Reliable Wide-Area Data Transfers for Streaming Workflows

Hemanta Sapkota and Engin Arslan

Abstract—Many large science projects rely on remote clusters for (near) real-time data processing, thus they demand reliable wide-area data transfer performance for smooth end-to-end workflow executions. However, data transfers are often exposed to performance variations due to the changing network (e.g., background traffic) and dataset (e.g., average file size) conditions, necessitating adaptive solutions to meet stringent performance requirements of delay-sensitive streaming workflows. In this paper, we propose *FStream++* to provide reliable transfer performance for large streaming science applications by dynamically adjusting transfer settings to adapt to changing transfer conditions. *FStream++* combines three optimization methods as *dynamic tuning*, *online profiling*, and *historical analysis* to swiftly and accurately discover optimal transfer settings that can meet workflow requirements. Dynamic tuning uses a heuristic model to predict the values of transfer parameters based on dataset characteristics and network settings. Since heuristic models fall short to incorporate many important factors such as I/O throughput and resource interference, we complement it with online profiling to execute a real-time search for a subset of transfer settings. Finally, historical analysis takes advantage of the long-running nature of streaming workflows by storing and analyzing previous performance observations to shorten the execution time of online profiling. We evaluate the performance of *FStream++* by transferring several synthetic and real-world workloads in high-performance production networks and show that it offers up to $3.6x$ performance improvement over legacy transfer applications and up to 24% over our previous work *FStream*.

Index Terms—Distributed workflows; streaming science applications; high-speed networks; throughput optimization; online profiling

1 Introduction

An increasing number of scientific applications demand reliable wide-area data transfer performance to quickly move the data from data sources to remote processing facilities. For example, The Laser Interferometer Gravitational-Wave Observatories (LIGO) [1] are located in Hanford, WA and Livingstone, LA, but the captured data is streamed to more than ten institutions across the United States (US) and Europe in real-time to take advantage of distributed computing power and enable collaboration. Similarly, astronomical survey project Legacy Survey of Space and Time (LSST) captures images of the southern sky at an observatory (Vera Rubin Observatory) in Chile but requires the data to be transferred to High Performance Computing (HPC) facilities in the US within 7 seconds of data capture to quickly coordinate with other observatories and detect transient events [2].

In addition to online streaming workflows, offline analysis of large scientific datasets also necessitates reliable data transfer performance to overcome storage and compute limitations. For example, when a bioscientist wants to process a large volume of genome sequence datasets stored in central repositories (e.g., NCBI), storage limitations may preclude the scientist from downloading the entire dataset before running the analysis. Thus, even offline data analysis jobs may need to adopt a streaming approach to overcome storage limitations and improve overall execution times. Therefore, devising reliable data transfer solutions is essential more than ever to accommodate this trend towards streaming workflows.

While it is relatively straightforward to attain high transfer throughput when transfer conditions are predictable and stable, it is rather challenging to do the same when the transfer conditions fluctuate over time. For example, dataset characteristics (i.e., average file size and the number of files) of a transfer may evolve based on observation/simulation status. Similarly, the long-running nature of streaming workflows makes them susceptible to changes in background traffic, especially when shared networks are used to run the transfers. Since the optimal transfers settings are highly dependent on the dataset characteristics, network settings, and background traffic conditions [3], [4], [5], an adaptive solution is essential to provide reliable network performance for long-running streaming workflows.

Although there are both proprietary (e.g., Google MillWheel [6] and Amazon Kinesis [7]) and open source (e.g., Apache Kafka [8]) solutions to handle streaming data flows, they are primarily designed to handle a large number of small messages between multiple producers and consumers. On the other hand, scientific applications have significantly higher data generation rates (reaching the orders of gigabytes per second [9]) in addition to storing data in non-volatile storage systems before transferring it to remote clusters. Therefore, commercial streaming frameworks do not fit well for the needs of scientific workflows that require both network and I/O optimizations to fully utilize available resources in high-performance clusters and networks [10].

Moreover, legacy high-speed file transfer applications are not well-suited for streaming workflows for several reasons. *First*, most high-speed data transfer applications (e.g., Globus [11] and bbcp [12]) do not accept new files

• *Hemanta Sapkota and Engin Arslan are with the University of Nevada, Reno*
E-mail: hsapkota@nevada.unr.edu, earslan@unr.edu

978-1-7281-6677-3/20/$31.00 ©2020 IEEE

to transfer queue once the transfer is initiated, hence they require a separate transfer task to be created for every file transfer request. This is, however, not desirable as it incurs channel setup and tear-down costs for every new transfer task. *Second*, they either do not tune transfer settings (e.g., the number of parallel connections and I/O block size) at all and thus yield poor transfer performance or use fixed transfer settings and fail to adapt to changing transfer conditions. *Third*, they do not provide a quality of service (QoS) guarantee for transfer throughput as they rely on best-effort transport protocols (e.g., TCP Cubic) to adjust transfer speeds. While this may be acceptable behavior for delay-tolerant jobs, streaming workflows have more stringent performance needs to effectively pipeline compute and transfer operations and analyze data in real-time. Even though some of these issues may be addressed through point solutions, we believe that a comprehensive solution is needed to satisfy all these requirements and facilitate the execution of streaming workflows in production high-speed networks.

In this paper, we introduce *FStream++* to provide reliable performance for streaming transfers of distributed science workflows. *FStream++* adopts the heuristic approach [13] to determine the initial transfer settings, but complements it with hill-climbing based online profiling to explore search space for a subset of transfer parameters in real-time. The evaluation results show that online profiling is a key to increasing resource utilization in high-speed networks and adapting to changing background traffic conditions. Finally, *FStream++* leverages the long-running nature of streaming workflows and derives regression models to quickly converge to optimal solutions, especially when throughput requirements of application change while the transfer is still running. In summary, we make the following contributions:

- We demonstrate that existing high-speed transfer optimization algorithms fall short to meet the stringent performance requirements of streaming workflows. We also show that application-layer transfer settings can be tuned in real-time to adapt to changing transfer conditions and sustain high performance.
- We design and develop *FStream++* that employs dynamic tuning, online profiling, and historical data analysis methods to offer reliable transfer performance for streaming workflows in the presence of changing dataset characteristics, background traffic, and throughput demands.
- We carry out extensive experiments using high-speed production networks with both synthetic and real-world datasets to compare the performance of *FStream++* against the state-of-the-art transfer applications and services under varying workload and traffic conditions. We find that *FStream++* outperforms the legacy transfer solutions by up $3.6x$ and FStream [14] by 24%.

Compared to our previous work on the optimization of streaming file transfers (*FStream* [14], this paper makes the following unique contributions: First, we use the hill-climbing method to shorten the convergence time of the online profiling module by around 50%. Given that online profiling may need to execute frequently to adapt to changing conditions, reducing its execution time leads to significant improvement in the overall transfer throughput. Second, we introduce a novel online modeling method to efficiently utilize historical profiling reports and swiftly converge to the optimal transfer settings when transfer conditions or quality of service requirements change. Specifically, while *FStream* uses match-action policy to utilize historical transfer logs, which require historical logs to have the same transfer settings as target transfers, *FStream++* derives regression models to mitigate this requirement and predict the performance of previously untested transfer settings. Third, we reveal that data transfers in shared production networks are indeed exposed to significant throughput fluctuations and show that *FStream++* offers reliable performance under such circumstances. Fourth and final, we present experimental results for *FStream++* in non-HPC environments to demonstrate that its optimization strategies are not only applicable to high-speed production networks.

2 Background and Related Work

TCP's poor performance in high-speed wide-area networks has led to the development of new congestion control algorithms, such as BBR [15], PCC [16], and HTCP [17]. While these TCP variants addressed some of the performance limitations, such as severe throughput degradation in the presence of random packet losses, end-host issues (e.g., low I/O throughput and misconfigured data transfer nodes) are increasingly becoming the source of bottlenecks in high-speed networks. ESnet introduced ScienceDMZ [18] architecture to mitigate some of the last mile problems. Precisely, it separates research traffic from regular internet traffic to minimize interference. It also creates an isolated path for science flows to bypass firewalls and other middleware devices. ESnet also documented preferred hardware and software configurations for end hosts (aka Data Transfer Node [19]) to handle high-speed transfers. Although ScienceDMZ and DTN architectures can mitigate some of the performance problems, several others reasons for poor transfer performance remain unsolved, such as low disk I/O performance, and network and I/O congestion.

Researchers also proposed application layer solutions to mitigate the most common performance problems by means of tuning application-level transfer settings. For example, command pipelining [20], parallel connections [21], concurrent file transfer [4], [5], socket buffer size [22], and I/O block size [23], [22] are among application layer parameters that can be tuned to significantly improve end-to-end data transfer performance without the need to change the underlying transfer protocols. Previous studies show that command pipelining (aka pipelining), network parallelism (aka parallelism), and concurrent file transfers (aka concurrency) are the most effective ones in increasing transfer throughput in high-speed networks [11], [3], [4], [13]. In a nutshell, pipelining eliminates the delay between consecutive file transfers by sending multiple transfer commands to source and destination endpoints such that the files can be transferred back-to-back without any delay in between [20]. Parallelism overcomes performance problems of TCP in wide-area networks by creating multiple connections to transfer the different portions of the same file. Finally,

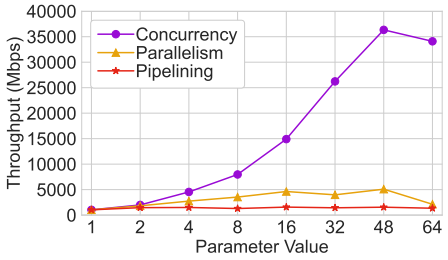


Fig. 1. Application-layer transfer parameters can improve transfer throughput significantly.

concurrency improves both I/O and network performance by transferring multiple files simultaneously on separate network connections. It is particularly important to increase I/O throughput in HPC facilities that are equipped with parallel file systems such as Lustre and GPFS. Please note that concurrency defines how many files to transfer simultaneously, whereas parallelism sets the number of network channels to use for each file transfer. To give an example, if concurrency is set to 2 and parallelism is set to 3, then two files will be transferred simultaneously, and each file will be streamed over three network connections, creating a total of 6 network channels.

Figure 1 shows the impact of pipelining, parallelism, and concurrency on transfer throughput when a set of 1 GiB files is transferred from Expanse [24] to Stampede2 [25] supercomputers that are connected with 40 Gbps network bandwidth. While the default values of these transfer parameters (i.e., one file is transferred ($concurrency = 1$) over one network channel ($parallelism = 1$) and transfer commands are sent one by one ($pipelining = 0$) yield less than 2 Gbps throughput, the optimal values attain more than 36 Gbps throughput. On the other hand, discovering the optimal setting requires domain expertise due to the large search space and complex interplay between the parameters. For example, while parallelism is helpful for large files, it may degrade performance when combined with concurrency. Moreover, using very large values for all parameters is not desirable as it will overload the network and end-system resources and increase power consumption at the end hosts [26]. Therefore, it is essential to discover optimal values that lead to high throughput and low overhead.

Researchers applied heuristic [11], [13], supervised [27], [4], and online [28], [3] learning models to predict the optimal values for these parameters. Globus, a widely adopted data transfer service, relies on heuristic models to predict the values for pipelining, parallelism, and concurrency parameters [11]. Although heuristic approaches perform better than the default settings, they fail to offer reliable performance due to failing to incorporate dynamic system conditions (e.g., network and disk interference) in decision making. For example, if a streaming application initially creates a dataset with predominantly small files, the heuristic models will choose a large pipelining value. However, this choice will become suboptimal if the application starts emitting fewer large files. Supervised models can yield close-to-optimal transfer performance in networks that they are trained for; however, deriving an accurate model requires a large amount of historical data to be collected in a variety of transfer settings (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 up-to-date data to adapt configuration changes (e.g., file system upgrade).

Online learning models offer promising alternatives as they can discover optimal values in the runtime; however, existing solutions in this domain are inadequate to provide a practical option for streaming transfers as their convergence time can take hours. For instance, Yun et al. proposed ProbData [28] to tune the number of parallel streams and buffer size in real-time using stochastic approximation, but the optimization process takes more than two hours to find a solution; thus, it is not suitable for real-time optimizations. Yildirim et al. proposed an online search algorithm to find optimal settings through an online search, but the proposed algorithm executes the search only once at the beginning of transfers, thus not well suited for streaming applications that require dynamic solutions to adapt to changing network settings and dataset characteristics.

3 System Design

As application-layer transfer parameters can be used to address both network and I/O bottlenecks, we introduce *FStream++* to tune these parameters in real-time to provide a robust transfer experience for delay-sensitive distributed streaming workflows. *FStream++* consists of *Data Finder* and *Transfer Controller* components as illustrated in Figure 2. *Data Finder* periodically checks for new files and classifies them based on file size to allow transfer applications to customize the transfer settings (e.g., high pipelining value for small files and high parallelism for large files), which is found to be an effective way of improving transfer throughput [13], [3]. We adopt a similar strategy described in [13] and place files to "small" group if their size is smaller $bandwidth/8$ and to "large" otherwise. Please note that *Data Finder* only captures metadata information of files, including name, path, and size; thus, its network usage is negligible compared to actual file sizes.

On the other hand, the *Transfer Controller* is responsible for selecting and applying transfer settings. To do so, it employs *Dynamic Tuning*, *Online Profiling*, and *Historical Analysis* methods. *Dynamic Tuning* predicts the values for all three parameters (i.e., pipelining, parallelism, and concurrency) using a heuristic model. *Online Profiling* and *Historical Analysis* methods, on the other hand, only tunes concurrency to reduce the execution time of online modeling and searching. As shown in Figure 1, concurrency, when tuned properly, is often sufficient to achieve high performance as it can mitigate both I/O and network bottlenecks. Hence, we rely on the heuristic model to predict the values of pipelining and parallelism and use *Online Profiling* and *Historical Analysis* to fine-tune the concurrency parameter.

3.1 Dynamic Tuning

Dynamic Tuning use an heuristic model to estimate the initial values of pipelining (pp_0), parallelism (p_0), and concurrency (cc_0) using following calculations:

$$\begin{aligned} pp &= \frac{BDP}{avgFileSize} \\ p &= Min(\left\lceil \frac{BDP}{bufferSize} \right\rceil, \left\lceil \frac{avgFileSize}{bufferSize} \right\rceil) \\ cc &= Min(\left\lceil \frac{BDP}{avgFileSize} \right\rceil, maxCC) \end{aligned} \tag{1}$$

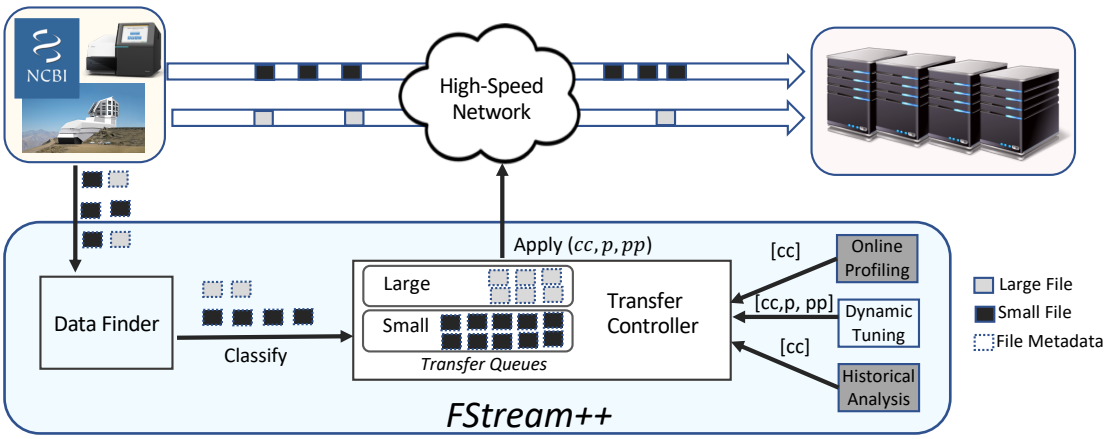


Fig. 2. The system architecture of *FStream++*. *Data Finder* discovers new files at the data source and passes them to *Transfer Controller* after classifying them based on file size. *Transfer Controller* utilizes Online Profiling, Dynamic Tuning, and Historical Analysis optimizations to determine the values of transfer parameters. Despite using similar approaches as FStream [14], *FStream++* extends *Online Profiling* and *Historical Analysis* (highlighted) methods significantly to quickly find the optimal transfer settings.

where BDP is bandwidth-delay product [29] of the network, $bufferSize$ is the maximum allowed TCP buffer size, and $avgFileSize$ is an average file size of the dataset. In a nutshell, the heuristic model returns high pipelining and concurrency values for small files and high parallelism and low-to-moderate concurrency values for large files. As discussed in Section 2, pipelining eliminates the delay between consecutive file transfers, parallelism overcomes TCP buffer size limitations. Since it is important to read/write multiple files simultaneously to take advantage of parallel file systems, especially when transferring a large number of small files, the heuristic model returns a high concurrency value for datasets dominated by small files.

As the data source creates new files, the dataset's characteristics may change. As an example, an application may create a few large files in the first interval and many small files in the next one due to the observation of certain events, which in turn requires the transfer settings to be updated based on the current content of the dataset to sustain high transfer performance. *Dynamic Tuning* thus calculates new values for transfer parameters (cc_1, p_1, pp_1) when new files are added to the transfer queues. Suppose the new values are different than the current ones (cc_0, p_0, pp_0). In that case, the *Transfer Controller* takes the following steps to update them: If the new concurrency value is smaller than the current value, $cc_0 - cc_1$ transfer processes [1] are marked as "passive"– not actively used for transfers but kept alive for potential use in the future. If the new concurrency value is higher than the current one, $cc_1 - cc_0$ additional processes are created. Since parallelism value can only be set when transfer processes are first created, it is not possible to update the parallelism value of a transfer process later. Therefore, *FStream++* first identifies existing processes, cc', whose parallelism value matches with the new parallelism value such that they can be reassigned. If the total number of remaining processes, $cc_0 - cc'$, whose parallelism value is different than the new one, they are restarted with the new parallelism value in addition to creating additional $cc_1 - cc_0$ new processes. Otherwise, $cc_1 - cc'$ of existing processes are restarted with the new parallelism value, and the rest is marked as passive. Since pipelining defines the number of outstanding commands for each transfer process, its value can easily be updated without restarting the connection.

1. Since concurrent transfers are handled by separate processes, the term "process" is used to refer each one of concurrent transfers

Algorithm 1 — Binary search-based *Online Profiling* method to discover the optimal concurrency level.

Global: `Throughput(k)` - Function that returns throughput when using k concurrency. Null if not available.
Input: `current` - current concurrency value; `lowerBound` - current lower bound for concurrency; `upperBound` - curent upper limit for concurrency;
Output: New concurrency value

```
function OptimalConcurrencySearch(lowerBound, current, upperBound)
    while True do
        upThr = Throughput(upperBound)
        lowThr = Throughput(lowerBound)
        curThr = Throughput(current)
        if (upThr == Null) then          ▷ We are still yet to reach maximum throughput, keep increasing
            lowerBound = current
            current = (current + upperBound) /2
            return(current)
        end if
        if (curThr > lowThr and upThr > curThr) then
            lowerBound = current
            current = (current + upperBound) /2
            return(current)
        end if
        if (curThr ≯ lowThr) then    ▷ Lower concurrency if higher values do not result in noticeable (e.g., 1%) throughput gain
            upperBound = current
            current = (current + lowerBound) /2
        else                         ▷ Inconclusive results, lower current value slightly
            current = current − 1
        end if
    end while
    return(current)
end function
```

Online Profiling

Although *Dynamic Tuning* adjusts the transfer settings by re-evaluating the Equation 1 periodically, its performance is inherently limited to the performance of the heuristic model. Previous work showed that heuristic models fail to guarantee high performance as transfer throughput depends on many factors that existing heuristic techniques cannot incorporate into their prediction policies, such as the degree of background traffic and file system configurations [4]. For example, Globus [11] adopts a heuristic model and sets the concurrency value to 2 when the average file size of

a dataset is larger than BDP, however, this mainly results in poor I/O performance due to failure to take full advantage of I/O parallelism, especially when using parallel file systems. Similarly, heuristic models are oblivious to background traffic, thus they fail to ensure performance guarantee in shared networks as optimal transfer settings are highly dependent on network congestion. Consequently, despite yielding higher performance compared to one-time optimization approaches (e.g., [3], [30]), *Dynamic Tuning* is constrained by the limitations of the heuristic method. We therefore introduce *Online Profiling* that executes real-time search to discover optimal transfer settings. Since evaluating the performance of a transfer setting takes several seconds [31], searching the optimal value for multiple transfer parameters in real-time can take a long time (i.e., minutes or even hours [28]). Thus, *Online Profiling* focuses only on concurrency as it can help to mitigate both I/O and network bottlenecks.

Algorithm 1 describes how *Online Profiling* explores the search space for the concurrency parameter. It takes the current concurrency value ($current$) as an argument and returns a new concurrency value using a binary search-based hill-climbing algorithm. It defines an upper ($upperBound$) and lower ($lowerBound$) boundary for concurrency and determines the search direction as well as step size when choosing a new value. The initial values of boundaries are configurable with default values of 1 for $lowerBound$ and 40 for $upperBound$. The $Throughput(x)$ function returns the throughput of concurrency value x based on measurements. If x is not evaluated yet or it was evaluated a long time ago, then it returns $Null$, indicating that it is not available. Even though The $Throughput$ method only returns the results from the last 60 seconds to ensure that it can adapt to changing network conditions. As an example, if background traffic changes, the throughput of all concurrency values will be negatively affected, thus using values that are obtained in different traffic conditions will cause confusion and lead to incorrect estimations. Thus, we limit the throughput history to close to the execution time of the online search algorithm (around 60 seconds as presented in the results) to be able to keep exploring the search space, thereby adapting to changing network and end-system traffic conditions.

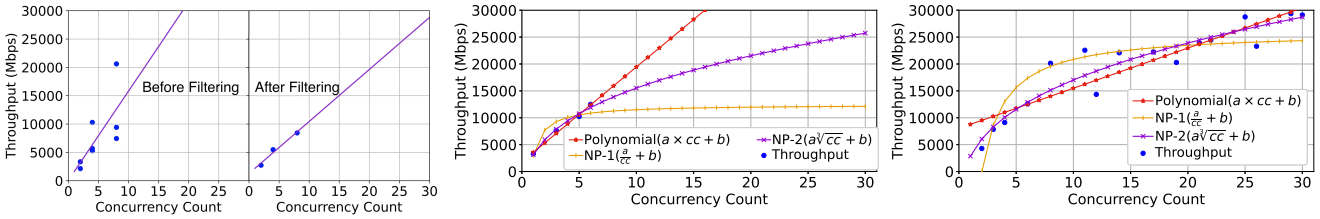
When $OptimalConcurrencySearch$ is method is called, the throughput of upper and lower concurrency boundaries may not be available (i.e., line 6), in which case we choose to keep the search in the upward direction to explore high concurrency values first. As we explore the solution space using the binary search, upper and lower boundaries will eventually be set to one of the evaluated concurrency values (lines 7, 12, and 17). Please note that while it is possible to evaluate the performance of boundary values right at the beginning of the search to make more precise decisions, it may have an adverse impact on the system performance due to creating too many I/O threads and network connections, especially if the optimal solution is far from the initial boundary values. Thus, we take a moderate approach and increase/decrease the value of concurrency in any direction gradually. Note that similar to the expiration of throughput history for concurrency values after a certain time, we also reset lower and upper boundaries to their initial values (1 and 40, respectively) to detect and adapt to changing transfer conditions. If the throughput of the currently evaluated concurrency value is not noticeably (e.g., 1%[2]) higher than the throughput of the lower bound, then *FStream++* prefers lower concurrency values to minimize system overhead while achieving close-to-maximum performance as shown in line 16.

Unlike *FStream* which increases its step size slowly, *FStream++* employs binary search to guarantee logarithmic convergence time. Specifically, *FStream* starts with a small concurrency value (i.e., 1) and doubles it as long as throughput increases, thus it takes several intervals before it can take large steps. For example, it takes five intervals for *FStream* to increase the concurrency value 1 to 16. On the other hand, *FStream++* can quickly jump to high values as it sets the concurrency value to the middle between current value and upper boundary as shown in lines 8 and 13. More importantly, *FStream++* is able to quickly move out of suboptimal zone by taking large steps at the beginning. For instance, if the $upperBound$ is set to 32 and the optimal concurrency is 32, then it will take $6-7$ steps for both *FStream* and *FStream++* to converge to the optimal. However, *FStream* spends 5 of 6 six steps on concurrency values that are less than half of the optimal, i.e., $\{1, 2, 4, 8, 16, 32\}$. In contrast, *FStream++* will test concurrency values in $\{1, 16, 24, 28, 30, 31, 32\}$ order. This in turn leads to average concurrency value of 10.5 for *FStream* and 21.6 for *FStream++* in the first six intervals, which allows *FStream++* to yield higher average throughput.

It is fair to say that *FStream* will converge to small optimal concurrency values faster than *FStream++*. Yet, taking larger steps is still preferable because of two reasons. First, as network capacity of high performance networks keeps increasing, optimal concurrency level for transfers is likely to be on the higher end of the potential values. As presented in the Evaluation section, the optimal concurrency value is more than 10 in all production networks while reaching to 40 in some networks. Second, despite incurring higher overhead, large concurrency values do not typically lead to significant drop in transfer throughput. As an example, even if it takes more steps for *FStream++* to converge to the optimal concurrency value of 4 compared to that of *FStream*, their overall throughput will be similar.

Once a new concurrency value is determined by $OptimalConcurrencySearch$ method, *FStream++* adjusts the number of concurrent processes to evaluate the performance of the new setting. To do so, we first utilize passive transfer processes before creating new ones to speed up the transition phase. As *FStream++* keeps track of the parallelism value of passive processes, they can be re-assigned to file groups (i.e., small or large) with the same parallelism value. For example, when *FStream++* decides to activate a passive process with parallelism value of 4, it will attempt to assign it to large file group, assuming that the large file group demands parallelism value 4. By doing so, it avoids to restart processes which speeds up the transition process. Please note that even though *Dynamic Tuning* also predicts

2. We define the noticeable increase rate as a value that is slightly higher than measurement noise. We observed that 1% increase expectation is sufficient in all experimental networks, but leave the optimization of it to future work.



(a) Data Preprocessing (b) Model representation with limited data (c) Model representation with full data

Fig. 3. Data preprocessing is used to remove outliers in historical data (a). Representation of regression model performance with limited (b) and full data (c). Model $a\sqrt[3]{cc} + b$ performs well both using limited and full data, thus can be used to derive models in the runtime.

the value of concurrency, *FStream++* will use the estimations done by *Online Profiling* as real-time search is a superior method over the heuristic model. However, *Dynamic Tuning* will still be used to adjust the level of parallelism and pipelining parameters.

3.2 Historical Analysis

Execution of *Online Tuning* can take up $50 - 60$ seconds (around $5 - 6$ steps with 12 seconds per step) to find a solution, which can have adverse impact on the performance especially if the throughput requirement of an application changes frequently. For example, when an application starts to produce more data than its usual rate for a short period of time due to transient events, it may demand higher throughput to handle the surge in a timely manner. Running *Online Tuning* from scratch is therefore undesirable due to relatively long execution time. *FStream++* thus exploits the long running nature of streaming workflows and caches previous profiling results to re-use them when applications' throughput requirement change. To do so, *FStream++* uses regression analysis on historical profiling results to derive a relationship between concurrency value and transfer throughput such that it can be used to make predictions for previously unseen throughput values. As an example, if *Online Profiling* tested concurrency values of $1, 16, 24$, and 26 recently while searching for the optimal concurrency value, we can use the obtained results to derive a regression model which can be used to predict the throughput of concurrency value of 8.

In the previous work, we proposed a simple match-action policy to utilize historical profiling reports, which requires an exact match between historical reports and target throughput utilize the information stored in the historical data. For example, if the historical data contains throughput reports for concurrency values 2 (5 Gbps) and 16 (20 Gbps), but application updates its throughput demand to 15 Gbps, then *FStream*'s simple match policy fails to utilize available reports since it is unable to extrapolate.

We apply few polynomial and non-polynomial models to discover the relationship between concurrency value and transfer throughput. As a polynomial model, we choose linear regression in the form of $t = a \times cc + b$, where a and b are coefficients, cc is concurrency value, and t is throughput. We apply $t = \frac{a}{cc} + b$ (*NP-1*) [32] and $t = a\sqrt[3]{cc} + b$ (*NP-2*) as non-polynomial models. Unlike the linear regression, the non-polynomial models can capture the diminishing impact of concurrency on transfer throughput. As can be seen in the Figure 1, while concurrency has somewhat linear relationship with the throughput initially, its impact starts to disappear at high values. The non-polynomial models differ in terms of how fast the impact of concurrency on transfer throughput disappears. Specifically, *NP-1* expects that throughput gain for increased concurrency will decline much faster compared to *NP-2*.

In regression analysis, we first preprocess historical data to remove outliers as throughput fluctuations are common in shared production networks due to transient issues such as bursty background traffic or I/O interference. We then categorize throughput reports based on their concurrency values in case there are multiple throughput reports for the same concurrency value[3]. We next calculate z-scores for each throughput report in the same concurrency category, which is commonly used for outlier detection [33]. The throughput reports with z-scores greater than 1 or less than -1 are discarded as they deviate from the average significantly. We finally take the average of remaining reports to represent the throughput of a concurrency value. We repeat the process for all distinct concurrency values before fitting the models. Figure 3(a) illustrates the impact of the described preprocessing on the performance of the linear regression. We observe that filtering out outliers significantly improves the estimation accuracy of the model.

Figure 3(b) and 3(c) illustrates the fitness of the regression models based on available historical data. When a transfer starts with a small throughput demand (e.g., 5 Gbps in Figure 3(b)), the historical data will only contain reports for low concurrency values. Absence of data for high concurrency values in turn causes linear regression to overestimate and *NP-1* to underestimate the throughput of high concurrency values. For instance, *NP-1* (i.e., $\frac{a}{cc} + b$) predicts that concurrency value of 30 will return around 10 Gbps throughput whereas the linear regression model estimates that the concurrency value of 15 will be sufficient to achieve 30 Gbps; both of which are far from actual observations. On the other hand, *NP-2* (i.e., $t = a\sqrt[3]{cc} + b$) is able to capture the relationship between concurrency and throughput more accurately in a way that it incorporates the diminishing impact of the concurrency while avoiding to converge too fast. As the transfer executes long enough and gathers throughput reports for higher concurrency values (i.e., as a result of setting the target throughput to higher values and executing *Online Profiling*), the linear regression and *NP-1* models evolve in a way the the slope of the linear

3. Since we execute a transfer setting for multiple intervals to accurately measure its performance, there will be multiple throughput reports for the same transfer setting.

Workload	Time Interval					
	1	2	3	4	5	6
1	S	L	S	L	S	L
2	L	S	L	S	L	S
3	XS	XL	L	L	L	L
4	S+L	S+L	S+L	S+L	S+L	S+L

TABLE 1
File types and arrival orders for synthetic datasets. File sizes range between 0-10MB (Extra Small (XS)), 10-100MB (Small (S)), 100MB-2GB (Large(L)) and >2GB (Extra Large (XL)).

regression decreases and convergence throughput for *NP-1* increases to 20 Gbps as shown Figure 3(c). Yet, they still result in high error rates, therefore, we use *NP-1* as a default regression model for *FStream++* to process the historical data. Since network conditions may change over time, we define maximum duration for an historical report to be included in the model. By default, it is set to 10 minutes but it can be extended to implement an adaptive method that adjusts the time limit based on the estimation error of the model. In other words, if the predictions made by a model turns out to be significantly far from the actual observations, then the older reports in the historical data can be removed before 10 minute threshold to quickly adapt to new conditions. We leave such optimization as a future work as we observe that background traffic in production networks typically change in the order of tens of minutes or hours.

4 Evaluation

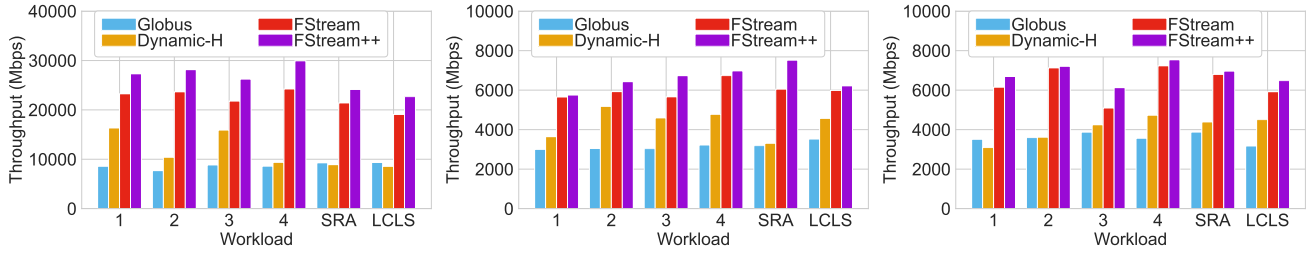
We used both synthetic and real-world workloads to evaluate the performance of *FStream++*. For synthetic workloads, we created multiple workloads with different data generation behaviors (i.e., average file and arrival orders), as shown in Table 1. Since streaming workflows produce a new file(s) at certain intervals, we add new files to the transfer directory at 30-second intervals to emulate streaming applications. We used files with different sizes as *Extra Small (XS)* (smaller than 10MB), *Small (S)* (between 10MB and 100MB), *Large (L)* (between 100MB-2GB), and *Extra Large (XL)* (larger than 2GB) to thoroughly evaluate the performance of *FStream++*. Specifically, Workloads 1, 2, and 3 are designed to evaluate the performance of the transfer applications under extreme scenarios in which the average size files changes drastically in consecutive intervals. For example, Workload 1 and 2 represent cases where streaming applications emit new files with completely different characteristics in successive intervals. On the other hand, Workload 4 represents a more stable data generation pattern. Note that if there are still files in the transfer queue when new files arrive, we add the new files to the end of the transfer queue, as shown in Figure 2.

In addition to synthetic workloads, we also used real-world workloads to represent 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 [34]. Although SRA workload is not generated in real-time, the massive size of the repository when combined with storage limitations of computing clusters necessitates users to download and process files in a streaming manner. As of January 2020, NCBI SRA stores 2.8M public genome sequence for "homo sapiens" with a total size of 1.4 PB. We took a sample of this dataset whose file size and file count distribution is shown in Figure 5. Although we significantly reduced the number of files due to time/storage constraints, the sampled subset follows a similar file size distribution as the original one.

Linac Coherent Light Source (LCLS) takes X-ray snapshots of atoms and generates terabytes of data per experiment, which is moved to supercomputers located in Berkeley, CA (Cori), Chicago, IL (Mira), and Oak Ridge, TN (Titan) in near real-time to carry out data analysis, thus it is an ideal example to demonstrate the potential benefits of improved wide-area transfer performance. The LCLS workload is dominated by large files as the average file size is 13.1 GB and the median file size of 4 GB [35]. Similar to synthetic workloads, we split the SRA and LCLS workloads into six groups randomly and added them to the transfer directory one by one in 30-second intervals. Besides changing dataset characteristics, we also evaluate the impact of dynamic network conditions in Section 4.3. We used three pairs of XSEDE [36] sites as Stampede2 [25] to Expanse [24], OSG [37] to Stampede2, and OSG to Expanse to run transfers, which are connected via a shared high-speed network with up to 40 Gbps bandwidth. The round trip times are 38ms, 32ms, and 58ms for Stampede2-Expanse, OSG-Stampede2, and OSG-Expanse transfers, respectively.

We compare *FStream++* against Globus [11], an adaptive version of the heuristic model [30] (*Dynamic-H* henceforth), and *FStream* [14]. Although all of the transfer algorithms use GridFTP as a transfer protocol, they configure its settings (i.e., pipelining, parallelism, and concurrency) differently[4]. Globus uses fixed transfer settings based on the average file size of a dataset or based on endpoint-specific predefined values. For example, it uses $< 20, 4, 4 >$ values for <pipelining, parallelism, concurrency> parameters for all Stampede2-Expanse transfers. Moreover, since Globus is a closed-source transfer service and does not allow new files to be added to ongoing transfer tasks, we initiate a separate transfer task for each batch of files in consecutive intervals using Globus Command Line Interface (CLI). If a new batch of files arrives while the transfer of previous files is still running, Globus will allow both transfers to run in parallel as long as the number of active transfers does not exceed 3. Please note that the performance of Globus will be higher when multiple of its instances execute in parallel to transfer a set of files that arrive in different intervals; one cannot rely on this as it heavily depends on the time gap between the arrival of new files. In other words, if the transfer of previous files is completed when new files arrive, Globus will not benefit from concurrent execution of multiple instances. Nevertheless, we used Globus "as-is" and submitted new transfers as soon as they arrived (i.e., in 30-second intervals). We also turned off the integrity verification as none of the compared solutions implements it. The *Dynamic-H* uses the heuristic model (Algorithm 1) to

4. Note that even though we utilized GridFTP as a transfer protocol, the parameters *FStream++* tunes are either readily available or can easily be implemented in other protocols such as FTP and bbcp



(a) Stampede2-Expanse (b) OSG-Stampede2 (c) OSG-Expanse

Fig. 4. Performance comparison of different algorithms in Stampede2-Expanse (a), OSG-Stampede2 (b), and OSG-Expanse (c) networks for six workload types. *FStream++* outperforms Globus, Dynamic-H, and *FStream* by up to $3.6x$, $3.1x$, and $1.24x$, respectively.

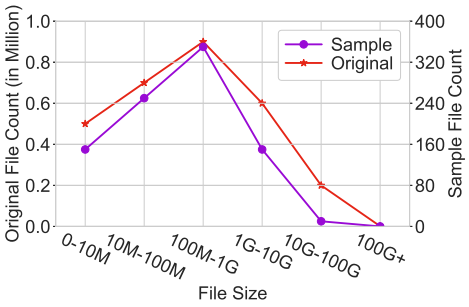


Fig. 5. File size distribution of original and sampled SRA dataset.

estimate the values of transfer parameters at the beginning of the transfer and updates them every time a new set of files arrive by rerunning Equation 1, similar to *Dynamic Tuning* component of *FStream++*. Thus, comparisons against the *Dynamic-H* demonstrate the benefit of *Online Profiling* and *Historical Analysis* components. Finally, *FStream* uses an adaptive approach to configure transfer settings similar to *FStream++*; however, the two differ in terms of how *Online Profiling* and *Historical Analysis* operates as detailed in Section 3.1 and 3.2. We repeat the transfers five times and report the average values.

4.1 Changing Workload Characteristics

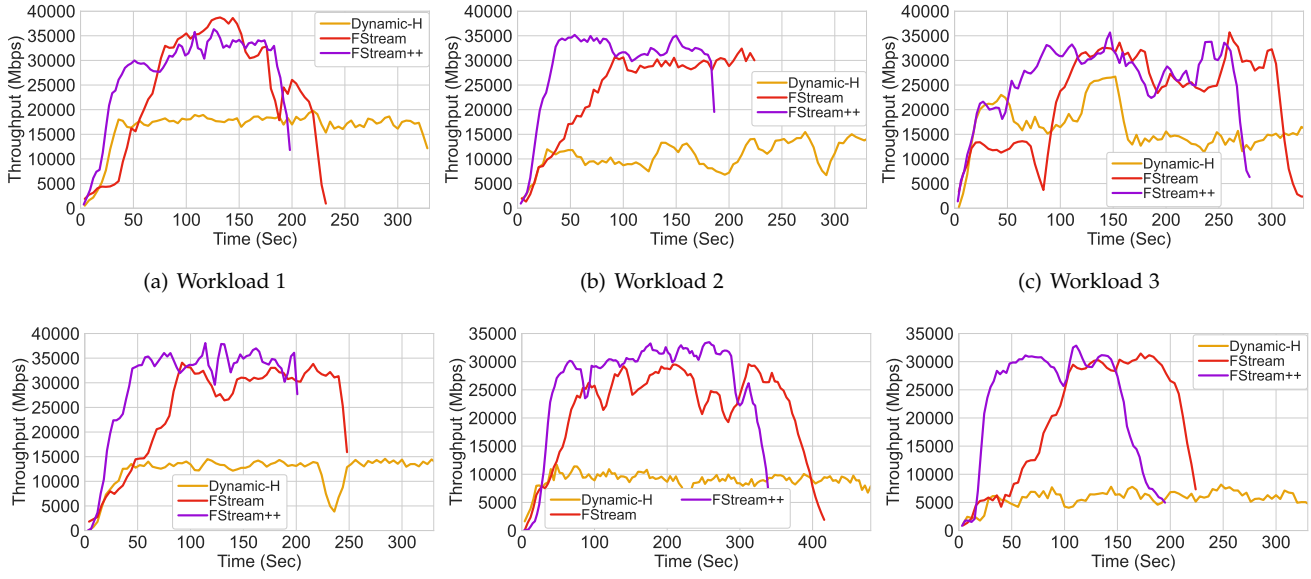
The average throughput results for all workload types are given in Figure 4. Adaptive approaches *Dynamic-H*, *FStream*, and *FStream++* outperform Globus, which adopts a static approach when tuning transfer settings. Specifically, they yield up to $1.9x - 3.6x$ higher throughput than Globus, especially for synthetic workflows with changing datasets in consecutive intervals (i.e., Workload 1, 2, and 3). As an example, *Dyanmic-H* attains up to $1.9x$, $1.7x$, and $1.4x$ higher throughput compared to Globus for Stamped2-Expanse, OSG-Stampede2, and OSG-Expanse, respectively. Please note that the performance gap would be much higher if Globus transfers are initiated sequentially to let one Globus transfer instance run in parallel, similar to other algorithms. Regardless, its performance still stays below *Dynamic-H* in most cases and much lower compared to *FStream* and *FStream++*. Despite yielding better performance than Globus, *Dynamic-H* falls below *FStream* and *FStream++* significantly, which demonstrates the importance of real-time optimization. Specifically, *FStream* outperforms *Dynamic-H* by up to $2.6x$, $1.83x$, and $2.0x$ in Stampede2-Expanse, OSG-Stampede2, OSG-Expanse transfers, respectively.

FStream++ attains up to $20 - 25\%$ higher throughput compared to *FStream* in Stampede2-Expanse, OSG-Stampede2, and OSG-Expanse transfers. The difference can be attributed to the efficient implementation of *Online Profiling*. As detailed in Section 3.1, unlike *FStream* which increases its step size slowly when scanning search space for concurrency parameter, the use of binary search-based hill-climbing method helps *FStream++* to take bigger steps initially and move out of the suboptimal region of the search space quickly. As transfers in the Stampede2-Expanse network obtain the highest throughput, we only present results for that network in the rest of the paper.

Figure 6 presents the instantaneous throughput of *Dynamic-H*, *FStream*, and *FStream++* for one of the five transfers whose average results are shown in Figure 4(a). *Dynamic-H* yields more than 20 Gbps only for Workload 3, in which the dataset in the first interval consists of very small files. This is because the heuristic model returns large concurrency values when the average file size of a dataset is very small. Both *FStream* and *FStream++* are able to reach close to 35 Gbps instantaneous throughput all workloads with the help of online optimization, however, there is a clear difference in how fast they can reach maximum speed. Except for Workload 3, it takes nearly 100 seconds for *FStream* to reach 30 Gbps instantaneous throughput, whereas it takes less than 50 seconds for *FStream++*. Similar to synthetic workloads, *FStream++* outperforms *FStream* for the transfer SRA and LCLS workloads as shown in Figure 6(e) and 6(f). The benefit of using a fast-converging algorithm (i.e., binary search in *FStream++* vs. linear search in *FStream*) in the online search phase is more visible in LCLS workload transfer since suboptimal transfer settings return significantly lower throughput when datasets are dominated by small files.

4.2 Stable Transfer Performance

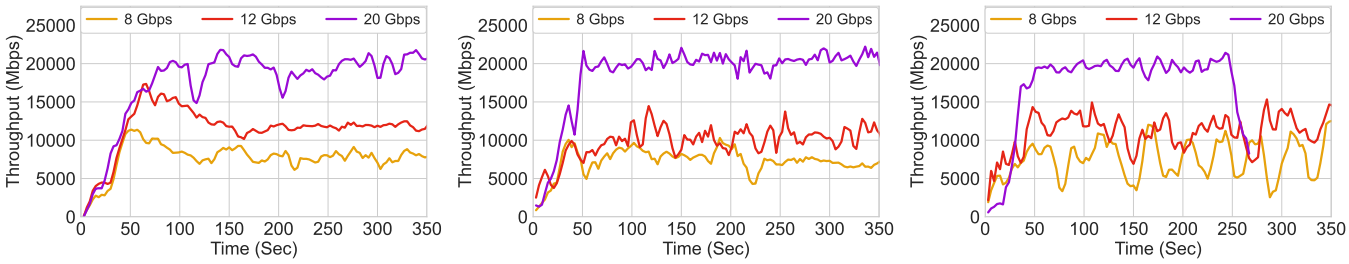
By default, *Online Profiling* searches for a concurrency value that can yield maximum transfer throughput. However, trying to maximize the throughput for all applications may not be the best objective, especially when using shared networks due to fairness concerns. Thus, we allow applications to specify the desired throughput for their transfers such that *FStream++* can focus on satisfying the application demand rather than maximizing the throughput. While one may argue that obtaining higher or lower throughput does not affect total transfer size, keeping the throughput at a minimum possible rate can have a considerable impact



(a) Workload 1 (b) Workload 2 (c) Workload 3

(d) Workload 4 (e) SRA Dataset (f) LCLS Dataset

Fig. 6. Instantaneous throughput of the dynamic heuristic (*Dynamic-H*), *FStream* and *FStream++* for synthetic and real-world datasets in Stampede2-Expanse network.



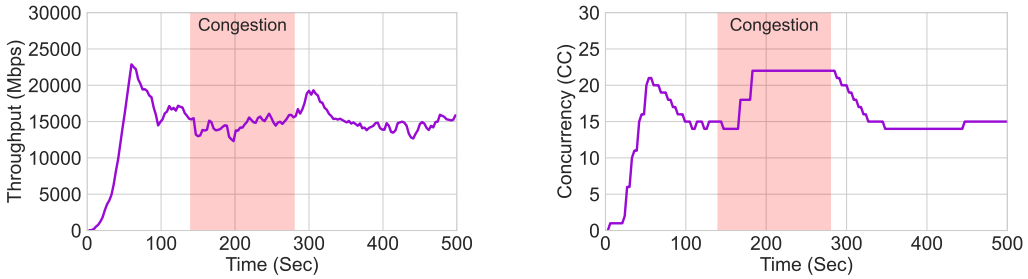
(a) Workload 1 (b) SRA Dataset (c) LCLS Dataset

Fig. 7. *FStream++* can be used to attain fixed, stable performance for workflows that do not require maximum transfer throughput for their execution.

on competing flows. For example, running a transfer with 6 Gbps throughput for 30 minutes versus with 3 Gbps throughput for 60 minutes will transfer the same amount of data, the transferring at a higher speed will have more adverse impact on contending flows due to using more of available resources during the transfer period. Moreover, transferring at very high speeds can overwhelm storage space at the destination facility if processing speed is much slower compared to transfer speed. Hence, *FStream++* supports fixed-speed transfers for streaming applications to keep transfer throughput within a close range of the desired value.

To achieve this goal, *FStream++* first adjusts the level of concurrency to bring the transfer throughput within the 20% range of the desired throughput, then configures pipelining and parallelism parameters to move it to the 10% range of the target. Figure 1 shows that while concurrency can be used to make significant improvements for transfer throughput, parallelism and pipelining can help to make small adjustments. Therefore, we pick parallelism for large files and pipelining for small files and increase/decrease their values by one until transfer throughput falls within the 10% range of the expected value. Note that *FStream++* still uses the binary search algorithm with a slightly different logic to stop increasing the concurrency once the throughput is 20% higher than the desired value.

To assess the effectiveness of the fixed throughput method, we transferred Workload 1, LCLS, and SRA workloads between Stampede2 and Expanse clusters. Figure 7 shows instantaneous transfer throughput when the desired throughput is set to 8 Gbps, 12 Gbps, and 20 Gbps. Except for the initial 50 seconds of transfers during which *Online Profiling* is executed for the first time, *FStream++* is able to keep the throughput within 10% of the desired throughput for all three datasets. In Workload 1, transfer throughput decreases significantly at around $120s$ and $210s$ when the desired throughput is set to 20 Gbps as a result of finishing the transfer of all large files. However, *FStream++* is able to recover from this quickly by readjusting the values of concurrency and pipelining for the remaining small files. In addition, transfer throughput fluctuates more for the LCLS dataset, especially for smaller target throughput values (e.g., 8 Gbps), which can be attributed to the fact that it is more challenging to fine-tune transfer throughput at lower speeds. Specifically, concurrency value of 1 returns $5-6$ Gbps throughput whereas concurrency value 2 returns $10-11$ Gbps. Since *FStream++* tries to configure the concurrency value until the throughput is within the 20% range



(a) Throughput (b) Concurrency

Fig. 8. Performance analysis of *FStream++* under varying background traffic conditions when transferring SRA workload between Stampede2 and Expanse clusters. It is able to keep the throughput very close to 15 Gbps desired throughput by transfer settings, one of which is concurrency.

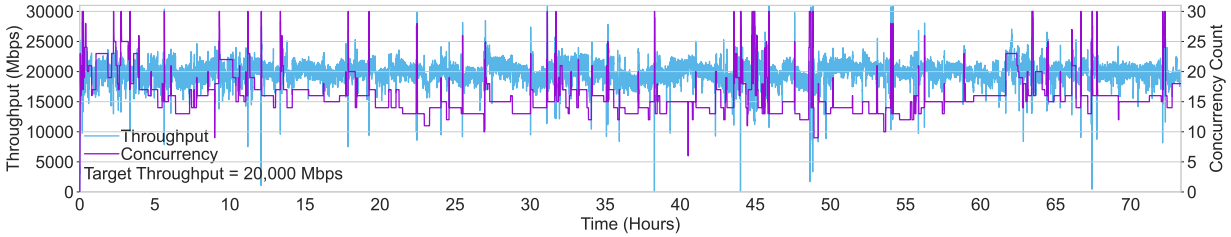


Fig. 9. The performance of *FStream++* in Stampede2-Expanse network when the desired throughput is set to 20 Gbps. *FStream++* changes the transfer settings to react to changing background traffic events, thereby providing reliable throughput performance to streaming transfers.

of the desired throughput, it is unable to achieve it due to large throughput changes between the two consecutive concurrency values.

4.3 Dynamic Background Traffic

In addition to changing dataset characteristics, background traffic may also vary over time, impeding the reliability of transfers. Yet, *FStream++* can detect and mitigate the impact of changing background traffic by periodically executing *Dynamic Tuning* and *Online Profiling* to readjust the transfer settings, since from *FStream++*'s perspective, throughput degradation triggers the same action regardless of the root cause. Figure 8 shows transfer throughput when the desired throughput is set to 15 Gbps for the transfer of SRA workload in the Stampede2-Expanse network. Although there is always some background traffic between Stampede2 and Expanse clusters due to being production systems, we inject additional traffic by running a memory-to-memory transfer between them to intensify the degree of the congestion. We start the transfers under normal background traffic and inject the additional traffic at around $140s$. We let the additional traffic run for 140 seconds and terminate it at around $280s$. The interval with additional traffic is marked with a "Congestion" label in the figure. We present both throughput and concurrency values to demonstrate *FStream++*'s reaction to the changing background traffic.

When the transfer first starts, *FStream++* uses the *Dynamic Tuning* and *Online Profiling* to discover the transfer settings that can satisfy the throughput requirement. When the background traffic is injected at $140s$, the throughput declines by 15%, which triggers *FStream++* to increase the concurrency value to bring the transfer throughput back to the desired level. When the background traffic is terminated at around $280s$, the transfer throughput increases sharply since the concurrency value used during the congestion period results in higher transfer throughput when the network

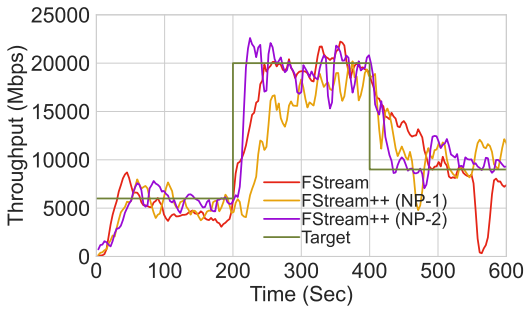


Fig. 10. Comparison of different regression models for online modeling. The non-polynomial model $t = a\sqrt[3]{cc} + b$ achieves faster convergence to target throughput.

is less congested. Hence, *FStream++* lowers the concurrency value to roughly what it was before the congestion to again bring the throughput to its desired range.

We also assessed the performance of *FStream++* for long transfers in the Stampede2-Expanse network for SRA workload. We set the desired throughput to 20 Gbps and run the transfer for three days. Since Stampede2 and Expanse are production clusters and connected by a shared network (i.e., XSEDENet), I/O and network interference are inevitable which affects the performance of long transfers even without injecting background traffic manually. Figure 9 shows that while concurrency value of $15-20$ is sufficient to satisfy throughput demand for most of the time, *FStream++* sets to as low as 10 and as high as 30 when the transfer throughput falls outside of the desired range, demonstrating the effectiveness of *FStream++* to sustain high performance in a production shared network.

4.4 Changing Throughput Requirement

Long-running streaming workflows may demand higher transfer performance at certain times to handle a sudden increase in data rates caused by transient events. For example,

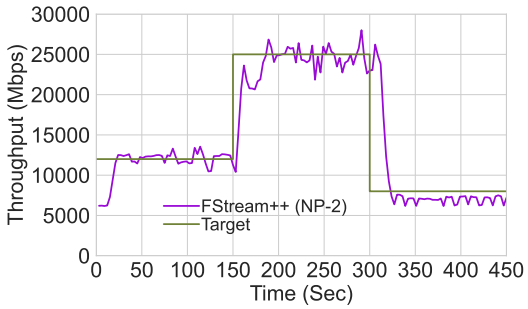


Fig. 11. Demonstration of *FStream++* in controlling transfer settings to meet desired transfer throughput between two data transfers nodes located in the same local area network.

Vera Rubin Observatory requires high-resolution images captured in Chile to be transferred to High Performance Computing (HPC) facilities in the United States within 7 seconds of data capture to quickly coordinate with other observatories and detect transient events [2]. As detailed in Section 3.2, although *Online Profiling* can adjust the transfer settings to meet new requirements, it can take $50 - 60$ seconds to find the optimal. Thus, *Historical Analysis* is designed to leverage the long-running nature of streaming workflows to cache previous performance observations such that they can be applied when throughput demand changes. Different from *FStream*'s match-action policy which requires an exact match in the historical data to benefit from it, *FStream++* implements online regression analysis to be able to predict transfer settings that are not evaluated previously.

Figure 10 evaluates the performance of two regression models along with *FStream*'s match-action policy. When target throughput is increased from 6 Gbps to 20 Gbps at $200s$, *FStream++* with *NP-2* is able to predict close-to-optimal concurrency value and converges to the desired throughput in less than 20 seconds whereas it takes close to 80 seconds for *NP-1*. Note that while *NP-1* is unable to find a solution, *Online Profiling* takes over and finds the right concurrency value through online search, hence they all converge to the optimal eventually. However, it takes longer for *Online Profiling* to converge to a solution, thus *Historical Analysis* using accurate regression models such as *NP-2* offers an opportunity to speed up the search process. It is worth noting that, when throughput is decreased to 9 Gbps, *FStream++* with *NP-2* again converges to the optimal setting faster than the other approaches, including the match-action method as adopted by *FStream*. *NP-2* reduces the convergence time from 45 seconds to 20 seconds (55% reduction) when target throughput is increased and reduces it from 80 seconds to 40 seconds when target throughput is decreased. Please note that since we wait for all active and pipelined transfers to complete before closing a channel (i.e., reducing concurrency value), it takes longer to converge to the optimal setting when target throughput is decreased.

Finally, we evaluate the performance of *FStream++* in non-HPC settings to demonstrate its effectiveness. Figure 11 shows instantaneous throughput for *FStream++* when it is used to transfer the SRA workload between two data transfer nodes in the local area network. The nodes are connected with a 40 Gbps link and use a RAID array with 4 NVMe SSD drives. We initially set the desired throughput to 12 Gbps but changed it to 25 Gbps and 8 Gbps later. When target throughput is increased from 12 Gbps to 25 Gbps, *FStream++* increases concurrency level from 2 to 12 first then settles at concurrency value 10. Similarly, it reduces concurrency value to 1 when target throughput is reduced to 8 Gbps at 300s to lower its throughput to the desired level.

5 Conclusion

As scientific applications demand reliable and high performance for wide-area data transfers to process data near real-time, it is becoming increasingly important to design streaming transfer applications that can meet the stringent demands of these applications. Existing transfer applications are designed for batch transfers, thus they fail to meet the expectations of streaming scientific workflows due to a lack of support for reliable performance. Therefore, this paper introduces *FStream++* to provide reliable transfer performance for large distributed streaming projects. *FStream++* implements multiple optimization methods to adjust transfer settings in real-time to adapt to changing network and dataset conditions, thereby sustaining high transfer throughput. The results gathered in various production and dedicated networks using synthetic and real-world workloads show that *FStream++* yields nearly $3.6x$ higher throughput than legacy transfer applications. It is also able to outperform its previous implementation, *FStream*, by up to 24% with the help of a binary search-based hill-climbing model, which helps to reduce the convergence time of the search phase significantly. In addition to maximizing transfer throughput, *FStream++* also allows transfer throughput to be set to a fixed rate such that it will try to adjust the transfer settings to keep the transfer throughput close to the desired rate even if higher throughput is possible. As future work, we plan to incorporate *FStream++* to commonly used workflow execution frameworks (e.g., Pegasus [38]) to facilitate its adoption by the research community. Moreover, we intend to extend *FStream++* with end-to-end integrity verification to detect and mitigate undetected errors that can happen while transmitting data in the network or while writing it to storage [39], [40].

6 Acknowledgments

This project is in part sponsored by the National Science Foundation (NSF) under award number OAC-1850353.

References

[1] "Laser Interferometer Gravitational-Wave Observatory (LIGO)," https://www.ligo.caltech.edu/, 2021.
[2] "High energy physics network requirements review final report," https://escholarship.org/uc/item/78j3c9v4, 2020.
[3] 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.
[4] 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.

[5] M. Arifuzzaman and E. Arslan, "Online optimization of file transfers in high-speed networks," in *High Performance Computing, Networking, Storage and Analysis, SC21: International Conference for*. IEEE, 2021.

[6] 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.

[7] "Kinesis," 2020, http://aws.amazon.com/kinesis/.

[8] "Apache Kafka," 2020, https://kafka.apache.org/.

[9] "Processing: What to record?" 2020, "https://home.cern/about/computing/processing-what-record".

[10] G. Fox, S. Jha, and L. Ramakrishnan, *STREAM2016: Streaming Requirements, Experience, Applications and Middleware Workshop*, Oct 2016. [Online]. Available: http://www.osti.gov/scitech/servlets/purl/1344785

[11] 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.

[12] "BBCP," https://www.slac.stanford.edu/ abh/bbcp/, 2015.

[13] 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.

[14] D. Ucar and E. Arslan, "Streaming file transfer optimization for distributed science workflows," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2020, pp. 187–197.

[15] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "Bbr: Congestion-based congestion control," *Queue*, vol. 14, no. 5, p. 50, 2016.

[16] M. Dong, T. Meng, D. Zarchy, E. Arslan, Y. Gilad, B. Godfrey, and M. Schapira, "{PCC} vivace: Online-learning congestion control," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 343–356.

[17] "H-tcp - congestion control for high delay-bandwidth product networks," https://www.hamilton.ie/net/htcp.htm, 2021.

[18] "ESNet Science DMZ," 2020, https://fasterdata.es.net/science-dmz/.

[19] "Data transfer nodes," 2018, http://fasterdata.es.net/science-dmz/DTN/.

[20] J. Bresnahan, M. Link, R. Kettimuthu, D. Fraser, I. Foster *et al.*, "Gridftp pipelining," in *Proceedings of the 2007 TeraGrid Conference*, 2007.

[21] 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.

[22] 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.

[23] M. J. Rashti, G. Sabin, and R. Kettimuthu, "Long-haul secure data transfer using hardware-assisted gridftp," *Future Generation Computer Systems*, vol. 56, pp. 265–276, 2016.

[24] "Expanse," https://www.sdsc.edu/services/hpc/expanse/, 2021.

[25] "Stampede2," https://www.tacc.utexas.edu/systems/stampede2, 2021.

[26] I. Alan, E. Arslan, and T. Kosar, "Energy-aware data transfer algorithms," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015, pp. 1–12.

[27] M. S. Z. Nine and T. Kosar, "A two-phase dynamic throughput optimization model for big data transfers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 2, pp. 269–280, 2020.

[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] "Bandwidth-delay product," https://en.wikipedia.org/wiki/Bandwidth-delay_product, 2021.

[30] 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.

[31] H. Sapkota, M. Arifuzzaman, and E. Arslan, "Sample transfer optimization with adaptive deep neural network," in *2019 IEEE/ACM Innovating the Network for Data-Intensive Science (INDIS)*. IEEE, 2019, pp. 69–76.

[32] H. Sapkota, B. A. Pehlivan, and E. Arslan, "Time series analysis for efficient sample transfers," in *Proceedings of the ACM Workshop on Systems and Network Telemetry and Analytics*, 2019, pp. 11–18.

[33] D. Ghosh and A. Vogt, "Outliers: An evaluation of methodologies," in *Joint statistical meetings*, vol. 2012, 2012.

[34] "Sequence read archive," https://www.ncbi.nlm.nih.gov/sra.

[35] 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.

[36] "Extreme Science and Engineering Discovery Environment," http://www.xsede.org/, 2020.

[37] "Open science grid," https://opensciencegrid.org/, 2021.

[38] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny, "Pegasus: Mapping scientific workflows onto the grid," in *European Across Grids Conference*. Springer, 2004, pp. 11–20.

[39] B. Charyyev, A. Alhussen, H. Sapkota, E. Pouyoul, M. H. Gunes, and E. Arslan, "Towards securing data transfers against silent data corruption." in *CCGRID*, 2019, pp. 262–271.

[40] 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.

Hemanta Sapkota received MS degree in Computer Science and Engineering at the University of Nevada, Reno in 2021 and received his BS degree in Software Engineering from Fudan University in 2016. His research focus is modeling, optimization, and anomaly detection for high-speed data transfers.

Engin Arslan has been an Assistant Professor at the Department of Computer Science and Engineering at the University of Nevada, Reno since 2017. He received his Ph.D. from University at Buffalo in 2016 and worked at National Science for Supercomputing Applications (NCSA) as a postdoctoral research associate for one year. His research interests include high-performance computing and networking, edge and cloud computing, computer networks, distributed systems, and file systems. He is the recipient of several prestigious awards including NSF CRII in 2019 and NSF CAREER in 2021.