sun, H. Bui, M. Romanus, N. Podhorszki, S. Klasky, H. Kolla, J. Chen, R. Hager, et al. Exploring data staging across deep memory hierarchies for coupled data intensive simulation workflows. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 1033–1042. IEEE, 2015.

[20] S. Ku, C. Chang, and P. Diamond. Full-f gyrokinetic particle simulation of centrally heated global itg turbulence from magnetic axis to edge pedestal top in a realistic tokamak geometry. *Nuclear Fusion*, 49(11):115021, 2009.

[21] A. Laga, J. Boukhobza, M. Koskas, and F. Singhoff. Lynx: A learning linux prefetching mechanism for ssd performance model. In *Non-Volatile Memory Systems and Applications Symposium (NVMSA), 2016 5th*, pages 1–6. IEEE, 2016.

[22] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. On the role of burst buffers in leadership-class storage systems. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–11. IEEE, 2012.

[23] J. Lofstead, M. Polte, G. Gibson, S. Klasky, K. Schwan, R. Oldfield, M. Wolf, and Q. Liu. Six degrees of scientific data: reading patterns for extreme scale science io. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 49–60. ACM, 2011.

[24] K.-L. Ma. In situ visualization at extreme scale: Challenges and opportunities. *IEEE Computer Graphics and Applications*, 29(6):14–19, 2009.

[25] F. Meng, L. Zhou, X. Ma, S. Uttamchandani, and D. Liu. vcacheshare: Automated server flash cache space management in a virtualization environment. In *USENIX Annual Technical Conference*, pages 133–144, 2014.

[26] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *ACM Sigplan Notices*, volume 27, pages 62–73. ACM, 1992.

[27] P. Pebay, D. Thompson, and J. Bennett. Computing contingency statistics in parallel: Design trade-offs and limiting cases. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pages 156–165. IEEE, 2010.

[28] M. Romanus, F. Zhang, T. Jin, Q. Sun, H. Bui, M. Parashar, J. Choi, S. Janhunen, R. Hager, S. Klasky, et al. Persistent data staging services for data intensive in-situ scientific workflows. In *Proceedings of the ACM International Workshop on Data-Intensive Distributed Computing*, pages 37–44. ACM, 2016.

[29] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Gather-scatter dram: in-dram address translation to improve the spatial locality of non-unit strided accesses. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 267–280. ACM, 2015.

[30] Z. Su, Q. Yang, Y. Lu, and H. Zhang. Whatnext: A prediction system for web requests using n-gram sequence models. In *Web Information Systems Engineering, 2000. Proceedings of the First International Conference on*, volume 1, pages 214–221. IEEE, 2000.

[31] R. University. Caliburn user guide, 2018.

[32] J. S. Vetter and S. Mittal. Opportunities for nonvolatile memory systems in extreme-scale high-performance computing. *Computing in Science & Engineering*, 17(2):73–82, 2015.

[33] I. Viola, A. Kanitsar, and M. E. Groller. Importance-driven feature enhancement in volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 11(4):408–418, 2005.

[34] L. Wan, M. Wolf, F. Wang, J. Y. Choi, G. Ostrouchov, and S. Klasky. Analysis and modeling of the end-to-end i/o performance on olcf's titan supercomputer. In *IEEE 19th International Conference on High Performance Computing and Communications*, pages 1–9. IEEE, 2017.

[35] T. Wang, S. Oral, M. Pritchard, B. Wang, and W. Yu. Trio: Burst buffer based i/o orchestration. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 194–203. IEEE, 2015.

[36] X. Wang, A. McCallum, and X. Wei. Topical n-grams: Phrase and topic discovery, with an application to information retrieval. In *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on*, pages 697–702. IEEE, 2007.

[37] J. Wei, H. Yu, R. W. Grout, J. H. Chen, and K.-L. Ma. Dual space analysis of turbulent combustion particle data. In *Visualization Symposium (PacificVis), 2011 IEEE Pacific*, pages 91–98. IEEE, 2011.

[38] 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*, 28(2-3):241–251, 2013.

[39] G. Zhang, B. E. Patuwo, and M. Y. Hu. Forecasting with artificial neural networks:: The state of the art. *International journal of forecasting*, 14(1):35–62, 1998.