University of Nevada, Reno

# Streaming Transfer Optimization for Distributed Science Workflows

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering

by

Davut Ucar

Engin Arslan/Thesis Advisor

May, 2020

**THE GRADUATE SCHOOL**

We recommend that the thesis
prepared under our supervision by

**DAVUT UCAR**

entitled

**Streaming Transfer Optimization for**
**Distributed Science Workflows**

be accepted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Engin Arslan, Ph.D., Advisor

Sushil J. Louis, Ph.D., Committee Member

Mohammed Ben-Idris, Ph.D., Graduate School Representative

David W. Zeh, Ph.D., Dean, Graduate School

May 2020

## ABSTRACT

Driven by advancements in computing and sensing technology, many scientific projects started to generate a huge volume of data that reaches to petabytes in scale. Distributed and collaborative nature of these projects requires produced data to be streamed to geographically distributed locations in a timely manner to enable real-time (or near-real time) processing. Thus, robust and predictable network performance is key to streamline end-to-end workflow execution of streaming projects. On the other hand, existing high-performance data transfer applications and services support "best-effort" Quality of Service (QoS) model, thus fail to sustain high transfer performance when transfer conditions (e.g., dataset characteristics, background traffic, etc.) deviate from initial observations., making them ill-suited for streaming workflows with stringent performance requirements.

In this thesis, we propose *FStream* to offer performance guarantees to time-sensitive streaming applications by continuously monitoring transfer performance and re-adjusting transfer settings to adapt to dynamic transfer conditions and sustain high transfer performance throughout the workflow execution. To achieve this goal in real-time, *FStream* employs *online profiling* to rapidly explore solution space of transfer settings and find the one that meets QoS requirements with minimal overhead. It also takes advantage of the long-running nature of streaming workflows and keeps track of past online profiling results to greatly reduce convergence time of future online profiling runs. We evaluated the performance of *FStream* by transferring several synthetic and real-world workloads using high-performance production networks and showed that it offers up to an order of magnitude performance improvement over state-of-the-art solutions.

# TABLE OF CONTENTS


**List of Tables** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **iv**

**List of Figures** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **v**

**Chapter 1: Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1**

**Chapter 2: Background and Related Work** . . . . . . . . . . . . . . . . . . . . **4**

**Chapter 3: System Design and Methodology** . . . . . . . . . . . . . . . . . . . . **10**

3.1 *FStream* Client Design . . . . . . . . . . . . . . . . . . . . . . . . . . 10

3.2 Static Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

3.3 Dynamic Tuning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

3.4 Online Profiling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

3.5 Historical Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

3.6 Quality of Service Support . . . . . . . . . . . . . . . . . . . . . . . . . 17

**Chapter 4: Evaluation and Results** . . . . . . . . . . . . . . . . . . . . . . . . . **19**

4.1 Dataset Characteristics . . . . . . . . . . . . . . . . . . . . . . . . . . 19

4.2 Experimental Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

4.3 *FStream* Performance Analysis . . . . . . . . . . . . . . . . . . . . . . 28

4.4 Tunable Parameters of *FStream* . . . . . . . . . . . . . . . . . . . . . . . . 29

4.5 Quality of Service with *FStream* . . . . . . . . . . . . . . . . . . . . . . . 31

**Chapter 5: Conclusion and Future Work . . . . . . . . . . . . . . . . . . . . . . . 33**

**References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38**

# LIST OF TABLES

4.1 File types and arrival orders for synthetic datasets. File sizes range between $0-10$ MB (Extra Small), 10-100 MB (Small), 100 MB-2 GB (Large) and $>2$ GB (Extra Large). . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

4.2 File characteristics and arrival orders of generated SRA and LCLS datasets. File sizes range between 0-10 MB (Extra Small), 10-100 MB (Small), 100 MB-2 GB (Large) and $>2$ GB (Extra Large). . . . . . . . . . . . . . . . . . 21

# LIST OF FIGURES

3.1 Flow of operations in *FStream* . . . . . . . . . . . . . . . . . . . . . . . . 11

3.2 Workflow of Smart Channel Distribution . . . . . . . . . . . . . . . . . . . . 13

4.1 File size distribution of SRA dataset. . . . . . . . . . . . . . . . . . . . . . . 19

4.2 Transfer performance comparison of algorithms in Stampede-Comet (a), OSG-Stampede (b), and Bridges-Comet (c) networks for six workload types. 23

4.3 Instantaneous throughput and concurrency changes of static and *FStream* for Workload $1-3$ in Stampede-Comet transfers. . . . . . . . . . . . . . . . . . 25

4.4 Instantaneous throughput results for Workload $\#4$ and real-world workloads in Stampede-Comet transfers. . . . . . . . . . . . . . . . . . . . . . . . 26

4.5 Breakdown of *FStream*'s performance into its main components; dynamic tuning (a), online profiling (b), and historical analysis(c) . . . . . . . . . . . 27

4.6 Instantaneous throughput and concurrency changes of static and *FStream* for synthetic and real-world workloads in Stampede-Comet transfers. . . . . . 30

4.7 Instantaneous throughput and concurrency changes of QoS transfers for Small, Large and Mixed (Small+Large) clusters. . . . . . . . . . . . . . . . 32

# CHAPTER 1
# INTRODUCTION

Advancements in instrument technologies and computing power paved the way for many scientific [1, 2, 3, 4, 5] and commercial applications [6, 7] to generate large volumes of data. For example, a large collaborative high energy physics project ATLAS [5] produces 1 Petabyte (PB) of data every second during an observation, which is reduced to 1-2 Gigabyte (GB) after filtering. Another large collaborative project Dark Energy Survey (DES) [4] captures the pictures of the southern sky and generates 500 GB of data each night. The successor of DES, Large Sky Survey Telescope (LSST) [8], will use 3.2 billion pixel camera to take high-definition images of the sky every night for 10 years and is expected to produce 50 TB of data every night.

This massive volumes of data often needs be streamed to remote facilities to be processed in real-time (or near real-time) to enable next-generation scientific discoveries. As an example, LSST observatory is built in Chile, but the captured images will be streamed to National Center for Supercomputing Applications in the Illinois, U.S. to be processed and stored. Similarly, Laser Interferometer Gravitational-Wave Observatories (LIGO) [1] are located in Hanford (Washington, US) and Livingstone (Louisiana, US), while captured data is transferred to more than ten institutions across the US and Europe in real-time to be analyzed. In addition to geographical separation of data source and processing facilities, multidisciplinary and collaborative nature of large science projects also requires data to be streamed to across the world. For example, the Dark Energy Survey project involves more than 400 scientists from 25 institutions in the United States, the United Kingdom, Brazil, Germany, Switzerland, and Australia, thus gathered data is moved across the globe to enable collaboration.

There are both proprietary (e.g., Google MillWheel [9] and Amazon Kinesis [10]) and open source (e.g., Apache Storm [11]) solutions to stream data from its source to arbitrary number of destinations. However, they are mainly designed for internet-scale applications which differ from large scientific applications in several ways, such as data scale and computational demands [12]. To overcome performance limitations, researchers proposed high-speed data transfer applications that can scale to tens of gigabytes-per-second speeds in high-performance networks [13, 14, 15, 16, 17]. However, these solutions lack the agility required to adapt to changing transfer conditions. In particular, dataset characteristics of streaming transfers might vary over time (e.g., few large files to lots of small files), which necessitates transfer settings to be re-calibrated to sustain high performance. Otherwise, fluctuating transfer rates may disrupt workflow execution by slowing down data ingestion or overwhelming computing resources.

In this study, we propose *FStream* to enhance the quality of service for streaming file transfers through dynamic transfer tuning, online performance profiling, and historical analysis. *FStream* adopts heuristic approach [18, 15] to determine the values for transfer settings, however, it incorporates dynamic tuning mechanism to periodically re-calibrate them to adapt to changing dataset characteristics. While dynamic tuning offers performance improvement over the "fixed" configuration, its dependence on heuristic algorithms inherently limits the performance gain. To overcome this limitation, *FStream* also employs online profiling to explore search space in real-time for a subset of parameters. The evaluation results show that online profiling is crucial to effectively utilize available capacity in high-performance networks and meet the performance requirements of delay-sensitive distributed workflows. *FStream* further exploits the long-running nature of streaming workflows to keep the history of earlier online profiling results to significantly reduces optimization time and improves overall transfer performance. In addition best-effort optimizations, *FStream* also supports QoS policies for transfer throughput in a way that it will search for

a transfer setting that can maintain transfer throughput in a desired range. In summary, we make the following contributions in this thesis:

- We propose *FStream* to offer robust transfer performance for streaming workflows through dynamic tuning, online profiling, and historical data analysis.
- We introduce quality of service (QoS) support for delay sensitive streaming workflows to achieve reliable transfer performance in the presence of unreliable network and end system conditions.
- We carry out extensive experiments in high-speed networks using both synthetic and real-world datasets to compare the performance of *FStream* against the state-of-the-art transfer applications and services.

The rest of the thesis is organized as follows: Chapter 2 presents the related work and background for transfer parameters. Chapter 3 describes the details of *FStream* and Chapter 4 presents evaluation results. Finally, we conclude the thesis with a summary and potential future directions in Chapter 5.

# CHAPTER 2
# BACKGROUND AND RELATED WORK

Researchers proposed congestion control algorithms for high-speed data transfers such as HSTCP [19], FastTCP [20], and Compound TCP [21]. Limited adoption of these protocols when combined with the emergence of non-network related performance issues (e.g., I/O read and write) have led the development of application-layer optimization techniques. They increase the transfer performance by tuning application-layer transfer parameters parallel network connections [22, 23], concurrent file transfers [17], command pipelining [24], buffer size [25], and I/O block size [26]. Previous studies showed that These parameters can significantly improve the end-to-end data transfer performance [18, 17].

Among the application layer protocol parameters, command pipelining (i.e., pipelining), parallel network connections (i.e., parallelism) and concurrent file transfers (i.e., concurrency) are found to be the most effective ones to enhance transfer performance [17, 15, 15, 27, 18].

Below we give definition and some insight for these three parameters:

- **Pipelining**: Pipelining parameter allows a protocol to send a sequence of data without getting response from receiver. When pipelining is not used, sender transmits data and waits in idle state until corresponding response from receiver reaches to sender. This results in under-utilization of the medium due to the delay between consecutive transfers. Pipelining also prevents TCP to go back slow start phase by keeping the window size in higher values because of ongoing transfer. Especially, when file sizes are small it is beneficiary to send multiple files in sequence by using pipelining, so the network is used constantly. Otherwise, transfer of datasets with mostly small sizes will always have less throughput compared to with larger datasets.

- **Parallelism**: Parallelism is the parameter which decides on how many streams will one file be divided over a network. For example, when parallelism value is 5, that means a file will sending via these 5 streams. This method is especially useful for larger files when buffer size is inadequate for filling all bandwidth. However, for small files parallelism causes more burden than bandwidth usage benefit. Even for large files, using more than enough parallel streams would lead overhead which makes the benefit unimportant. Hence, it should be carefully set with respect to file types in a dataset. It also helps to use available bandwidth more efficient in the beginning of the transfer. A single stream will use a small part of bandwidth due to slow start phase of TCP but several streams will increase the usage of bandwidth in this phase.

- **Concurrency**: Concurrency is the number of files that will be transferred at the same time. We use GridFTP for high performance transfers and it has some fundamental differences between concurrency and parallelism which affect their comparative performance. Parallelism is used for single file while concurrency can be used to transfer several files. Also, concurrency creates multiple processes for reading the data while parallelism use only one thread for reading data. When files are small in size, concurrency can help for better usage of bandwidth. It is also beneficiary to use with parallel file systems which provides parallel reads from disk.

Most of the channel based transfer protocols require the completion of the entire transfer and be acknowledged before the execute next transfer command. These requirements cause delays between two transfers due to the round trip time (RTT) between individual transfers. Therefore, sending individual small-sized files through the network can cause undesirable delays during transfers. Pipelining solves this issue via queuing up multiple transfer commands at the server. Parallelism is a parameter that is suitable for transferring large-sized files as multiple partitions over the network. Thus, achieving desirable

throughput with large files is possible with optimizing of parallelism. Lastly, concurrency is referring to sending multiple files over the network simultaneously over the transfer channels. Both small and large files have the benefit of using concurrency but, opening too many channels for transferring multiple files could cause decreasing of throughput [16]. However, optimal values of these parameters vary depending on file characteristics(i.e. file size and the number of files), network settings(i.e. bandwidth, round-trip-time, and background traffic on the network), and end-system configurations (i.e. file system and transfer protocol chosen) [14]. Therefore, finding the best parameter sets for every transfer is a challenging task.

Previous attempts to tune these application-layer transfer configurations can be categorized into four groups as heuristic models [13, 15], supervised [23, 27, 14], semi-supervised [14], and online [26, 28, 17] learning algorithms. Heuristic models generally obtain better throughput compared to baseline configurations, but they fail to sustain good performance in all conditions as it depends on many factors that heuristics are unable to incorporate into the model, such as network and disk interference. Supervised models can yield close-to-optimal transfer performance in the networks that they are trained for; however, deriving an accurate model requires large amount of historical data for different conditions (e.g., file size, file count, RTT, bandwidth and traffic condition), which may take weeks or months. Yet, they need to be periodically re-trained with recent data to adapt system configuration changes (e.g., file system upgrade). Online learning solutions offer promising alternatives as they can discover optimal parameter configuration in the runtime. However, existing solutions in this domain fall short to offer a practical option for streaming transfers as convergence time may take hours.

Globus is a web service to schedule large data transfers [13], which is widely-adopted in research community. However, previous work showed that it falls short to achieve optimal performance in most networks due to relying on simple heuristic to estimate transfer

**Algorithm 1** — Calculation of protocol parameter values via heuristic method

1: **function** FINDOPTIMALPARAMETERS(avgFileSize, BDP, bufferSize, maxCC)
2: $pipelining = \frac{BDP}{avgFileSize}$
3: $parallelism = Min(\lceil \frac{BDP}{bufferSize} \rceil, \lceil \frac{avgFileSize}{bufferSize} \rceil)$
4: $concurrency = Min(Max(\frac{BDP}{avgFileSize}, 2), maxCC)$
5: **return** (pipelining, parallelism, concurrency)
6: **end function**

settings [14]. Yun et al. proposed ProbData [28] to tune the number of parallel streams and buffer size for memory-to-memory TCP transfers using stochastic approximation. ProbData is able to explore the near-optimal configurations through sample transfers but it takes several hours to converge which makes it impractical to use for the majority of transfers in production high-speed networks as they last less in the order of minutes [29]. Also, we have observed that background traffic changes drastically over several hours in shared networks [14], so it may even fail to converge due to large variations in sample transfers. Rao et al. [26] presented the *D-W* method to tune the number of parallel flows using stochastic gradient descent. It works by sampling $w$ configurations for $d$ repetitions in each step. The performance of *D-W* algorithm heavily depends on initial configuration setting as well as $d$ and $w$ values which requires domain knowledge to set properly. Similar to ProbData, *D-W* algorithm also focuses on profiling for memory transfers in dedicated networks, this is not a viable option for streaming file transfers whose transfer conditions (e.g., file size, file count) may evolve over time.

In previous work [14, 18], we proposed heuristic and historical data based solutions to tune transfer settings for batch transfers. The heuristic algorithm estimates transfer settings based on network (i.e., bandwidth and RTT) and dataset characteristics (i.e., file size, number of files) in a way that it returns high pipelining and concurrency values for small files and high parallelism for large files if TCP buffer size imposes restrictions. Historical data based model, HARP, on the other hand, collects a large amount of training data under various transfer conditions (e.g., traffic type, file size, etc.) to train regression models. These models are then solved for the maximum transfer throughput to estimate optimal transfer settings. While one-time transfer tuning solutions such as the heuristic and HARP perform

well for batch transfers, they do not fit well for streaming data transfers as transfer conditions may deviate from initial settings. For example, if the streaming applications initially creates a dataset with the predominantly small files, the heuristic and HARP would create large number channels (aka concurrency) which would stay active even if the streaming applications transitions to few large files or alternates between small and large files. This in turn would only overload end systems if dataset becomes large file dominant as similar performance could be achieved with much less channels when dataset is dominated by large files. On the contrary, if the application first generates large files, then the heuristic will create fewer channels and cause bottleneck when dataset becomes small file dominated. As a result, streaming applications require dynamic transfer tuning to adapt to changing dataset settings while minimizing system overhead.

Bhat et al. proposed adaptive data buffering approach for streaming file transfer in fusion simulation workflow[30]. The proposed approach models the data generation rate by the simulation workflow and use this information to make data buffering decision based on network performance to avoid data loss. For example, when background traffic intensifies, the transfer speed falls below data generation speed, necessitating storing excess data in local storage until network performance recovers. While this paper aims to address the network performance issues by calibrating local storage settings, we take a different approach and aim to sustain transfer performance when transfer throughput degrades due to changing dataset characteristics or increasing background traffic. In another study, Branson et al. introduced cooperative data streaming for user-defined inquiries that utilizes independent, autonomous, and potentially heterogeneous sites [31]. To achieve this, they create independent jobs from user inquiries and submit them to different sites to take advantage of computing power of multiple sites for streaming applications. Aside from the throughput optimization, researchers also studied energy consumption [32, 33], integrity verification [34, 35, 36, 37], and scheduling [38] topics for high-performance data trans-

fers.

# CHAPTER 3
# SYSTEM DESIGN AND METHODOLOGY

As we aim to sustain high-performance data transfers for streaming file transfer between source and destination sites, we developed *FStream* that periodically checks source path to find new files and tunes transfer parameters to adapt to changing transfer conditions. This section details the design principles of *FStream* to achieve this goal.

## 3.1 *FStream* Client Design

As opposed to batch transfers, data in streaming workflows arrive over time. Therefore, we designed and developed the *FStream* that can meet with stream file transfer requirements. In our design, the client data finder checks the source path to find transferable new files periodically as shown in Figure 3.1. Once dataset information is gathered, if there is no initial transfer yet, the client starts initial transfers with establishing connections between sites. It also analyzes file characteristics and conducts data clustering to separate small and large files. Files that are smaller than $bandwidth/8$ are placed into "small" file cluster and the rest is to "large" cluster such that any files whose transfer would run less than $0.4$ second ($\frac{fileSize*8}{bandwidth} < 0.4$) is considered small. After clustering complete, the client sets different transfer parameters using the heuristic approach (Algorithm 3) since using the same setting for small and large files can lead to sub-optimal performance [15]. Thus, the system starts initial transfer between sites. Also, the client keeps checking the source path for new upcoming files every $20$ seconds.

In case there is an existing transfer running and also, if there are new files that appeared at the source path that need to transfer, the client collects the file list and characteristics from the source path. After that, it continues with controlling the existing clusters and

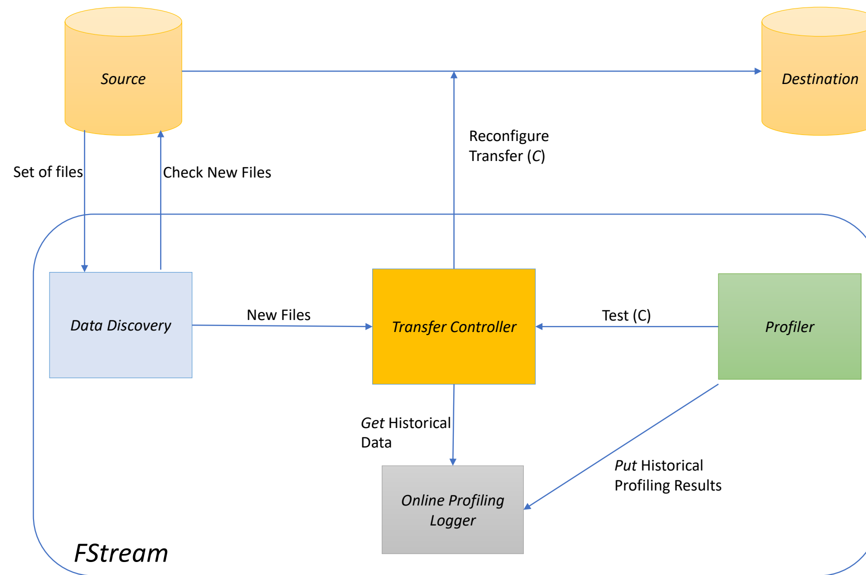



Figure 3.1: Flow of operations in *FStream*

merging the new files within the existing clusters. If there is no suitable cluster for new files (for example, there is the only *large* files exist in transfer) then the client creates a new cluster accordingly. After completion of merging the files, the client assigns new parameters according to one of the chosen algorithm that we explained in detail below. Thus, the system accomplishes smooth stream file transfer between high-end computers.

## 3.2 Static Approach

In the static approach, we used the heuristic algorithm(Algorithm 3) to find near-optimal parameters for transfers. While transferring files that have different characteristics required different sets of parameters for achieving optimal throughput. In general, using correct pipelining settings could increase throughput value significantly for small files. Dividing large files into the small parts and transferring in multiple parallel streaming channels helps to increase the achievable throughput. Both small and large files transfer can be utilized via using concurrency. In this approach, the client splits the mixed sized dataset into different-

sized clusters and assigns different parameters set to these clusters.

The transfer client checks the source periodically for upcoming files for transferring as explained before. After detecting new files at the source path, then transfer control module (as shown in Figure 3.1) creates clusters and starts transferring. When new data arrived in the source then the client can merge the data set with existing/non-transferred data clusters and reshape the clusters according to file sizes. Also, the client creates a new cluster in case there is not an available cluster that exists. After partitioning files, the client uses initial settings for the rest of the transfer for every new dataset. Since the heuristic algorithm designed for batch transferring, it required to adapt to the stream file transfer environment for a fair comparison. Therefore, we needed to adjust this algorithm to transferring files in a dynamically changing file environment without changing the algorithm logic. Thus, while we perform performance tests we didn't change initial transfer parameters during existing transfer sessions.

Among the application layer parameters, pipelining and concurrency are the most effective parameters to enhance throughput during small file transfers. Therefore, choosing the correct parameter sets for the small file transfer is crucial. We used bandwidth-delay-product (BDP) and average file size of each cluster (line $2$ in Algorithm 3) to decide to set pipelining. To set parallelism value we considered the BDP, average file size, and the TCP buffer size(line $3$ in Algorithm 3) and to decide the concurrency level we used BDP, file count, average file size and maximum concurrency level(line $4$ in Algorithm 3).

## 3.3 Dynamic Tuning

As new files are detected by the data finder, *FStream* combines them with the existing files in transfer queue. Since characteristics of new files could be different than the existing ones, *FStream* re-runs data clustering to create new file clusters and then estimates new parameters $(cc_1, p_1, pp_1)$ using Algorithm 3. If new settings are different than the current

ones, *FStream* takes following steps to transition to new settings: If desired channel count ($cc_1$) is less than existing one ($cc_0$), then extra channels ($cc_0 - cc_1$) are marked as "passive"– not actively used for transfers but kept alive for potential use in the future–. Otherwise, ($cc_1 - cc_0$) new channels are added to accommodate new configuration. Since parallelism value of a channel cannot be updated after it is created, it creates a hurdle to reuse existing channels unless their parallelism, $p_0$ matches with requested parallelism, $p_1$. Therefore, *FStream* first identifies current channels, $cc'$, whose parallelism value matches with the requested channels such that they can be reassigned after updating their pipelining value. If the remaining of existing channel count ($cc_0 - cc'$) is less than desired level ($cc_1 - cc'$), then all remaining channels will be restarted with parallelism $p_1$ in addition to creating additional $cc_1 - cc_0$. Otherwise, $cc_1 - cc'$ of existing channels will be restarted with updated parallelism $p_1$ and the rest will be marked as passive. Since pipelining defines the number of outstanding commands, its value can easily be updated without the need to restart channels.

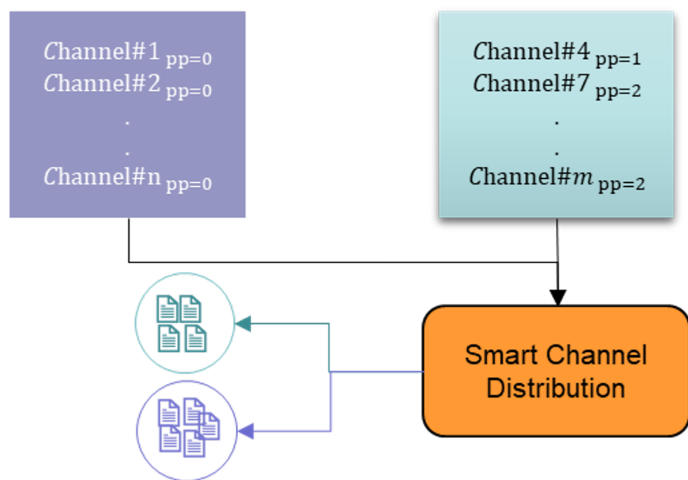



Figure 3.2: Workflow of Smart Channel Distribution

To ease the cost of updating parallelism, we developed *smart channel distribution* technique to assign existing channels to related clusters dynamically. With this, we keep channels with their parallelism configuration, and when a cluster requires that set of parallelism the algorithm assigns the channel primarily to that cluster as shown in Figure 3.2. Thus, the system avoids the cost of the restart channel effort and can keep the transfer rate stable.

### 3.4 Online Profiling

Re-adjusting transfer settings by incorporating new files and updating transfer parameter accordingly helps to adapt to changing dataset characteristics. However, it still depends on the heuristic method to determine new values for transfer parameters, which is bound to under-perform when end system settings deviate from the expectations. For example, it predicts that transferring two files concurrently (i.e., $concurrency = 2$) would suffice for large files to yield close-to-optimal performance even if maximum allowed channel count, $maxCC$ is higher. However, this assumption does not take the storage I/O performance into consideration, which has been shown to be a common source of bottlenecks in wide-area data transfers [39]. Consequently, despite yielding higher performance than the one-time tuning (i.e., "static") approach, the "dynamic" tuning approach is unavoidably constrained by the limitations of the heuristic method. We therefore introduce online profiling to explore parameter space in real-time to find settings that would lead to higher resource utilization. Since searching large search space in real-time is time consuming, *FStream* focuses mainly on the *concurrency* parameter since it does not only increase I/O throughput by reading multiple files concurrently but also opens up new network channels to mitigate network-related bottlenecks.

Algorithm 2 describes how online profiling explores search space for concurrency level when achieved throughput is below the required/desired level. The $runProfiling$ method is called periodically to ensure that current throughput, `currentThroughput`, satisfies user requirement, `desiredThroughput` at all times. Note that, since it's hard to achieve exact throughput value by means of application-layer adjustments, we accept throughput results that are at a close range of `desiredThroughput`. For the sake of simplicity, we defined the range as 15% and leave the detailed assessment of its impact on the stability and performance as a future work. If current throughput is significantly below or above the user defined range, then *FStream* starts to search for the concurrency value ($newCC$) to

**Algorithm 2** — Online Profiling discovers the concurrency level in real-time to satisfy throughput requirements.

```
Global: fileClusters - Dataset to transfer; maxCC - Maximum allowed concurrency level; bandwidth - Available network
 bandwidth; ρ - Percentage of bandwidth to utilize
 Input: avgThroughput - average transfer throughput of last interval
Output: Added or removed channels
 1: function OnlineProfiler(currentThroughput)
 2:     desiredThroughput = ρ ∗ bandwidth                                    ▷ Default value of ρ is 0.8
 3:     if currentThroughput is not x% range of desiredThroughput then       ▷ Default value of x is 15
 4:         if not checked historical data before then
 5:             newCC = profilingHistory.get(fileCluster, desiredThr)
 6:         end if
 7:         if newCC is not found in historicalData then
 8:             perChannelThroghput = avgThroughput/currentCC
 9:             requiredCC = desiredThroughput/perChannelThroghput
10:             if requiredCC > currentCC then                               ▷ Avoid drastic concurrency changes
11:                 newCC = Min(requiredCC, currentCC ∗ 2, maxCC)
12:             end if
13:         end if
14:         if newCC > currentCC then
15:             createChannels(newCC − currentCC)
16:         else
17:             disableChannels(currentCC − newCC)
18:         end if
19:         currentCC = newCC
20:         allocateChannels(fileCluster, currentCC)
21:     end if
22:     wait(interval)
23:     currentThroughput = calculateAverageThr()
24:     if currentThroughput > bandwidth then                                ▷ Update bandwidth
25:         bandwidth = currentThroughput
26:     end if
27:     profilingHistory.put(fileCluster, currentCC, currentThroughput, timestamp)
28:     OnlineProfiler(currentThroughput)
29: end function
```

satisfy the expectation. Note that user can specify the throughput expectation by choosing $\rho$ values accordingly. For example, if a streaming application necessitates more 6 Gbps sustained network throughput in a $10$ Gbps network, $\rho$ can be set to $0.7$ such that online profiling will explore the minimum concurrency value to keep the throughput between $6.3$ Gbps and $7.7$ Gbps.

Dynamic profiling is executed periodically at every $interval$ algorithm in every $10$ seconds during transfers. Also, we update the maximum available bandwidth value when instant throughput value hits a bigger throughput. Thus, the system can adapt to the possible higher maximum achievable bandwidth values. During performing this algorithm we limited maximum concurrency value due to avoiding overburden to the network.

### 3.5 Historical Analysis

When *FStream* determines that current throughput is beyond the acceptable range, then it first refers to past profiling reports, `profilingHistory`, to benefit from historical data. `profilingHistory` keeps track of two types of profiling, one for file size and the other for file type (i.e., small or large). The motivation is to first lookup past profiling results with file sizes similar to file size of current transfer (`fileCluster`). If it is not available, then it will search for reports for a dataset with similar file type (i.e., small or large) as `fileCluster` to take advantage of past profiling reports as much as possible. For example, if `profilingHistory` contains results for a dataset with 20MB average file size, it can be used to determine starting point for a dataset with 5MB average size if they both are classified as 'small' files to help online profiling converge faster.

In case `profilingHistory` does not include similar results or it has already been utilized it in a previous round, *FStream* will calculate per-channel throughput by dividing average throughput by current channel count (line 8). Per-channel throughput is then used to estimate the required channel count to fulfill throughput expectation. To avoid drastic changes in channel counts, we allow channel count to at most double in each profiling interval (line 11. Once new concurrency value, `newCC`, is determined either through using historical data or extrapolating `currentCC`, *FStream* either adds new channels or disables (i.e., mark as "passive") some of existing channels. $createChannels$ method will first utilize "passive" channels before creating new ones to speed up concurrency transitions. *FStream* also keeps track of parallelism value of "passive" channels such that they can be assigned to file clusters whose parallelism matches with channel's parallelism. For example, when *FStream* decides to activate a channel with parallelism value $4$, it will attempt to assign it to "large" file cluster in the current dataset to take full benefit from this channel. By doing so, *FStream* avoids the cost of channel restart and alleviate throughput fluctuations. Finally, *FStream* adds recent profiling results to `profilingHistory`

**Algorithm 3** — Quality of Service Implementation for Transfer Throughput–

```
    Global: fileClusters - Dataset to transfer;
 1: function LIMITEDSPEED(currentThroughput, desiredThroughput)
 2:     diff = Abs(currentThroughput − desiredThroughput)
 3:     if currentThroughput < desiredThroughput then
 4:         if diff > desiredThroughput * 0.5 then
 5:             newChannelCount = currentChannelCount * 2
 6:         end if
 7:         if (diff < desiredThroughput * 0.5) && (diff > desiredThroughput * 0.2) then
 8:             newChannelCount = currentChannelCount + 2
 9:         end if
10:         if (diff < desiredThroughput * 0.2) && (diff > desiredThroughput * 0.1) then
11:             newChannelCount = currentChannelCount + 1
12:         end if
13:         if (diff < desiredThroughput * 0.1 && diff > desiredThroughput * 0.05) then
14:             if fileCluster == SMALL then
15:                 incrementPipelining(fileCluster)
16:             end if
17:             if fileCluster == LARGE then
18:                 incrementParallelism(fileCluster, 1)
19:             end if
20:         end if
21:     else
22:         Decrease concurrency, parallelism, and concurrency in a similar way as above
23:     end if
24:     updateClustersChannels(fileCluster)
25:     return (fileCluster)
26: end function
```

along with timestamp to be able to benefit from them in the future. The timestamp lets the *FStream* to prioritize recent reports over older ones to capture seasonal trends.

## 3.6 Quality of Service Support

Although *dynamic tuning online profiling*, and *historical analysis* increases transfer performance significantly, they do not offer performance guarantee. Therefore, we extend the *FStream* to support QoS for transfer throughput by configuring transfer settings in a way that transfer throughput is kept within a small range of desired speed. While concurrency is the most effective parameter to increase and decrease the transfer throughput, we need to change other parameters (i.e. parallelism, pipelining) to control throughput in high granularity.

We used the heuristic algorithm to calculate and start the initial transfer for this approach. Algorithm 3 describes how QoS implementation works. It monitor the transfer throughput and updates the concurrency value gradually to bring the difference between

current throughput and desired throughput smaller than a threshold (by default 5%). Since achieving the exact desired throughput value is nearly impossible by only tuning application layer parameters, we tried to stabilize the transfer rate withing a reasonable range of desired throughput. Therefore, the $limitSpeed$ method is called periodically to ensure that the difference between current throughput, $currentThroughput$, and the required throughput, $desiredThroughput$, is smaller than a threshold. The algorithm calculates the difference between the current throughput value and the desired throughput value (line $2$). Then, if the gap is bigger than the $50\%$ of the desired value then the algorithm doubles/halves the channel count. We used this approach to close the large gap between the desired value and the current throughput value quickly. We chose $50\%$ as our first condition because, if the real-time throughput value is less than half of the desired value, then we can converge faster by just doubling/halving the concurrency value. If the difference is between $50\%$ and $20\%$ of the desired value we increased/decreased the channel count by $2$, and if the difference is between $20\%$ and $10\%$ of the desired value we increased/decreased channel count by $1$. We also tried these two conditions with different threshold values, however, our observation showed that these ranges converged yield the best performance in terms of accuracy and convergence time. Lastly, If the difference is between $10\%$ and $5\%$, we changed the pipelining and parallelism values according to their file cluster sizes for precise tuning.

# CHAPTER 4
# EVALUATION AND RESULTS

In this section, we provide details of datasets that has been used in our performance tests and evaluation of test results.

## 4.1 Dataset Characteristics

We utilized both synthetic and real-world workloads to evaluate the performance of *FStream*. To emulate the streaming nature of distributed workflows, we inject a new batch of files to transfer directory at $20 - second$ intervals. If the transfer of a dataset from the previous interval is not completed, then we merge the remaining dataset with new files. Table 4.1 shows the file size of new data at different intervals for synthetic workloads.

While the first three workloads are designed to study the impact of changing dataset characteristics at different intervals. Firstly, with workload $\#1$, we examine the impact of the case of if the transfer starts with *small* files and continues with *large* files and arrive 2 more intervals like this. In this case, we observed the heuristic algorithm tends to assign a greater number of concurrency value to transfer. With workload $\#2$, we examine the impact of the if transfer starts with *large* file and continue with *small* files, and arrive 2 more intervals

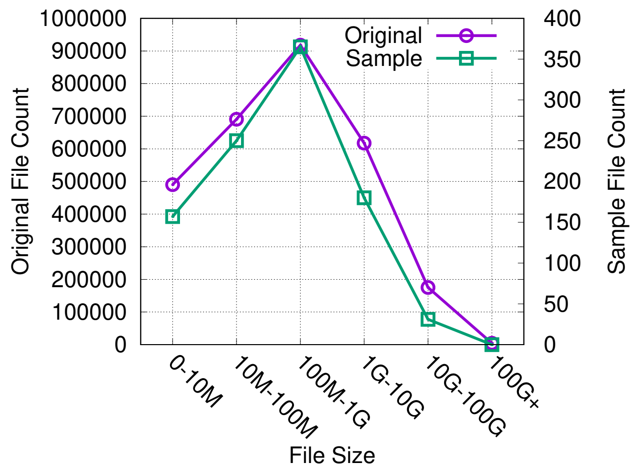



Figure 4.1: File size distribution of SRA dataset.

| Workload | Interval | | | |
|---|---|---|---|---|
| | 1 | 2 | 3 | 4 |
| 1 | Small | Large | Small | Large |
| 2 | Large | Small | Large | Small |
| 3 | Extra Small | Extra Large | Large | Large |
| 4 | Small+Large | Small+Large | Small+Large | Small+Large |

Table 4.1: File types and arrival orders for synthetic datasets. File sizes range between $0 - 10$ MB (Extra Small), $10$-$100$ MB (Small), $100$ MB-$2$ GB (Large) and $> 2$ GB (Extra Large).

like this. In this test case, we observed the transfer tends to start with a lower number of concurrency value. With workload #3, we observe the effect of starting with *extra small* files and continue with *extra large* files. Test results showed that starting with a greater number of concurrency values leads to the best performance of the static approach. Also, we observed the impact of mixed dataset arriving during intervals with the last synthetic workload. Lastly, with workload #4 we examine to assess the performance when a new batch of files always include both small and large file types.

We also generated dataset to emulate streaming bioinformatics and free-electron laser physics workflows. The bioinformatics workflow (henceforth SRA workflow) streams genome sequence data from Sequence Read Archive (SRA) and runs a series of transformations to extract process-ready SAM/BAM files [40].

The massive size of SRA repository, when combined with limited computing resources at campus clusters, requires researchers to rely on streaming workflows to download and process only a small number of files at any given time. We scanned $2.8M$ genome sequence for “homo sapiens” and took a subset of it to use in the experiments whose data distribution is illustrated in Figure 4.1. Although we significantly reduced the number of files due to time constraints, the sampled subset follows similar file size distributions as the original. We also created a dataset to represent data flow in the Linac Coherent Light Source (LCLS) experiment as described in [41]. LCLS takes X-ray snapshots of atoms and generates terabytes of data per experiment, which has to be moved to supercomputers located in

| | SRA | | LCLS | |
|---|---|---|---|---|
| Interval | Avg. Size | File Count | Avg. Size | File Count |
| 1 | 10.79 MB,<br>642.5 MB | 135,<br>265 | 40.92 MB,<br>1.19 GB | 62,<br>98 |
| 2 | 47.32 MB,<br>394.34 MB | 153,<br>247 | 53.02 MB,<br>1.81 GB | 80,<br>80 |
| 3 | 17.54 MB,<br>833.35 MB | 169,<br>231 | 23.17 MB,<br>2.1 GB | 48,<br>112 |
| 4 | 917.34 MB | 49 | 26.76 MB,<br>1.22 GB | 51,<br>69 |

Table 4.2: File characteristics and arrival orders of generated SRA and LCLS datasets. File sizes range between 0-10 MB (Extra Small), 10-100 MB (Small), 100 MB-2 GB (Large) and $>$ 2 GB (Extra Large).

Berkeley, CA (Cori), Chicago, IL (Mira), and Oak Ridge, TN (Titan) expeditiously to carry out real-time analyses. LCLS dataset is dominated by large files where average file size of 13.1 GB and median file size of 4 GB. Similar to synthetics workloads, we split the SRA and LCLS dataset into four groups randomly and scheduled them in four intervals. Table 4.2 shows the file and transfer characteristics according to intervals of generated SRA and LCLS files. File sizes in the SRA dataset generated the range of 1 Mb to 9 Gb and dataset size is a total of 511 Gb. On the other hand, files in the LCLS dataset generated the range of 1 Mb to 13 Gb and dataset size is a total of 311 Gb. We divided both datasets into 4 different intervals with a fixed number of files for every interval.

## 4.2 Experimental Results

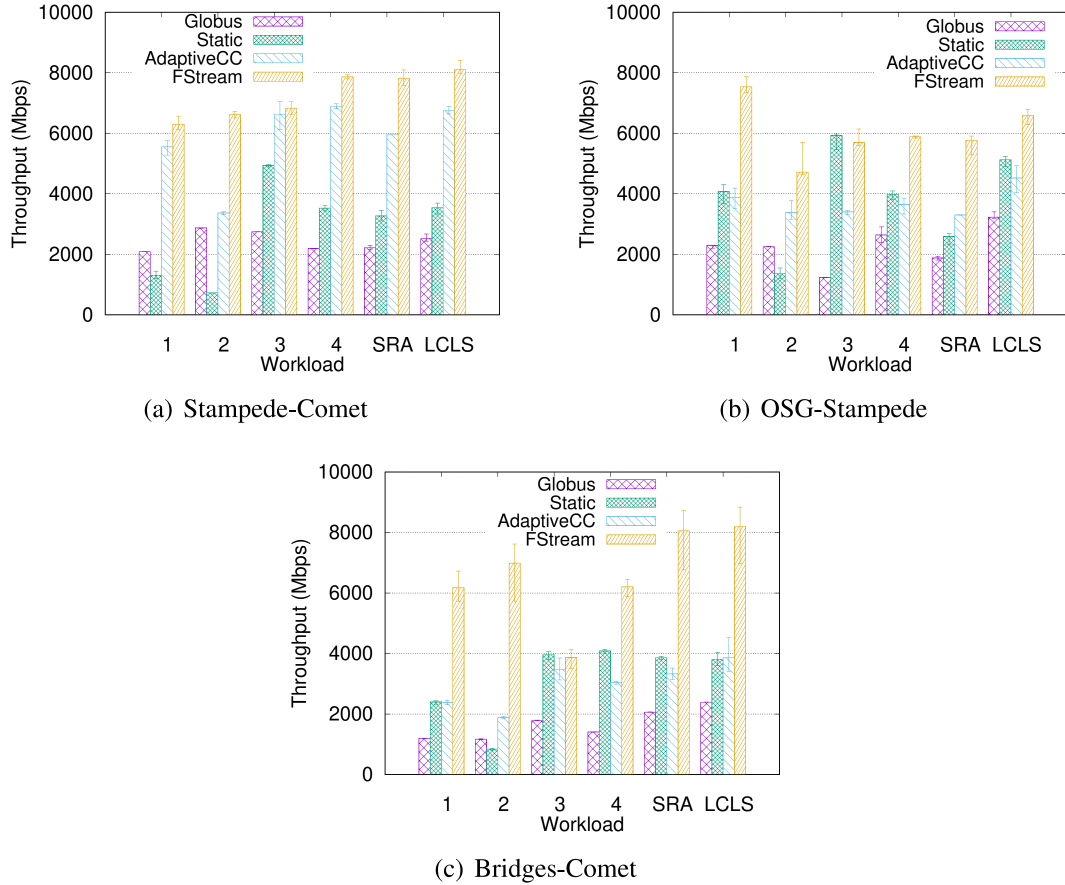
We used three pairs of XSEDE [42] sites, Stampede-Comet, OSG-Stampede, and Bridges-Comet to run transfers, which are connected with a high-speed network with up to 40 Gbps network capacity. The round trip time is 38ms, 32ms, and 58ms in Stampede-Comet OSG-Stampede, and Bridges-Comet connections, respectively. We compared *FStream* against Globus [13], the Static (i.e. heuristic approach [18]), and AdaptiveCC [15]. Although all of the transfer algorithms use GridFTP as a transfer protocol, they configure its settings

(i.e. pipelining, parallelism, and concurrency) differently[1]. Globus configures the transfer settings on the user's behalf using a fixed set of predetermined configurations. The Static uses Algorithm 3 to estimate the parameter values at the beginning of the transfer and use them throughout the transfer. AdaptiveCC, similar to the Static, uses a heuristic to determine initial settings, but then adjust the value of concurrency in real-time based on a utility function that rewards high throughput and low channel count.

We set the time between every interval conducted as $20$ seconds for every test. Also, we set the bandwidth limit to $10$ Gbps for synthetic dataset transfers initially because of the OSG network infrastructure limited by $10$ Gbps network card. Additionally, we limited to maximum concurrency value by $10$ and we decided to use $\%80$ of maximum possible bandwidth to avoid overwhelming the network traffic. However, we explored the performance of *FStream* beyond these settings in upcoming sections. Finally, we tested every workload five times and each workload tested sequentially by all four algorithms to diminish changing network background effect on tests.

The average throughput results for all workload types is given in Figure 4.2. Except Workload $\#3$, *FStream* outperforms the Static approach by $2.3x$-$9.1x$ in average transfer throughput in Stampede-Comet transfers. The largest gain happens when large files in the first interval are followed by small files in the second interval (Workload $\#2$). This is because the Static method predicts transfer settings based on initial dataset configuration (i.e., for large files), resulting in extremely low throughput (less than $800$ Mbps in Stampede-Comet network) when the dataset is dominated by small files in the next interval. *FStream* yields $3.4x$ and $8.3x$ higher than the Static in OSG-Stampede and Bridges-Comet transfers respectively for Workload $\#2$. On the other hand, when the transfer starts with the dataset that contains very small files (Workload $\#3$), the Static algorithm estimates large concurrency value and yields slightly higher throughput ($4\%$) than *FStream*, which

[1]Note that even though we utilized GridFTP, the settings that we aim to configure are either readily available or can easily be implemented in other protocols




(a) Stampede-Comet

(b) OSG-Stampede

(c) Bridges-Comet

Figure 4.2: Transfer performance comparison of algorithms in Stampede-Comet (a), OSG-Stampede (b), and Bridges-Comet (c) networks for six workload types.

starts with similar parameter configuration as the Static but later switches to new parameter configuration when the dataset is dominated by large files.

By adjusting the concurrency value in real-time, AdaptiveCC outperforms the Static approach in Stampede-Comet transfers by up to $4.5$ times, however, achieve similar or slightly lower performance in OSG-Stampede and Bridges-Comet transfers. Compared to *FStream* , AdaptiveCC yields up to $1.96x$, $1.94x$, and $3.6x$ lower throughput in Stampede-Comet, OSG-Stampede, and Bridges-Comet transfers. Despite using real-time concurrency tuning similar to *FStream*'s online profiling, it differs in several ways. First, it tunes the concurrency value once at the beginning of the transfer, thus fails to adapt changing transfer conditions. This limitation, however, cannot be simply mitigated by re-executing its profil-

ing method periodically since simply changing increasing/decreasing channel optimizing their parallelism values will be insufficient to take full advantage of available channels. Second, it oblivious to the user/application performance requirements and solely aims to increase concurrency until throughput increase falls below certain percentage. This approach is however likely to lead to unnecessarily high transfer throughput in some cases and unreasonably low performance in other cases as distributed workflow may have stringent transfer requirements to operate. Third and final, it does not keep track of historical data, so it needs to restart its search phase every time dataset settings conditions change. *FStream* solves this issue by not only remembering optimal settings for different file sizes but also different file types.

Performance of *FStream* is $3.4x$, $2.3x$, and $1.3x$ more for SRA and $3.2x$, $2.2x$, and $1.2x$ more for LCLS dataset compared to Globus, Static and AdaptiveCC, respectively, in Stampede-Comet experiments. The performance gain becomes $3x$, $2.2x$, and $1.74x$ for SRA and $2x$ and $1.28x$, and $1.45x$ for LCLS transfers in OSG-Stampede network. Finally, it outperforms all algorithms by $2.1 - 3.9x$ for SRA and LCLS transfers in Bridges-Comet network. As a result, *FStream* yields $3.3x$, $2.84x$ and $1.75x$ higher throughput than Globus, Static, and AdaptiveCC in all networks for all workload types. In the rest of the experiments we present results only for the Stampede-Comet network due to space limitations.

Figure 4.3 illustrates instantaneous throughput (4.3(a) 4.3(c), and 4.3(e)) and corresponding channel counts (4.3(b) 4.3(d), and 4.3(f)) for the Static, AdaptiveCC and *FStream* in one of the representative transfers for Workload $1-3$. Since *FStream* is aimed at achieving $80\%$ of available bandwidth (i.e.,$10$ Gbps), it increases its concurrency value after few profiling steps to meet the demand. Consequently, its instantaneous throughput reaches over $8$ Gbps for Workload $\#1$ and $\#2$. Even though AdaptiveCC also identifies concurrency value $10$ to be optimal, it falls short to adjust the parallelism of channels when workload changes from small file dominated to large files dominated, hence obtains nearly $50\%$ less

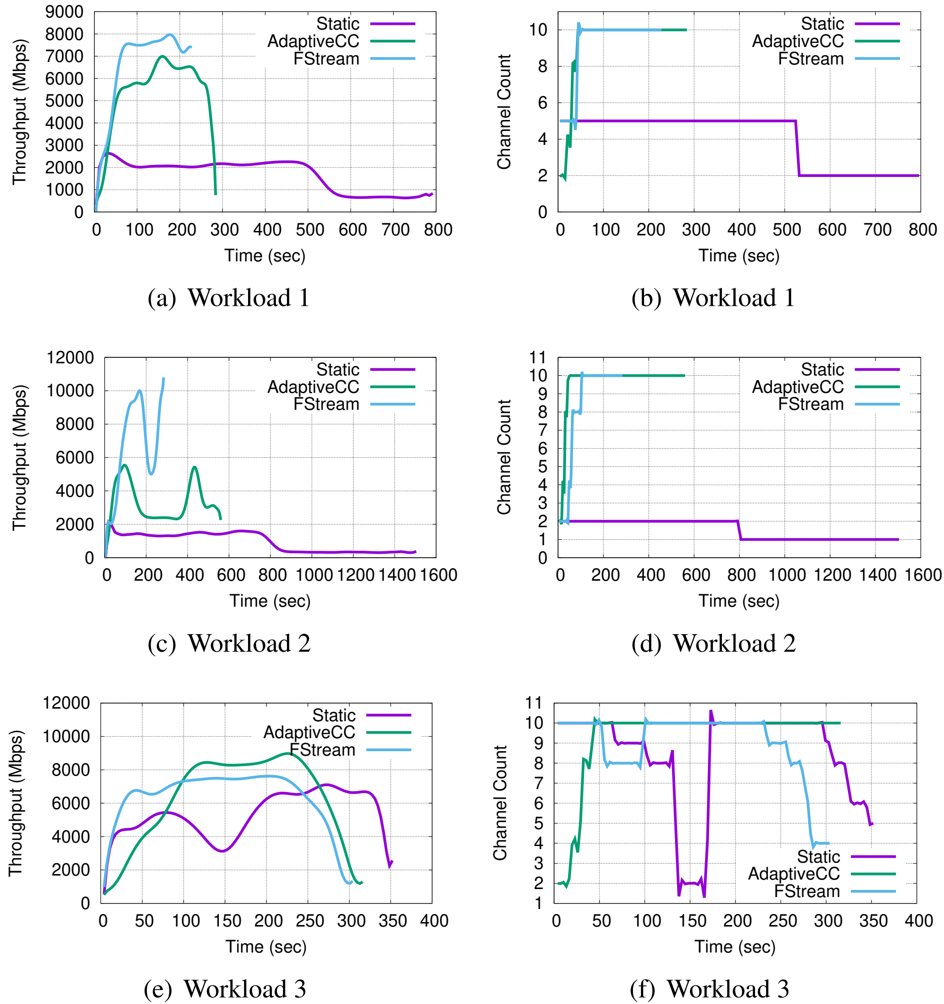
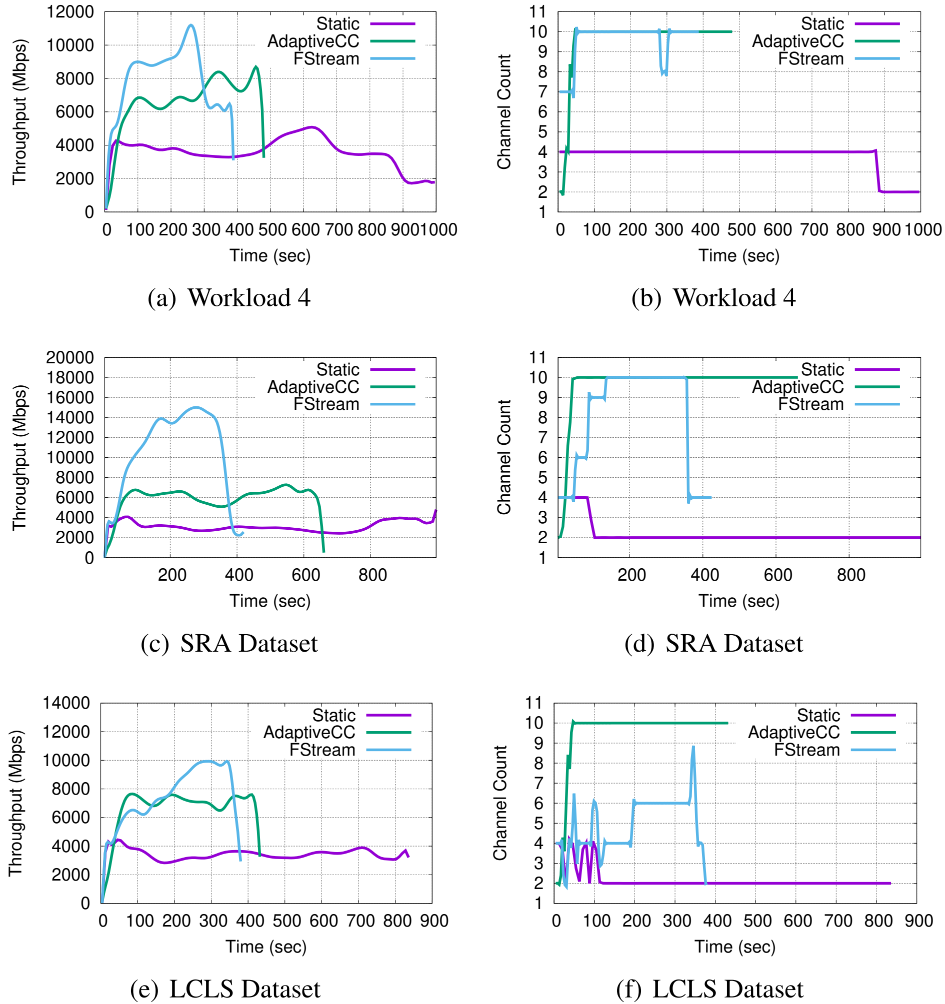



Figure 4.3: Instantaneous throughput and concurrency changes of static and *FStream* for Workload $1-3$ in Stampede-Comet transfers.

throughput. On the other hand, *FStream* struggles to reach the requirement for Workload #3 until $100s$ since even maximum permitted concurrency value (i.e., $10$) is insufficient to yield more than $7$ Gbps when the dataset is dominated by very small files whose average size is less than $10$ MB. Yet, it performs slightly better than the AdaptiveCC with the help of quick convergence of its online profiling method. On the other hand, the Static approach performs well only for Workload #3 in which the dataset in the first interval consists of very small files, letting the heuristic algorithm to predict the concurrency value as $10$. Since the Static keeps using the same settings for the rest of the transfer, it achieves similar




(a) Workload 4 (b) Workload 4

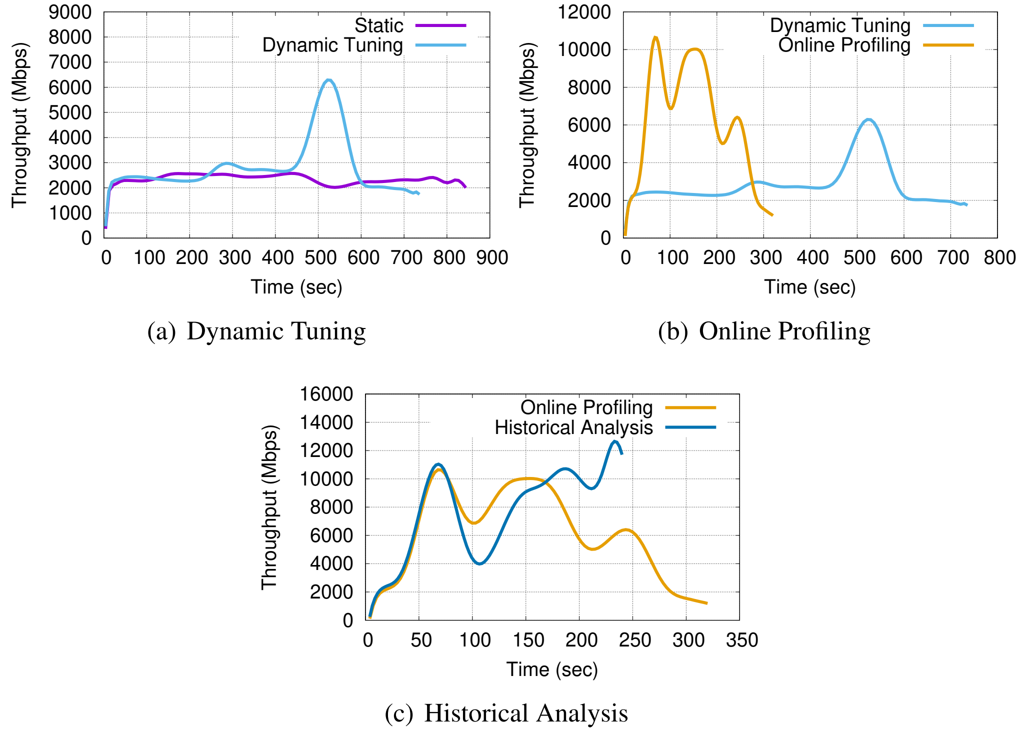
(c) SRA Dataset (d) SRA Dataset

(e) LCLS Dataset (f) LCLS Dataset

Figure 4.4: Instantaneous throughput results for Workload $\#4$ and real-world workloads in Stampede-Comet transfers.

throughput as *FStream*. Unlike heuristic solution, *FStream* lowers its concurrency once the dataset is dominated by large files (at around $120s$ in Figure 4.3(f)) to avoid overloading end systems and network for marginal performance gain.

We also analyzed the algorithms' performance with real-world dataset SRA and LCLS characteristics. Figure 4.4 presents instantaneous throughput and concurrency values when streaming workload contains both small and large files at all times. The presence of large files for the entire transfer duration helps *FStream* to reach over $11$ Gbps for workload $\#4$, $15$ Gbps for SRA workflow, and $10$ Gbps for LCLS workflow as shown in Fig-




(a) Dynamic Tuning

(b) Online Profiling

(c) Historical Analysis

Figure 4.5: Breakdown of *FStream*'s performance into its main components; dynamic tuning (a), online profiling (b), and historical analysis(c)

ure 4.4(a), 4.4(c), and 4.4(e). While bandwidth is entered as $10$ Gbps in the configuration file, *FStream* is able to notice that it can yield higher throughput and updates *bandwidth* value accordingly to override misconfiguration as explained in line 24 in Algorithm 2. Correcting misconfigured bandwidth value lets *FStream* to adapt the changing network conditions such as upgraded capacity. SRA dataset transfer of Stampede2 to Comet showed that the algorithm reached beyond $10$ Gbps that provided maximum bandwidth limit. However, dynamic adaptive and static algorithms' performance was similar for both datasets and sites because the distribution of the datasets was mixed size chunks for every interval.

The Static approach also obtains better results these experiments, but its throughput does not reach beyond $4$ Gbps, leading to $2.1x$, $3.2x$, and $1.8x$ higher transfer times for Workload $\#4$, SRA, and LCLS transfers, respectively.

### 4.3 *FStream* Performance Analysis

Compared to the Static approach, *FStream* adopts three main techniques to enhance the transfer performance of streaming workflows as dynamic tuning, online profiling, and historical analysis. In order to quantify the performance gain by each of these components, we breakdown its performance for a workflow that generates small, large, extra small, and large files in consecutive intervals in Figure 4.5. We arranged data arrival rate times (i.e. intervals) in each experiment in a way that a new dataset is added after the transfer algorithm finishes the prior dataset to isolate dataset of intervals, which led instantaneous throughput to drop between intervals. We first compared the Static approach against dynamic tuning of the component of *FStream* in Figure 4.5(a). The result shows that even though they start with similar transfer settings and obtain close throughput performance until the 480s, dynamic tuning estimates a new set of parameters in the third interval when extra small files are added. This change lets the dynamic tuning method to finish $13.6\%$ seconds faster than the Static approach even though they use the same settings for the first, third, and fourth intervals. Note that the performance gain of dynamic tuning would be much more apparent for Workload $\#2$ where the workload starts with large files but switches to small files later since it will cause the static to use small concurrency and pipelining values for small files in subsequent intervals.

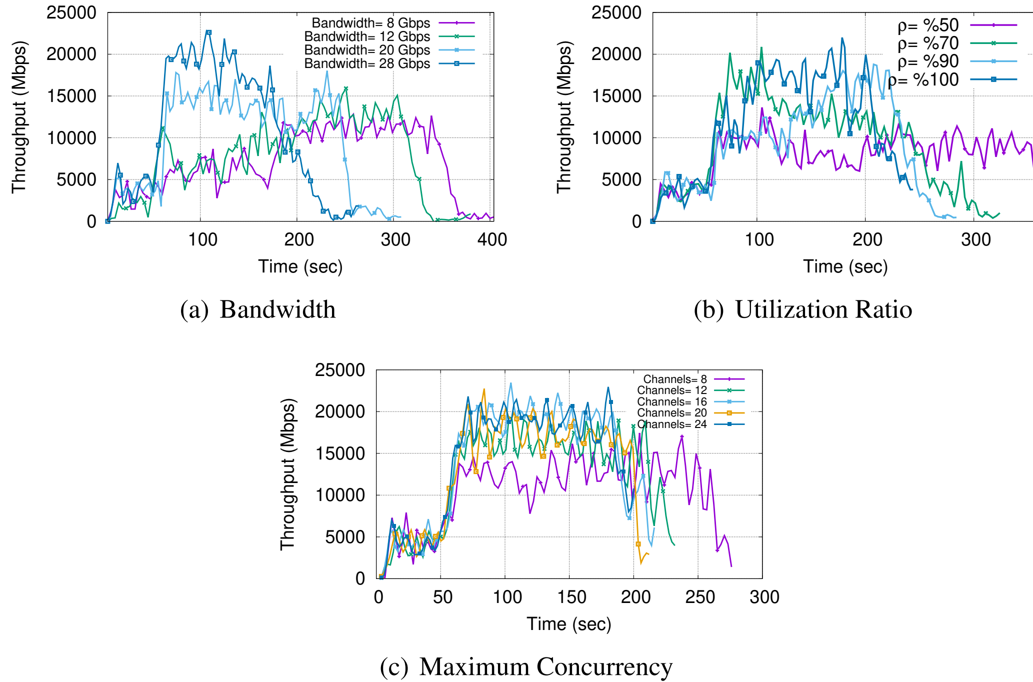
Coupling dynamic tuning with online profiling helps to transcend the limits of heuristic by adjusting the concurrency level in real-time, which cuts the transfer time by around $55\%$ over dynamic tuning as can be seen in Figure 4.5(b). Finally, augmenting online profiling with historical analysis further reduces transfer time by $25\%$ compared to “just” online profiling. Historical analysis and online profiling perform similar until $150s$, at which point the third dataset (extra small) is added. Even though file size of extra small and small datasets differ, *FStream* can still benefit from the profiling results of the first interval as historical analysis will return reports from similar file types (i.e., small or large) if file size

based look-up does not work. This mitigates the need to rerun online profiling again and increases throughput faster. Similarly, large files in the fourth interval are transferred using the profiling results of the large files in the second interval, assisting *FStream* to reach $15$ Gbps throughput in a matter of seconds in the fourth interval. As a result, dynamic tuning, online profiling, and historical analysis help *FStream* to cut the transfer time by $71\%$ compared to the Static approach.

### 4.4 Tunable Parameters of *FStream*

The results presented in earlier sections were gathered when *FStream* is configured with following options: *Bandwidth* is $10$ Gbps, utilization ratio ($\rho$) is $0.8$, and maximum concurrency ($maxCC$) is $10$. The default setting will expect *FStream* to obtain transfer throughput between $7.2$ and $8.8$ Gbps ($bandwidthx\rho$) using maximum of $10$ channels. While this yields more than $6$ Gbps average throughput in most cases with up to $14$ Gbps throughput for SRA transfers, one can tweak these settings to control the aggressiveness of *FStream* to meet unique workflow requirements. Note that these settings are different than transfer parameters in a way that they are more intuitive to configure and does not need to be updated once configured even though dataset characteristics change. For instance, if a workflow requires more than $10$ Gbps throughput at all times in a network with $20$ Gbps bandwidth regardless of dataset characteristics, one can enter $bandwidth = 20$ and $\rho = 0.5$ and $maxCC = 100$ such that *FStream* will search for the concurrency level to satisfy the requirement via online profiling.

Figure 4.6(a) shows the impact of setting *bandwidth* to values between $8$ Gbps and $28$ Gbps when $\rho = 0.8$ and $maxCC = 20$. In all cases, *FStream* starts with the settings predicted by heuristic algorithm, thus yields $3-5$ Gbps throughput. After $40-50$ seconds, it realizes that current throughput is lower than *desired* level, especially for high bandwidth cases. It then estimates new concurrency value to reach desired performance. This results




(a) Bandwidth

(b) Utilization Ratio

(c) Maximum Concurrency

Figure 4.6: Instantaneous throughput and concurrency changes of static and *FStream* for synthetic and real-world workloads in Stampede-Comet transfers.

in high concurrency estimation when *bandwidth* is set to $28$ Gbps, which leads to over $20$ Gbps throughput. On the other hand, when *bandwidth* is set to $8$ Gbps, throughput increases gradually as *desired* is initially calculated to be between $5.4$ Gbps ($= 8 * 0.8 - 8 * 0.8 * 0.15$) and $7.3$ Gbps ($= 8 * 0.8 + 8 * 0.8 * 0.15$) which is easily satisfied at around $60s$. However, as *FStream* is configured to adjust bandwidth to high observed values, it incrementally increases *bandwidth* and *predicted* based on observed throughput results. As a result, throughput slowly increases to more than $10$ Gbps.

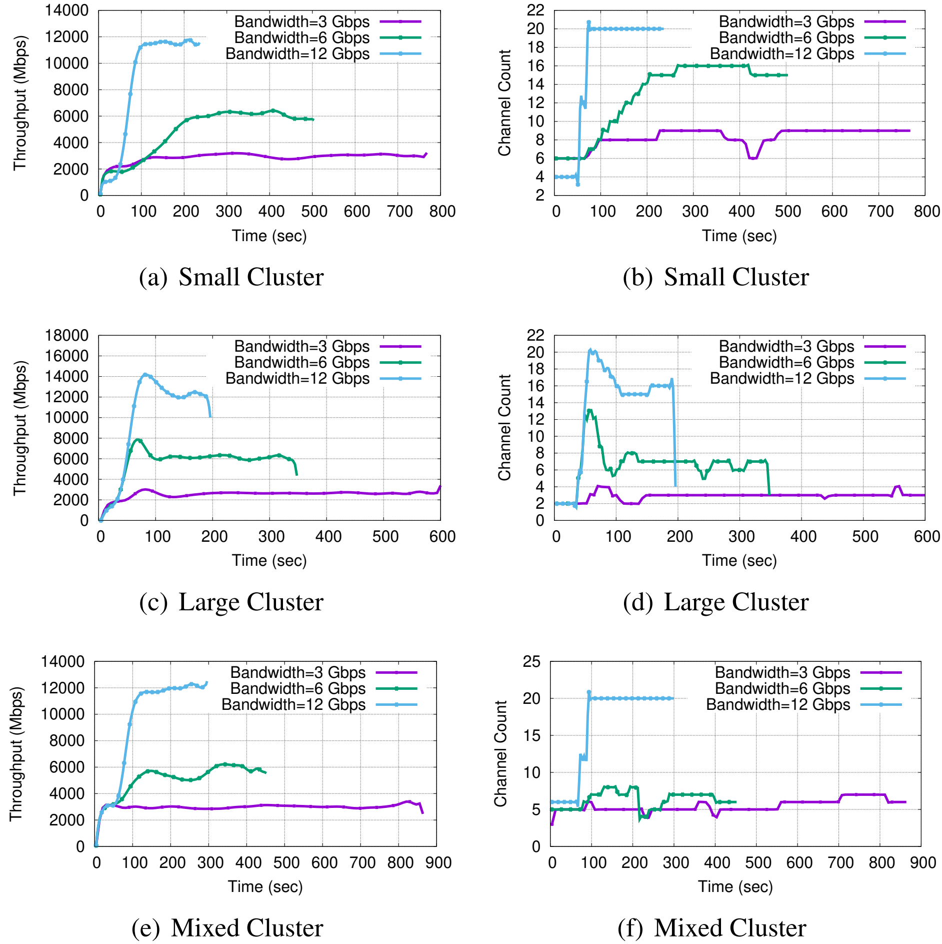
In addition, increasing bandwidth threshold ($\rho$) can also be used to make *FStream* more aggressive to obtain high throughput. Figure 4.6(b) shows that by setting $\rho$ to $1$, transfer time can be reduced from $350s$ to $240s$ compared to $\rho$ value of $0.5$. Maximum concurrency will also help to increase throughput, which is especially important when small files dominate the dataset. Note that, its benefit is limited in Figure 4.6(c) as SRA dataset contains both small and large files, thus even $maxCC = 8$ would yield reasonably well (i.e.g, more

than 15Gbps) performance.

## 4.5 Quality of Service with *FStream*

The results that we provided so far are mainly focused on achieving maximum possible performance. However, this can unnecessarily overwhelm the network and end system resources. Therefore, we tested the *Quality of Service* algorithm with three different datasets (small, large, and mixed) to maintain transfer performance at a desired rate. *FStream* settings are configures as follows: $maxCC$ is $20$, utilization ratio ($\rho$) is $1.0$, *bandwidth* is $20$ Gbps. We tested each dataset with the $desiredThroughput$ value of $3$ Gbps, $6$ Gbps, and $12$ Gbps.In all cases, *FStream* used the heuristic algorithm to decide initial transfer values. We performed the tests between Stampede2-Comet.

Figure 4.7 illustrates instantaneous throughput (4.7(a) 4.7(c), and 4.7(e)) and corresponding channel counts (4.7(b) 4.7(d), and 4.7(f)) for small, large, and mixed dataset transfers. Figure 4.7(a) shows transfer performance for small files with different desired throughput values. As can be seen in the figure, we can achieve the desired *QoS* value by only configuring concurrency in $40-50$ seconds for $3$ Gbps and $12$ Gps. On the other hand, it takes around $200$ seconds for $6$ Gbps desired throughput. Figure 4.7(b) shows that the convergence time of the algorithm is highly dependent on the difference between the initial throughput value and the desired value. Since the algorithm duplicates channel count when the gap is higher(more than $50\%$), it took a shorter time to converge. We also observed that the error rate is around $4\%$ for $3$ Gbps, $5\%$ for $6$ Gbps, and $11\%$ for $12$ Gbps QoS experiments. The higher error rate for the higher throughput value is related to *maxCC* value because it reached to maximum channel count quickly as it is shown in Figure 4.7(b). Among all tests, the *QoS* algorithm performed the best results for small files. One of the reasons is that when achieved throughput gets closer to the desired value the algorithm tunes the pipelining accordingly without a significant cost.




(a) Small Cluster

(b) Small Cluster

(c) Large Cluster

(d) Large Cluster

(e) Mixed Cluster

(f) Mixed Cluster

Figure 4.7: Instantaneous throughput and concurrency changes of QoS transfers for Small, Large and Mixed (Small+Large) clusters.

The results for large files are shown in Figure 4.7(c). Although we observed more fluctuations in throughput, *FStream* is still able to achieve throughput close to desired level. Specifically, the error rate is $18\%$ for $3$ Gbps, $7\%$ for $6$ Gbps, and $10\%$ for $12$ Gbps. We notice that it is necessary to run the parallelism in addition to concurrency for large file transfers to meet desired performance. However, changing parallelism value of existing connections is a costly operation due to the requiring to restart connections. Finally, we conducted QoS transfers with mixed dataset. Results showed that error rate is $6\%$ for $3$ Gbps, $19\%$ for $6$ Gbps, and $9\%$ for $12$ Gbps. As a result, *QoS* algorithm is able to keep the difference between desired throughput and actual throughput below $20\%$ for all datasets.

# CHAPTER 5
# CONCLUSION AND FUTURE WORK

Streaming transfers have different characteristics than batch transfers such as strict performance requirements and varying dataset settings. Thus, while one-time transfer tuning may be sufficient for batch transfers but it falls short to meet stringent performance requirement of streaming transfers. In this thesis, we introduced *FStream* to enhance network performance of streaming applications through real-time performance adjustments. *FStream* monitors the throughput of streaming transfer periodically and re-configure tunable transfer parameters when transfer conditions deviate from initial observations such that high transfer performance can be sustained. Specifically, *FStream* employs *Online Profiling* to find the optimal transfer settings for an observed system conditions. It further integrates historical analysis into transfer tuning methodology to exploit long-running nature of streaming transfers by leveraging from past online profiling executions in finding to minimize the overhead of online profiling. Moreover, *FStream*offers quality of service (QoS) support for streaming applications to ensure reliable transfer performance such that time-sensitive streaming applications can be executed in shared networks.

The results gathered in multiple networks for synthetic and real-world workloads reveal that *FStream* outperforms the batch transfer applications by up to $9x$ when the file size of the streaming dataset evolves over time. Its performance is also more than $2x$ higher when tested with real-world application scenarios such as Sequence Read Archive and Linac Coherent Light Source workflows. We also find that, *FStream*'s *QoS* algorithm is able to sustain desired transfer performance with as low as $4\%$ error rate for small files, $7\%$ error rate for large files, paving the way for performance critical streaming applications to execute in shared networks.

As future work, further transfer settings (in combination with existing ones) can be explored to enable fine-granular transfer performance control. These parameters include but not limited to file system type (i.e., parallel vs single-disk), storage I/O performance, transport-layer protocol (e.g., TCP Cubic, BBR). Moreover, although we artificially injected some background traffic to test the performance of *FStream*'s QoS support, we intend to assess it under heavy background traffic conditions as well to gain deeper insight into potential issues and pitfalls. Finally, we will integrate *FStream* to workflow management systems to enable its wide adoption by the science community.

**REFERENCES**

[1] D. Sigg, “The advanced ligo detectors in the era of first discoveries,” in *Interferometry XVIII*, International Society for Optics and Photonics, vol. 9960, 2016, p. 996 009.

[2] M. Wolf, G. Eisenhauer, and P. Widener, “Rethinking streaming system construction for next-generation collaborative science.,” Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2016.

[3] J. Loveday, “The sloan digital sky survey,” *Contemporary Physics*, vol. 43, no. 6, pp. 437–449, 2002.

[4] *Dark Energy Survey*, https://www.darkenergysurvey.org/.

[5] *A Toroidal LHC ApparatuS Project (ATLAS)*, http://atlas.web.cern.ch/.

[6] D. Evans, “The internet of things: How the next evolution of the internet is changing everything,” *CISCO white paper*, vol. 1, no. 2011, pp. 1–11, 2011.

[7] A. Bifet and E. Frank, “Sentiment knowledge discovery in twitter streaming data,” in *International conference on discovery science*, Springer, 2010, pp. 1–15.

[8] *Large Synoptic Survey Telescope*, https://www.lsst.org/, 2020.

[9] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, “Millwheel: Fault-tolerant stream processing at internet scale,” *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[10] *Kinesis*, http://aws.amazon.com/kinesis/, 2017.

[11] *Storm*, http://storm.apache.org/, 2017.

[12] G. Fox, S. Jha, and L. Ramakrishnan, *STREAM2016: Streaming Requirements, Experience, Applications and Middleware Workshop*. 2016.

[13] B. Allen, J. Bresnahan, L. Childers, I. Foster, G. Kandaswamy, R. Kettimuthu, J. Kordas, M. Link, S. Martin, K. Pickett, and S. Tuecke, “Software as a service for data scientists,” *Communications of the ACM*, vol. 55:2, pp. 81–88, 2012.

[14] E. Arslan and T. Kosar, “High speed transfer optimization based on historical analysis and real-time tuning,” *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[15] E. Arslan, B. A. Pehlivan, and T. Kosar, “Big data transfer optimization through adaptive parameter tuning,” *Journal of Parallel and Distributed Computing*, vol. 120, pp. 89–100, 2018.

[16] E. Arslan, K. Guner, and T. Kosar, “Harp: Predictive transfer optimization based on historical analysis and real-time probing,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’16, Salt Lake City, Utah: IEEE Press, 2016, 25:1–25:12, ISBN: 978-1-4673-8815-3.

[17] E. Yildirim, E. Arslan, J. Kim, and T. Kosar, “Application-level optimization of big data transfers through pipelining, parallelism and concurrency,” *IEEE Transactions on Cloud Computing*, vol. 4, no. 1, pp. 63–75, 2015.

[18] E. Arslan, B. Ross, and T. Kosar, “Dynamic protocol tuning algorithms for high performance data transfers,” in *European Conference on Parallel Processing*, Springer, 2013, pp. 725–736.

[19] S. Floyd *et al.*, “Highspeed tcp for large congestion windows,” 2003.

[20] C. Jin, D. X. Wei, and S. H. Low, “Fast tcp: Motivation, architecture, algorithms, performance,” in *IEEE INFOCOM 2004*, IEEE, vol. 4, 2004, pp. 2490–2501.

[21] K. Tan, J. Song, Q. Zhang, and M. Sridharan, “A compound tcp approach for high-speed and long distance networks,” in *Proceedings IEEE 25TH IEEE International Conference on Computer Communications (INFOCOM)*, IEEE, 2006, pp. 1–12.

[22] T. J. Hacker, B. D. Noble, and B. D. Atley, “Adaptive data block scheduling for parallel streams,” in *Proceedings of HPDC ’05*, ACM/IEEE, Jul. 2005, pp. 265–275.

[23] E. Yildirim, D. Yin, and T. Kosar, “Prediction of optimal parallelism level in wide area data transfers,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 12, pp. 2033–2045, 2011.

[24] N. Freed, *SMTP service extension for command pipelining*, http://tools.ietf.org/html/rfc2920.

[25] J. Semke, J. Mahdavi, and M. Mathis, "Automatic tcp buffer tuning," in *Proceedings of the ACM SIGCOMM'98 conference on Applications, technologies, architectures, and protocols for computer communication*, 1998, pp. 315–323.

[26] N. S. Rao, Q. Liu, S. Sen, G. Hinkel, N. Imam, I. Foster, R. Kettimuthu, B. W. Settlemyer, C. Q. Wu, and D. Yun, "Experimental analysis of file transfer rates over wide-area dedicated connections," in *IEEE 18th High Performance Computing and Communications*, IEEE, 2016, pp. 198–205.

[27] M. S. Z. Nine, K. Guner, Z. Huang, X. Wang, J. Xu, and T. Kosar, "Big data transfer optimization based on offline knowledge discovery and adaptive sampling," in *Big Data (Big Data), 2017 IEEE International Conference on*, IEEE, 2017, pp. 465–472.

[28] D. Yun, C. Q. Wu, N. S. Rao, Q. Liu, R. Kettimuthu, and E.-S. Jung, "Data transfer advisor with transport profiling optimization," in *Local Computer Networks (LCN), 2017 IEEE 42nd Conference on*, IEEE, 2017, pp. 269–277.

[29] Z. Liu, R. Kettimuthu, I. Foster, and N. S. Rao, "Cross-geography scientific data transferring trends and behavior," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ACM, 2018, pp. 267–278.

[30] V. Bhat, M. Parashar, M. Khandekar, N. Kandasamy, and S. Klasky, "A self-managing wide-area data streaming service using model-based online control," in *2006 7th IEEE/ACM International Conference on Grid Computing*, IEEE, 2006, pp. 176–183.

[31] M. J. Branson, F. Douglis, Z. Liu, and F. Ye, *Method for inter-site data stream transfer in cooperative data stream processing*, US Patent 8,688,850, 2014.

[32] I. Alan, E. Arslan, and T. Kosar, "Energy-performance trade-offs in data transfer tuning at the end-systems," *Sustainable Computing: Informatics and Systems*, 2014.

[33] I. Alan, E. Arslan, and T. Kosar, "Energy-aware data transfer algorithms," in *Proceedings of Supercomputing*, 2015.

[34] E. Arslan and A. Alhussen, "A low-overhead integrity verification for big data transfers," in *2018 IEEE International Conference on Big Data (Big Data)*, IEEE, 2018, pp. 4227–4236.

[35] B. Charyyev, A. Alhussen, H. Sapkota, E. Pouyou, M. Gunes, and E. Arslan, "Towards securing data transfers against silent data corruption," in *IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing*, IEEE/ACM, 2019.

[36] S. Liu, E.-S. Jung, R. Kettimuthu, X.-H. Sun, and M. Papka, “Towards optimizing large-scale data transfers with end-to-end integrity verification,” in *Big Data (Big Data), 2016 IEEE International Conference on*, IEEE, 2016, pp. 3002–3007.

[37] B. Charyyev and E. Arslan, “Riva: Robust integrity verification algorithm for high-speed file transfers,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 6, pp. 1387–1399, 2020.

[38] T. Kosar, E. Arslan, B. Ross, and B. Zhang, “Storkcloud: Data transfer scheduling and optimization as a service,” in *Proceedings of the 4th ACM workshop on Scientific cloud computing*, ACM, 2013, pp. 29–36.

[39] Y. Liu, Z. Liu, R. Kettimuthu, N. Rao, Z. Chen, and I. Foster, “Data transfer between scientific facilities–bottleneck analysis, insights and optimizations,” in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019, pp. 122–131.

[40] Y. Kodama, M. Shumway, and R. Leinonen, “The sequence read archive: Explosive growth of sequencing data,” *Nucleic acids research*, vol. 40, no. D1, pp. D54–D56, 2012.

[41] M. Yang, X. Liu, W. Kroeger, A. Sim, and K. Wu, “Identifying anomalous file transfer events in lcls workflow,” in *Proceedings of the 1st International Workshop on Autonomous Infrastructure for Science*, 2018, pp. 1–4.

[42] XSEDE, *Extreme Science and Engineering Discovery Environment*, http://www.xsede.org/, 2018.