Toward Efficient Data Prefetching Algorithms for Scientific Domains

Reza Nasirigerdeh
University of California, Santa Cruz

Carlos Maltzahn
University of California, Santa Cruz

Frank Würthwein
University of California, San Diego

Brian Bockelman
University of Nebraska-Lincoln

Matevz Tadel
University of California, San Diego

Michael A. Sevilla
University of California, Santa Cruz

Abstract

Recent increase in computing capability of commercial clouds has motivated scientific communities to transition their applications and infrastructures to the cloud. Data prefetching plays a crucial role in this new cloud-based infrastructure in which client applications access large volumes of scientific data resident at the remote storage system. Because each scientific domain has its own data format and its own client applications, the common prefetching algorithms such as those based on byte streams do not work efficiently for every scientific domain. In this paper, we argue that efficient data prefetching algorithms for specialized scientific use cases can be designed using domain-specific knowledge. We propose a novel prefetching algorithm for a particular scientific domain, which leverages domain-specific knowledge relating to the logical layout and the transfer unit of data. We illustrate the efficiency of our algorithm via simulation and accuracy and recall metrics.

1 Introduction

Scientific applications involve large scale mathematical and numerical calculations to solve scientific problems [1]. These applications need huge amount of compute, storage, and networking resources to perform the required calculations [2]. The resource requirements of scientific applications typically addressed with dedicated high throughput computing systems which are geographically distributed over different administrative domains [2].

Recently, there is an increasing interest among commercial cloud providers to show they can meet the resource requirements of scientific applications [2]. There is also a growing interest among scientific communities to leverage cloud computing to take advantage of common benefits of cloud environments such as scaling up or scaling down the resources based on the requirements of the applications, easy access to large scale distributed infrastructures, and no need for capacity planning [3, 1].

In both dedicated and cloud environments, data prefetching plays a crucial role for the performance and cost of the scientific applications running on the client which access large amount of data resident at the remote storage servers [4]. The data prefetching algorithms might work based on byte streams or logical units comprising data. The former algorithms either prefetch whole file (full-file prefetching) or predict the byte ranges that will be requested by the application in the future based on the byte ranges requested in the past. The latter algorithms are aware of the logical layout of data, and as a result, they predict the logical data units required by the application later based on the pattern of accesses to the logical data units so far.

Byte stream based prefetching algorithms are ineffecient in most cases. The exception is that whole file is needed to be processed by the application which occurs very rare. On the other hand, prefetching algorithms which are aware of logical data units can be efficient provided that they enriched with domain specific knowledge such as the organization of the logical data units in the file, the pattern of accesses to those data units, and etc.

Considering the fact that each scientific domain has its own data format and client applications, we think that new efficient prefetching algorithms should be designed specifically for each scientific domain, taking into the consideration this domain specific knowledge. In this paper, we focus on one such scientific domain, High Energy Physics (HEP). We propose a novel and efficient prefetching algorithm which works based on basket, logical data units compromising ROOT [5] files leveraged by HEP community. Although the proposed prefetching algorithm has been designed for HEP, the concepts and methodology presented in the paper is general enough to be applied to other scientific domains.

The contributions of this paper are as follows :

- We illustrate that byte stream based prefetching algorithms such as full-file prefetching, currently used by HEP community, does not work efficiently.
- We argue that efficient prefetching algorithms can be designed for scientific domains by taking into account the domain specific knowledge.
- We propose a novel prefetching algorithm for HEP based on baskets, logical data units of ROOT files, which leverages the HEP specific domain knowledge to prefetch data.

The rest of this paper is organized as follows: Section 2 provides a background on HEP datasets, ROOT file format, and AAA global data federation. Section 3 describes the problem addressed by the paper and the motivation behind our work. Section 4 reviews the related work in the area of data prefetching. Section 5 describes our prefetching algorithm and section 6 evaluates the efficiency of our algorithm by discussing the simulation results. Section 7 concludes the paper with a brief conclusion.

2 Background

The Compact Muon Solenoid (CMS) collaboration [6] at the Large Hadron Collider (LHC) [7] depends today on a global computing infrastructure of several hundred petabytes of disk accessed by several hundred thousands CPU cores. Thousands of scientists in the collaboration derive physics results from the data. Physics data is organized into *datasets*. Each dataset is comprised of a collection of *events*, in which each event is the representation of a proton-proton collision at LHC. Each event in the dataset has a set of characteristics associated with it. HEP community employs ROOT [5] file format to store the datasets in the underlying storage system. The *logical structure* of a ROOT file consists of a set of *trees*, each tree has a set of *branches*, and each branch has an ordered set of *entries* grouped into an ordered set of *baskets*.

A tree in a ROOT file is analogous to a table. Each branch of the tree is akin to a column of the table and an entry of the branch is similar to a value of the column. Notice that a branch entry is not necessarily a scalar value. It can be an array of values (e.g. array of integers or array of array of integers). Therefore, to be more precise, we can say that a tree is similar to a non-relational table in which a column of the table can be another table. Basket is the unit of data that is stored into or retrieved from the ROOT file. An ordered set of branch entries are grouped together into a basket. The basket is first compressed and then written into the file [8].

Figure 1: Physical structure of a ROOT file [8]. Baskets contain data and trees include only metadata.

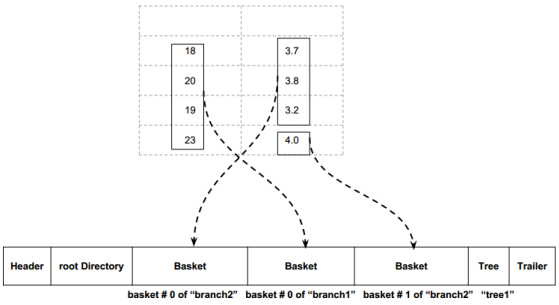

Figure 2: Storing the data of a simple table into a tree in a ROOT file.

The *physical structure* of a ROOT file consists of a *header*, *root directory*, set of baskets, set of trees, and a *trailer* (Figure 1). Header includes metadata describing whole ROOT file such as the size of the file and trailer is used to append data to the existing file. root directory is similar to "/" directory in UNIX and includes all the baskets and trees of the ROOT file. Baskets contain the actual data (branch entries) and trees only include the metadata such as their branches, total number of entries, and etc. Additionally, baskets also contain some metadata including the branch they belong to, range of branch entries they contain, and etc. Notice that although branches are not shown in the physical structure of ROOT files, they are written to the file as part of the tree. Moreover, the baskets of the same branch are not necessarily consecutive in the file and they can have different sizes.

To make the concepts of tree, branch, entry, and basket more clear, consider a simple table with four rows and two columns (Figure 2). The table is mapped into a tree "tree1" with two branches ("branch1" and "branch2") in the ROOT file. Entries 0^{th} to 3^{th} of the first branch (all values of the first column) are grouped into one basket (*basket*#0 of "branch1"). Similarly, the entries 0^{th} to 2^{th} and entry 3^{th} of branch2 inserted into first (*basket*#0 of "branch2") and second basket of branch2 (*basket*#1 of "branch2"), respectively. The number of entries inside a basket depends on the size of the branch entries and the maximum size of the baskets associated with the branch, which is defined by the user.

HEP datasets, which are set of events, can be considered as a table in which each row is an event and each column is a characteristic of the event. Because the size of a HEP dataset is very large, it stored in a set of ROOT files (Figure 3). That is, each ROOT file contains a portion

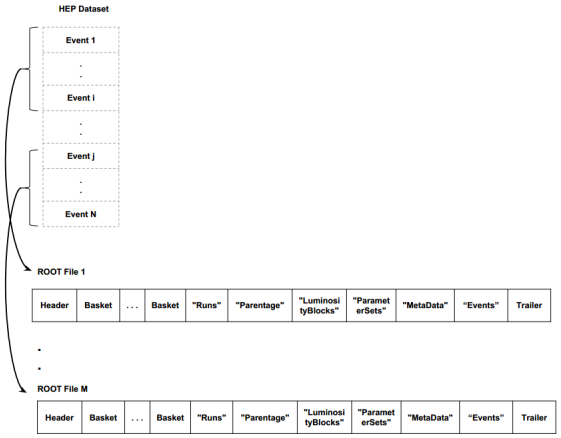

Figure 3: A HEP dataset stored in a set of ROOT files; all files have the same set of trees and each tree has the same branches in all files; that is, all ROOT files have the same structure. The events are stored in the baskets of *Events* tree.

of the dataset (subset of events comprising the dataset). The dataset is stored in the baskets of a tree called *Events* tree. Each branch of the *Events* tree is a characteristic of the event. The ROOT files also contain other trees such as *MetaData* whose baskets contain metadata about the dataset. The *Events* tree is large (several gigabytes) but the other trees are small (several megabytes). All the ROOT files associated with a dataset have the same trees, and each tree has the same branches. In other words, all the ROOT files storing a given dataset have the same structure. On the other hand, they store different subset of events. For instance, the first ROOT file might contain the first one million events of the dataset, the second ROOT file might include the second one million events of the dataset and so on and so forth.

In a typical HEP analysis workflow, an individual scientist's custom executable is applied to the dataset. The purpose of the processing is to filter out interesting events, to calculate physics quantities of interest for these filtered events, and to save the results into a smaller dataset for further study. The CMS Remote Analysis Builder (CRAB) [9] creates multiple instances of the custom executable and assigns a portion of the dataset to an instance for processing. We refer to these instances as *jobs*, and the collection of all instances required to process a dataset as a *task*. A task thus consists of many jobs, each of which processes a different portion of the dataset. Jobs can run in different clients.

All the physics data is organized into a global data federation [10], know as AAA (Any Data, Any Time, Anywhere) data federation, using the XRootd [11] software and protocol. AAA data federation is comprised of a set

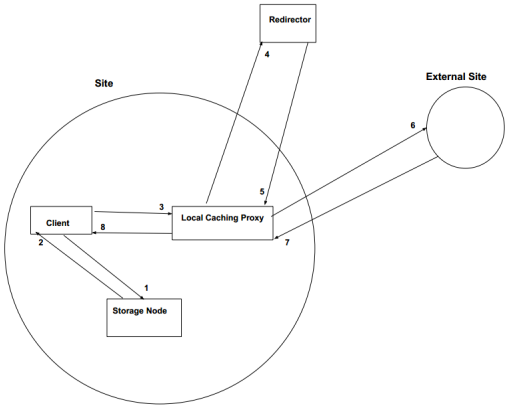

Figure 4: Procedure to retrieve a ROOT file by the client

of *sites* (independent administrative domains) which are coordinated by a set of *redirectors*. In each site, there are a collection of clients and storage nodes as well as a local caching proxy. The client running the job follows the following steps to retrieve the ROOT file (portion of the dataset) for processing (Figure 4):

1. The client requests to access the ROOT file.
2. If the ROOT file is located at a local storage node (i.e. a storage node at the same site the client is running), it is given to the client.
3. Otherwise, the client contacts the local caching proxy at its site.
4. The local proxy contacts the redirector to retrieve the file from the other sites.
5. The redirector finds the site hosting the file and gives the information needed to retrieve the file to the local proxy.
6. The local proxy contacts the external site.
7. The local proxy retrieves the file from the external site.
8. The client gets the ROOT file from the local proxy.

The prefetching algorithm runs in the local caching proxy. If the prefetching algorithm is full-file prefetching, the local proxy prefetches the whole ROOT file from the external site and gives it to the client which has requested the file. If our prefetching algorithm is employed in the local proxy, it prefetches a set of baskets of the ROOT file that are highly likely requested by the client in the future.

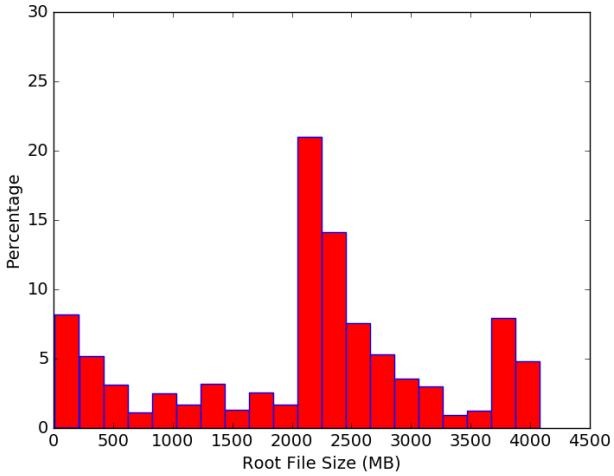

Figure 5: Size of Root Files

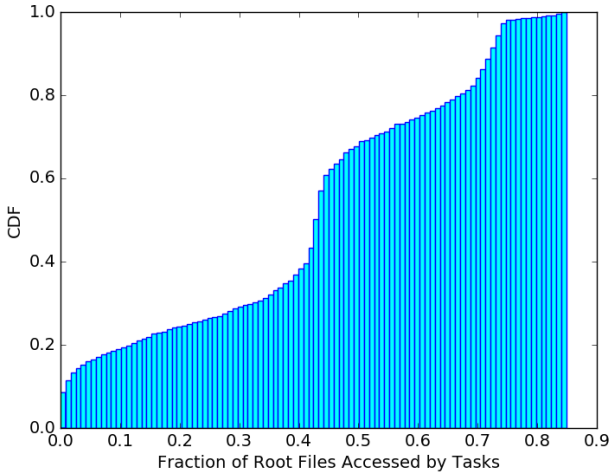

Figure 6: CDF of the fraction of the size of the ROOT files accessed by jobs

3 Motivation

CMS currently employs a full-file prefetching method which is not aware of baskets (logical data units of root files). In this strategy, if a client requests a basket of a file, the caching proxy starts prefetching the whole file from the data federation and makes it available to the client. The size of the ROOT files can be quite large. As seen in Figure 5, a histogram of the file size for a common dataset (W2JetsToLNu) in CMS, the median file size is greater than 2GB.

Moreover, different tasks only need to access a fraction of the data in a file; Figure 6 illustrates Cumulative Distribution Function (CDF) of the fraction of the size of files accessed by jobs (Only unique bytes requested by the client are considered in the calculations). For about 98 percent of the root files, the tasks need to access less than 70 percent of the root file size. It means that for almost all files in this trace, the caching proxy would prefetch an extra 30 percent that would not be needed by the jobs processing those root files.

Finally, as a physicist wants to examine all events logically grouped in a dataset, task commonly accesses a large percentage of files in the dataset. The client-side prefetching strategies are only done within a single job, yet the access pattern is common across the entire task: only the storage layer sees across the multiple jobs of a task. As an example, within the CMS-provided trace, one task was composed of 142 different jobs that accesses a total of 283 different files from the W2JetsToLNu dataset (out of a total of 303 files).

Putting together all three observations that the size of root files is quite large, jobs only need to access a fraction (less than 70 percent) of the size of the root files not the whole, and jobs commonly require to access a large number of root files storing the dataset, we can see how inefficient the full-file prefetching strategy is. To indicate this inefficiency in terms of numbers, consider the example of the task we mentioned previously which accesses 283 root files. If the size of each root file is about 2GB (Figure 5) and only 70 percent of the root file size is required to be accessed by the task (Figure 6), then total of $283 \times 0.3 \times 2GB = 169.8GB$ of unnecessary data is prefetched by the caching proxy.

This large volume of unnecessarily prefetched data is very costly for both underlying storage system and communication network. It wastes lots of CPU cycles, and I/O (network and disk) bandwidth. Therefore, we need much more efficient prefetching algorithms than blind full-file prefetching for applications. These algorithms should essentially be aware of logical units of data requested by the application rather than byte streams. This way, they can predict the data units that will be requested by the application in the future based on the pattern of the accesses to the data units in the past. Our proposed prefetching algorithm is such kind of algorithm. It works based on basket, which is the logical data unit requested by a task running in a client.

4 Related Work

In this paper we are providing evidence that the full performance benefit of prefetching data might not be available unless prefetching policies are expressed in domain-specific abstractions and the data can be (remotely) ac-

cessed using these abstractions. Scientific communities are using world-wide data grids [12] that are storing data in file systems and rely on prefetching entire files [13]. Prefetching policies within file systems [14] are also based on byte stream abstractions only. Similarly, prefetching in the World-wide Web [15] is based on files and their context in web pages and websites. Content distribution networks (CDNs) (see for example [16]) are partitioning large files into byte-level chunks to alleviate main-memory overhead and reduce network congestion [17]. On the other hand, prefetching strategies in language runtimes are data-structure aware and therefore can rely on the semantics of the data [18]. With the advent of programmable storage systems [19] there is now an opportunity to overcome the semantic gap imposed for so many years by byte-stream abstractions and to express prefetching policies in terms of the meanings of data.

5 Proposed Prefetching Algorithm

We propose a novel prefetching algorithm which is more efficient than blind full-file prefetching strategy. Our algorithm is motivated by the following observations:

1. Each basket of a branch contains a subset of the entries of that branch.
2. All the ROOT files in a given dataset have the same structure; that is, they all have the same trees and branches but with different data (baskets).
3. It is common for a task (i.e. different jobs of the task) to access a large number of ROOT files in the dataset (sometimes all files in the dataset), and as a result, very large volume of data.
4. If the application requests a basket of a branch, it is highly likely that it will request the other baskets of the same branch.
5. The size of *Events* tree is much larger than the size of the other trees in the ROOT file.

The algorithm only considers *Events* tree which contains the dataset and its size is much larger than the other trees (fifth observation); it first creates a $N \times K$ matrix, called $prefetch\ matrix$ (Figure 7). In the matrix, each row indicates an event number and each column represents a branch of the *Events* tree. N is the total number of events in the dataset and K is the total number of branches of the *Events* tree. The value of each element of the matrix is either zero or one and they are all initially zero. If the application requests a basket of branch j and the basket contains the event numbers p through q, then the algorithm sets the elements (p, j), $(p+1, j)$, ..., (q, j) to one (This is based on the first observation that each basket of a branch touches a set of events). The algorithm continues getting the baskets and filling the prefetch matrix until it reaches the pre-determined value of T baskets. We refer to T as the number of trains baskets used to fill the prefetch matrix.

	Branch 1	....	Branch j	....	Branch K
Event 1	0	0	0	0	0
.	0	0	0	0	0
Event p	0	0	1	0	0
.	0	0	1	0	0
.	0	0	1	0	0
Event q	0	0	1	0	0
.	0	0	0	0	0
Event N	0	0	0	0	0
Weight	Weight 1	...	Weight j	...	Weight K

Figure 7: Prefetch Matrix

Afterwards, it computes the sum of the elements for each branch (column of the matrix). We refer to this sum as the weight of the branch. Higher weights means more number of events touched. Finally, the algorithm sorts the branches based on their weight in decreasing order and prefetches all the baskets of the top Q branches (second and forth observations).

6 Evaluation

We leveraged simulation to evaluate the efficiency of the proposed prefetching algorithm. We chose the trace of accesses from three different tasks over three different datasets. The accesses in the trace files had the form $(start\ offset, size, file\ name)$, indicating that the job requested the byte range from $start\ offset$ to $(start\ offset\ +\ size)$ of the ROOT file $file\ name$. We employed RootUtils [20] library to translate the requested byte ranges to the corresponding baskets. The tasks were running during the time period 01/01/2017 through 02/01/2017 at UCSD site. Table 1 lists the number of jobs of each task, and the number of accesses of each task to the corresponding dataset. Table 2 lists the dataset name and the total number of ROOT files in the datasets. Notice that different jobs of the same task might access to the same ROOT file in the dataset because they are running in different client machines. Therefore, the number of accesses of a task to the dataset can be greater than the number of the ROOT files in the dataset (e.g. task2 over dataset2).

We used the baskets of the first 4 ROOT files accessed by the task to fill the prefetch matrix. We evaluate the ef-

Task	# of Jobs	# of Accesses	Dataset
Task1	24	70	Dataset1
Task2	142	331	Dataset2
Task3	17	143	Dataset3

Table 1: Tasks used in the simulation

Dataset	Name	# of ROOT Files
Dataset1	*W2JetsToLNu*	431
Dataset2	*tZq_ll*	303
Dataset3	*QCD_Pt*	163

Table 2: Datasets leveraged in the simulation

ficiency of the algorithm using the well-known measures of *accuracy* and *recall*, and in terms of both basket and byte. The basket is the unit of access to the underlying storage system and byte is the unit of data transferred in the communication network.

The accuracy is defined as $\frac{TP+TN}{TP+TN+FP+FN}$. This measure shows that how much the prefetching algorithm is efficient in prefetching baskets/bytes that should be prefetched and not prefetching baskets/bytes that should not be prefetched. True Positives (TP) is the number of baskets/bytes correctly prefetched (Job needed them and they were prefetched). False Positives (FP) is the number of baskets/bytes wrongly prefetched (Job did not need them but they were prefetched). True Negatives is the number of baskets/bytes correctly not prefetched (Job did not require them and they were not prefetched). False Negatives (FN) is the number of baskets/bytes wrongly not prefetched (Job needed them but they were not prefetched). The recall is defined as $\frac{TP}{TP+FN}$. This metric shows that how much our algorithm is efficient in prefetching only the baskets/bytes that should be prefetched (coverage of the baskets/bytes that needed to be prefetched).

In the following subsections, we present the results of accuracy and recall for task3. We do not discuss the results of these metrics for the other tasks because they are very similar to task3. Moreover, we present the results of the metrics for each file accessed by the task.

6.1 Accuracy

Figure 8 illustrates the byte accuracy (accuracy in terms of byte) of full-file prefetching strategy and the proposed algorithm for different values of M. M is the percentage of the branches prefetched by the proposed algorithm. That is, if the number of prefetched branches is Q and the total number of branches is K, then $M = \frac{Q}{T} \times 100$. The full-file prefetching is equivalent to $M = 100$ in which all branches are prefetched. Notice that the accuracy

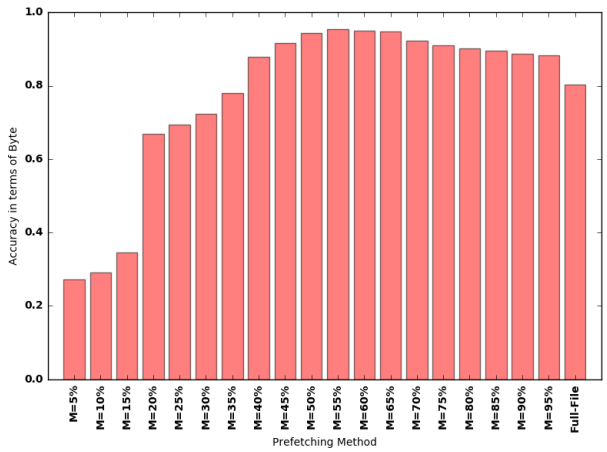

Figure 8: Accuracy in terms of Byte for Task3

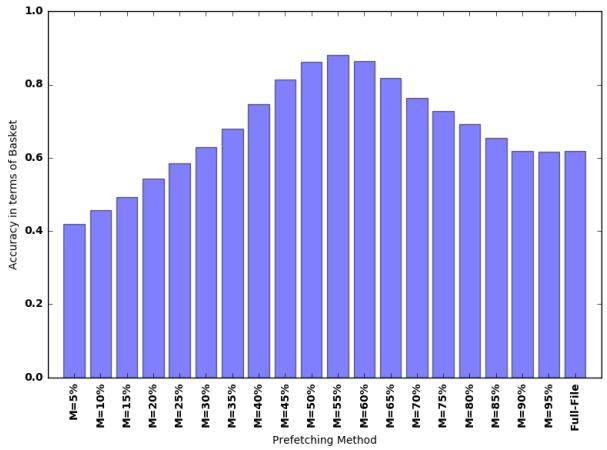

Figure 9: Accuracy in terms of Basket for Task3

shown in the figure is calculated considering all the files accessed by the task.

According to the figure, the byte accuracy increases as the value of M increases from 5% to an optimal value (55%). Further increase of M (from optimal value to 95) results in decrease of the byte accuracy because it reduces the True Negatives (it prefetches bytes not needed by the jobs). Moreover, the byte accuracy of the proposed algorithm for optimal value of $M = 55\%$ is higher than that for the full-file prefetching strategy.

Figure 9 plots the basket accuracy (accuracy in terms of basket) for different values of M. The results of basket accuracy are akin to the results of byte accuracy. The basket accuracy of the proposed algorithm for the optimal $M = 55\%$ is much better than basket accuracy of the full-file prefetching. Comparing the basket and byte accuracy results shows that best byte and basket accuracy

TP=208.804GB	FP=9.865GB	TP=28.256 Million	FP=2.225 Million
FN=2.089GB	TN=41.55GB	FN=3.953 Million	TN=17.612 Million

(a) The proposed algorithm

TP=210.893GB	FP=51.418GB	TP=32.21 Million	FP=19.836 Million
FN=0	TN=0	FN=0	TN=0

(b) Full-file prefetching

Figure 10: Confusion matrix for the proposed algorithm with optimal $M = 55\%$ (top) and full-file prefetching (bottom) in terms of Gigabytes (left) and million baskets (right).

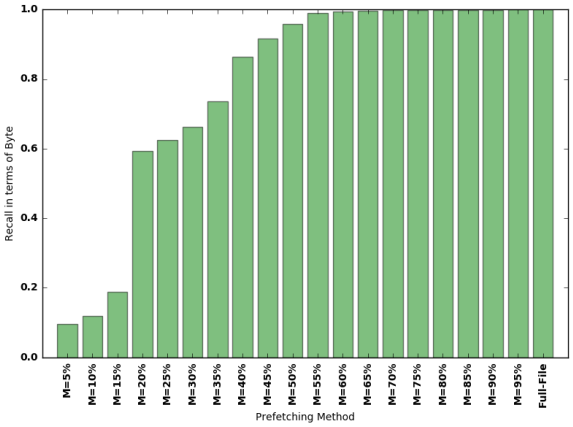

Figure 11: Recall in terms of Byte for Task3

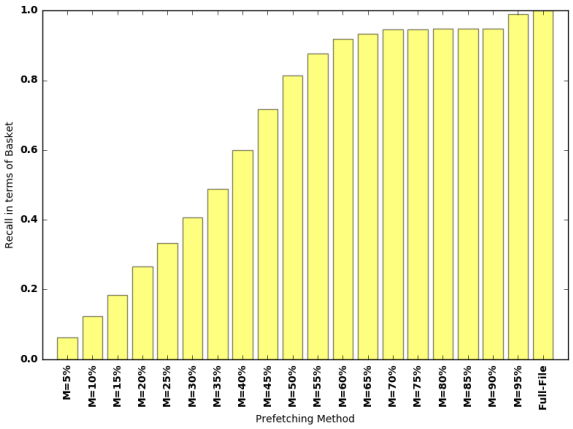

Figure 12: Recall in terms of Basket for Task3

are obtained from the same optimal value of M (50%) and basket accuracy is lower than byte accuracy for both full-file prefetching and our algorithm.

Figures 10a and 10b show the confusion matrix for the proposed algorithm with optimal value $M = 55\%$ and full-file prefetching, respectively. Full-file prefetching strategy prefetches about 51 Gigabytes and 20 million baskets that is not required by the task, causing a great deal of bandwidth wastage for both network and disk. On the other hand, these values is much lower for our algorithm (around 10 Gigabyte and 2 million baskets), indicating that our algorithm is much more efficient than the full-file prefetching.

6.2 Recall

Unlike accuracy metric which indicates how much a prefetching algorithm is efficient in both prefetching required baskets/bytes and not prefetching unnecessary baskets/bytes, the recall measure is indicator of how much the algorithm is efficient in only prefetching the required baskets/bytes. Figure 11 plots the recall of full-file prefetching and the proposed algorithm for different values of M. Full-file prefetching has the best recall, which is 1. This is because it prefetches whole file, which means it also prefetches whole baskets/bytes required by the task. Moreover, the recall increases as the value of M increases. For the value of $M = 55\%$, which our algorithm has the optimal accuracy, the byte recall value is 0.98, indicating that the proposed algorithm prefetches the most bytes required by the task.

Figure 12 illustrates the results of basket recall (recall in terms of basket) for different values of M. The results are similar to those from byte recall. The basket recall of full-file prefetching is the best and the basket recall of the proposed algorithm increases as the value of M increases. Basket recall for the $M = 55\%$, which the algorithm has the best accuracy is about 85%.

6.3 Accuracy and Recall per File

In the previous subsections, we presented the results of accuracy and recall per task. That is, we considered the total files accessed by the task to compute the accuracy and recall metrics for the task. Here, we present the results per each file accessed by the task. To this end, we consider Task3 and the value of $M = 55\%$ for which the algorithm had the best byte/basket accuracy. Figures 13 and 14 show the byte/basket accuracy of the algorithm for each file accessed by the task. According to the figures, the byte and basket accuracy for 134 files out of 139 files are greater than 0.9, and 0.8, respectively. Figures 15 and 16 indicate the byte/basket recall of the algorithm per each file. As shown in Figure 15, the byte and basket

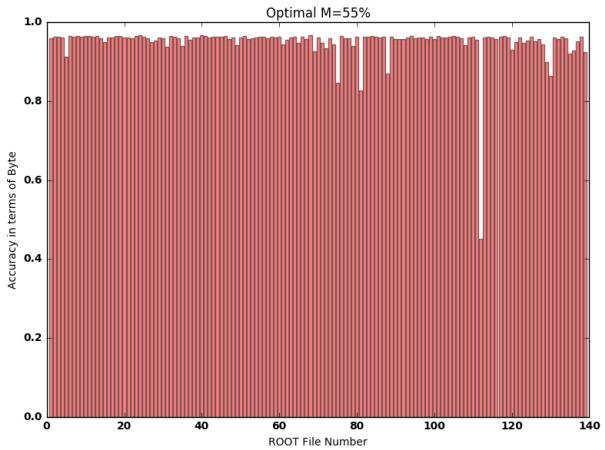

Figure 13: Byte accuracy per each file accessed by Task3

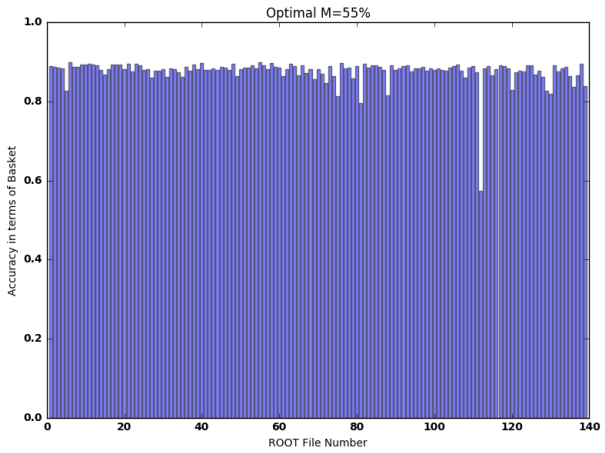

Figure 14: Basket accuracy per each file accessed by Task3

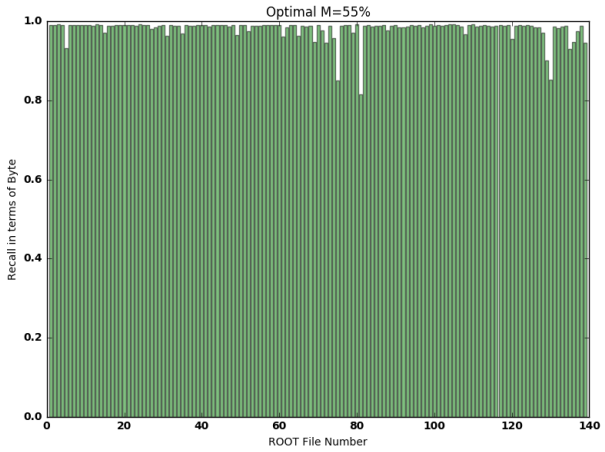

Figure 15: Byte recall per each file accessed by Task3

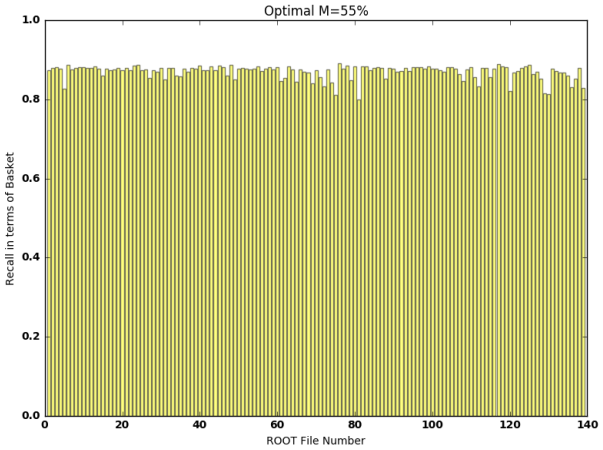

Figure 16: Basket recall per each file accessed by Task3

recall for 135 files out of 139 files is greater than 0.95 and 0.8, respectively. These results of accuracy and recall per each file indicate that the proposed algorithm also works efficiently for most of the files accessed by the task.

7 Conclusion

In this paper, we showed that common byte stream based prefetching algorithms (e.g. full-file prefetching) do not work efficiently for scientific domains such as High Energy Physics. Moreover, we argued that we can use scientific domain specific knowledge (e.g. logical data units comprising the files, their access pattern and etc) to design much more efficient prefetching algorithms for these domains. We proposed a novel prefetching algorithm for High Energy Physics based on the basket, the logical data unit in ROOT files. Through a set of simulation results, we show that the proposed algorithm is much more efficient than blind full-file prefetching.

In the future works, we can apply the concepts presented in the paper to different scientific domains including Genomics and Weather Forecasting to design efficient prefetching algorithms for these domains considering their domain specific knowledge. Moreover, we can have real implementation of our prefetching algorithm at the caching proxy to see whether it also works efficiently in the real system.

Acknowledgments
This work is supported by National Science Foundation under award 1705021.

References

[1] C. Vecchiola, S. Pandey, and R. Buyya, "High-performance cloud computing: A view of scientific applications," in *Pervasive Systems, Algorithms, and Networks (ISPAN), 2009 10th International Symposium on*, pp. 4–16, IEEE, 2009.

[2] B. Holzman, L. A. T. Bauerdick, B. Bockelman, D. Dykstra, I. Fisk, S. Fuess, G. Garzoglio, M. Girone, O. Gutsche, D. Hufnagel, H. Kim, R. Kennedy, N. Magini, D. Mason, P. Spentzouris, A. Tiradani, S. Timm, and E. W. Vaandering, "Hepcloud, a new paradigm for hep facilities: Cms amazon web services investigation," *Computing and Software for Big Science*, vol. 1, p. 1, Sep 2017.

[3] G. Lin, B. Han, J. Yin, and I. Gorton, "Exploring cloud computing for large-scale scientific applications," in *Services (SERVICES), 2013 IEEE Ninth World Congress on*, pp. 37–43, IEEE, 2013.

[4] Z. Zhang, A. Kulkarni, X. Ma, and Y. Zhou, "Memory resource allocation for file system prefetching: From a supply chain management perspective," in *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, (New York, NY, USA), pp. 75–88, ACM, 2009.

[5] R. Brun and F. Rademakers, "ROOT: An object oriented data analysis framework," *Nucl. Instrum. Meth.*, vol. A389, pp. 81–86, 1997.

[6] S. Chatrchyan *et al.*, "The CMS Experiment at the CERN LHC," *JINST*, vol. 3, p. S08004, 2008.

[7] L. Evans and P. Bryant, "LHC Machine," *JINST*, vol. 3, p. S08001, 2008.

[8] R. NasiriGerdeh, J. Pivarski, M. A. Sevilla, and C. Maltzahn, "Root files for computer scientists," Tech. Rep. UCSC-SOE-18-09, UC Santa Cruz, May 2018.

[9] D. Spiga, S. Lacaprara, W. Bacchi, M. Cinquilli, G. Codispoti, M. Corvo, A. Dorigo, A. Fanfani, F. Fanzago, F. Farina, M. Merlo, O. Gutsche, L. Servoli, and C. Kavka, "The cms remote analysis builder (crab)," in *High Performance Computing – HiPC 2007*, (Berlin, Heidelberg), pp. 580–586, Springer Berlin Heidelberg, 2007.

[10] K. Bloom, T. Boccali, B. Bockelman, D. Bradley, S. Dasu, J. Dost, F. Fanzago, I. Sfiligoi, A. M. Tadel, M. Tadel, C. Vuosalo, F. Wrthwein, A. Yagil, and M. Zvada, "Any data, any time, anywhere: Global data access for science," in *2015 IEEE/ACM 2nd International Symposium on Big Data Computing (BDC)*, pp. 85–91, Dec 2015.

[11] A. Dorigo, P. Elmer, F. Furano, and A. Hanushevsky, "Xrootd - a highly scalable architecture for data access," vol. 4, pp. 348–353, 04 2005.

[12] D. Thain, J. Basney, S.-C. Son, and M. Livny, "The kangaroo approach to data movement on the grid," in *HPDC '01*, 2001.

[13] S. Venugopal, R. Buyya, and K. Ramamohanarao, "A taxonomy of data grids for distributed data sharing, management, and processing," *ACM Comput. Surv.*, vol. 38, no. 1, 2006.

[14] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," 1995.

[15] V. N. Padmanabhan and J. C. Mogul, "Using predictive prefetching to improve world wide web latency," *SIGCOMM Comp. Comm. Review*, vol. 26, no. 3, pp. 22–36, 1996.

[16] K. Park and V. S. Pai, "Scale and performance in the CoBlitz large-file distribution service," 2006.

[17] M. Crovella and P. Barford, "The network effects of prefetching," in *IEEE Infocom '98*, (San Francisco, CA), 1998.

[18] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *ASPLOS 1992*, October.

[19] M. A. Sevilla, N. Watkins, I. Jimenez, P. Alvaro, S. Finkelstein, J. LeFevre, and C. Maltzahn, "Malacology: A programmable storage system," in *EuroSys '17*, (Belgrade, Serbia), April 23-26 2017.

[20] R. NasiriGerdeh, J. Pivarski, M. Tadel, M. A. Sevilla, C. Maltzahn, B. Bockelman, and F. Wurthwein, "Rootutils: a library to better understand root files," Tech. Rep. UCSC-SOE-18-10, UC Santa Cruz, May 2018.