Efficient Data Selective Scheme for Parallel and Distributed Environment

Prof. S. Padmavathi[1] Thomas Abraham[2]
[1]Head of Dept. [2]M. Phil. Student
[1,2]Department of Information Technology
[1,2]Marudupandiyar College, Thanjavur, India

Abstract— Now a Days, many scientific applications spend a significant portion of their execution time in accessing data from files. Various optimization techniques exist to improve data access performance, such as data perfecting and data layout optimization. However, optimization process is usually a difficult task due to the complexity involved in understanding I/O behaviour. Tools that can help simplify the optimization process have a significant importance. In this paper, we introduce a tool, called IOSIG, for providing a better understanding of parallel I/O accesses and information to be used for optimization techniques. The tool enables tracing parallel I/O calls of an application and analysing the collected information to provide a clear understanding of I/O behaviour of the application. We show that performance overheads of the tool in trace collection and analysis are negligible. The analysis step creates I/O signatures that various optimizations can use for improving I/O performance. Input and output signatures are compact, easy-to-understand, and parameterized representations containing data access pattern information such as size, strides between consecutive accesses, repetition, timing, etc. The signatures include local I/O behaviour for each process and global behaviour for an overall application. We illustrate the usage of the IOSIG tool in data perfecting and data layout optimizations.

Key words: Optimization Techniques, Parallel I/O

I. INTRODUCTION

Parallel file systems (PFS) have been widely used in high-performance computing (HPC) systems during the past few decades. A PFS, such as OrangeFS [1], Lustre [2] and GPFS [3], can achieve superior I/O bandwidth and large storage capacity by accessing multiple file servers simultaneously. However because of the existing performance gap between file servers and CPU, the so called I/O wall, current PFSs cannot fully meet the growing data access requirements of many HPC applications [4], especially for data intensive HPC applications.

NAND flash based solid state disks (SSD) are attracting attention in HPC domains [5]. An SSD is a purely electronic device without mechanical components, thus can provide low access latency, high data bandwidth, lower power consumption, lack of noise, and shock resistance. However, due to the high cost of SSDs and the inherent merits of HDDs (high capacity and decent peak bandwidth for sequential requests), building a large file system solely based on SSDs may be unfeasible for most systems. Therefore, a hybrid PFS, which is comprised of both HDD servers (HServer) and SSD servers (SServer), provides a promising solution for data-intensive applications [6], [7].

A high-performance hybrid PFS must rely on an efficient data layout, which is an algorithm defining a file's data distribution across available servers. To achieve even data placement, most traditional layout methods utilize a fixed-size stripe to dispatch data across multiple servers. There are three typical such schemes: the one dimensional horizontal layout, one-dimensional vertical layout, and two-dimensional layout [8], as shown in Fig. 1. To further improve the storage performance, numerous efforts are devoted to the file data layout optimizations, such as data stripe resizing [9], data replication [10], and data reorganization [11]. However, most current schemes are designed and optimized for homogeneous servers. When applied to hybrid PFSs, such schemes have the following three limitations.

First, the heterogeneity of file servers may significantly decrease the overall system performance. Traditional layout schemes usually distribute the same number of file stripes on each server. However, due to their intrinsic properties, SServers almost always outperform HServers [12]. In this case, SServers are easily left idling while HServers continue to process their requests when they concurrently serve a large file request. This inter-server load imbalance leads to underutilization of system hardware resources.

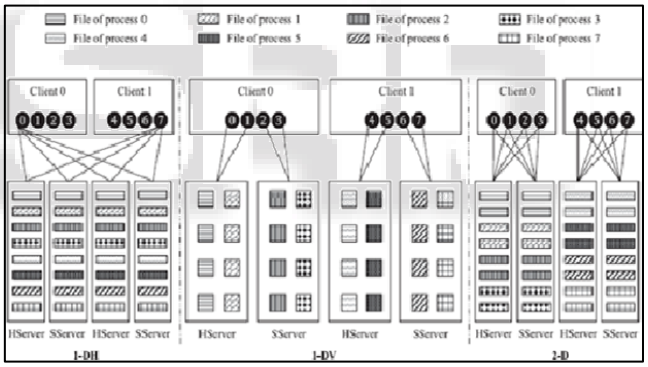


Fig. 1: Three typical data layout polices in PFS.

Second, due to the changes of access patterns across different applications, current layout schemes, designed for a specific set of access patterns, are no longer efficient. For example, the commonly used one-dimensional horizontal layout in OrangeFS [1], is only suitable for large parallel file requests, but performs poorly for small requests with a high degree of access concurrency [8]. As access patterns of different applications may vary, in terms of request size, access type (read or write), and access concurrency, a data layout strategy optimized for an application's access pattern is not efficient for other applications.

Third, as applications become more complex, the access patterns within an application can vary considerably, rendering conventional static layout approaches incapable of adapting to the frequently changed access patterns. For example, the application may have a small number of concurrent I/O requests at one moment, but a burst of I/O requests at another, and the requests in each phase have different sizes.

II. PROBLEM DEFINITION

The goal of memory optimizations is to improve the effectiveness of the memory hierarchy the memory hierarchy typically composed of caches, virtual memory, and the translation look aside buffer (TLB), reduces the memory access time by exploiting the execution's locality of reference.

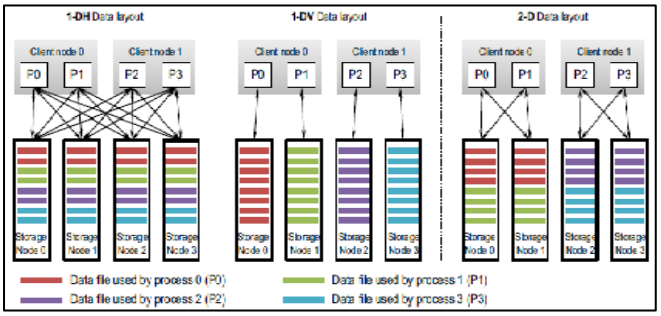


Fig. 2: Three data layouts in parallel system

The opportunity for the optimizer is to help the memory hierarchy by enhancing the program's inherent locality, either temporal or spatial, or both. To alter the temporal locality, one must modify the actual algorithm of the program, which has proved possible for (stylized) scientific code, where transformations such as loop tiling and loop interchange can significantly increase both temporal and spatial locality. However, when the code is too complex to be transformed—a situation common in programs that manipulate pointer-based data structures—one must resort to transforming the layout of data structures, improving spatial locality. Many data-layout optimizations have been proposed with their specific goals ranging from reordering structure fields to object in lining.

III. IMPLEMENTATION

We propose a framework that enables effective and robust data-layout optimizations. Our framework finds a good data layout by searching the space of possible layouts, using profile feedback to guide the search process. A naive approach to profile guided search may transform the program to produce the candidate data layout, then recompile and rerun it. Clearly, such a search cycle is uninformative and too slow (or cumbersome) to be practical.

Indeed, to avoid this tedious cycle our framework evaluates a candidate layout by simulating its fault rate (e.g., cache-miss rate or page-fault rate) on a representative trace of memory accesses. Simulation is more informative than rerunning since it allows us not only to measure the resulting miss rates of the candidate data layouts, but also to identify the objects that are responsible for the poor memory performance. We use this information to narrow the layout search space by determining which levels of memory hierarchy need to be optimized, selecting optimizations that may influence the affected levels of memory hierarchy, and finally by excluding infrequently accessed objects from consideration.

Trace-based fault-rate simulation is not only more informative but it is also faster than editing, recompiling and rerunning the program. To simulate the program without re-compilation, we "pretend" the program was optimized by remapping the original addresses of data objects to reflect the candidate data layout. To make the iterative search even more practical, we speed up the simulation by compressing the trace as a context-free grammar, which in turn allows us to develop algorithms for memorized simulation.

Data-layout optimizations synthesize a layout with good spatial locality generally by attempting to place contemporaneously accessed memory locations in physical proximity while ensuring that frequently accessed memory cells do not evict each other from caches. It turns out that these goals make the problem of finding a "good" layout not only intractable but also poorly approximable. The key practical implication of this hardness result is that it may be difficult to develop data-layout heuristics that are both robust and effective.

A. *Contributions*

We present a framework for data-layout optimization of general- purpose programs that permits composing multiple optimizations. Unifying existing profile-based data-layout optimizations, the framework operates by first heuristically selecting a space of profitable layouts, and then iteratively searching for a good layout, using profile-guided feedback.

We develop techniques for efficiently evaluating a candidate layout optimization using a trace of memory references. These techniques, based on memorization, miss-based trace compaction, and on-demand simulation, make the framework's iterative data layout search efficient enough to be practical. Using the framework, we re-implement two existing data-layout optimizations: Field Reordering and Custom Memory Allocation. The experimental results show that our iterative-search versions outperform the existing single-pass heuristic optimizations.

We compare HAS with two other data layout schemes, the application-aware scheme (ADL) [8] and the storage aware scheme (SDL) [17]. In ADL, file data is placed across the hybrid file servers with one of the three policies according to the application's access pattern, but each server is assigned a fixed-size file stripe. In SDL, the file stripe sizes for the hybrid servers are determined by the server performance but only 1–DH policy is chosen, without fully considering application access patterns. Since ADL and SDL have shown considerable performance improvements over the default data layout scheme namely the fixed-size striping in previous work [8], [17], we do not compare HAS with the default scheme in this paper.

B. *Application-Aware Scheme (ADL)*

I/O data access is a recognized performance bottleneck of high-end computing. Several commercial and research parallel file systems have been developed in recent years to ease the performance bottleneck. These advanced file systems perform well on some applications but may not perform well on others. They have not reached their full potential in mitigating the I/O-wall problem. Data access is application dependent. Based on the application-specific optimization principle, in this study we propose a cost-intelligent data access strategy to improve the performance of parallel file systems.

We first present a novel model to estimate data access cost of different data layout policies. Next, we extend the cost model to calculate the overall I/O cost of any given application and choose an appropriate layout policy for the application. A complex application may consist of different data access patterns. Averaging the data access patterns may

not be the best solution for those complex applications that do not have a dominant pattern.

We further propose a hybrid data replication strategy for those applications, so that a file can have replications with different layout policies for the best performance. Theoretical analysis and experimental testing have been conducted to verify the newly proposed cost-intelligent layout approach. Analytical and experimental results show that the proposed cost model is effective and the application-specific data layout approach achieved up to 74% performance improvement for data-intensive applications.

We cost-intelligent data access strategy to link the interface optimization with the data layout optimization techniques, which is beneficial to various types of I/O workloads. This study makes the following contributions. We propose a cost model of data access for parallel file systems, which can estimate the response time of data accesses for different data layout policies. We present a static data layout optimization based on the overall cost estimation, which can choose an appropriate data layout for applications with specific I/O patterns.

We dynamic data access strategy with hybrid replication for those with mixed I/O workloads in the case that one layout policy cannot benefit all data accesses. We implement the dynamic replication selection in MPI-IO library, which can automatically dispatch data access to one replica with the lowest access cost. Our analytical and experimental results show that the newly proposed cost-intelligent application-specific data layout approach is very promising and has a real potential in exploring the full potential of parallel file systems. Experimental testing is conducted under the MPI program environment and PVFS2 file system. Experimental results show that the accuracy of the cost model in identifying an efficient data layout is in the range of 80~91%, and the hybrid replication strategy can achieve 13~74% performance improvement compared to single fixed layout policy. In summary, the proposed cost-intelligent application-specific data layout optimization is feasible and effective, and should be further investigated to explore the full potential of parallel file systems. The proposed hybrid replication strategy trades the available storage capacity for the precious data access I/O performance.

C. Pattern-Direct and Layout Aware

To achieve the goal of alleviating I/O bottleneck and to satisfy the requirement of the I/O optimization's adaptability, we design and implement the Pattern-Direct and Layout Aware (PDLA) replication scheme for parallel I/O systems. We design PDLA based on the following facts. 1) Contiguous data access is preferable.

This stays true for both hard disk drives (HDD) and solid state disks (SSD) [10]. 2) Data layout matters. Data layout in parallel file systems can largely influence the I/O performance. Modern parallel file systems support multiple data layout policies. Users can choose to distribute some data on one single storage node, on a set of nodes, or on all available nodes. The previous work [8] shows that, for applications with different data access patterns, the optimal data layouts are different. The optimal data layout yields the

IV. SYSTEM ARCHITECTURE

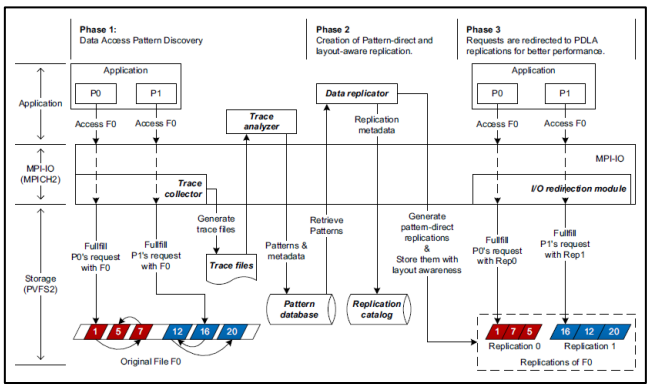


Fig. 3: Data access pattern discovery

A. Trace Collector

The trace collector simply traces MPI-IO calls defined in the MPI standard. It gathers information of all standard MPIIO file operations, such as file open/close, file read/write, and seek. The MPI-IO file operations can be blocking or non-blocking, and collective or non-collective. The trace collector captures the I/O operation parameters by using the Profiling MPI interface (PMPI). The Profiling MPI interface reroutes MPI calls to user-defined instrumentations wrapped around the actual MPI calls. The trace collector is implemented as a library, which can be linked to any application we want to trace. Other than this linking step, there is no need for programmer intervention.

B. Trace Analyser

Trace analyser performs offline trace analysis and utilizes the "template matching" approach to recognize data access patterns from trace files. To some extent, one trace file is a list of file operation records, and each record contains an operation's data access information. The analyser uses a cursor to mark its analysis progress in the trace. It starts from the first record and moves the cursor forward to scan all records until reaching the end. During scanning, the analyser always picks a predefined access pattern as a template, to check whether it matches the records around the cursor.

Once a match is found, the cursor moves forward along with the same pattern in the trace, until the match does not hold. If there is no match for the first template, the analyser switches to other templates and tries again. If the analyser fails to find a match for all templates, it skips the current record, moves the cursor forward, and starts over the matching at the new position. The analyser produces all the local patterns by analysing each trace file.

C. Pattern Database

Both the data replicator and the request redirection module in MPI-IO need to retrieve an application's data access patterns. We keep these metadata in "Pattern Database". It saves the mapping relation between applications and their data access patterns, including: 1) which application a pattern belongs to; 2) which file a pattern depends on; 3) the rank of a process that a pattern belongs to; and 4) which local access pattern is included in a global access pattern. Pattern Database also saves the metadata on the runtime environment of the owner application, including mainly the parameters of the system that will be used to determine the optimal data layout for the generated replication files.

D. Data Replicator

The data replicator is a lightweight daemon program that runs in the background. It monitors a queue that contains all the data access patterns that need to be replicated on. When the trace analyser inserts a new access pattern in to the pattern database it also en-queues the same pattern into this global queue.

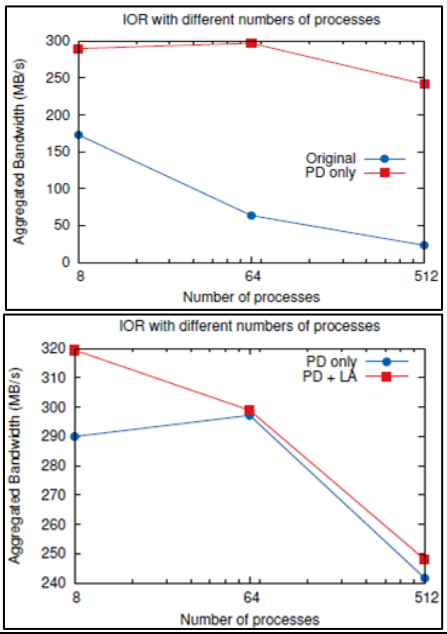
E. I/O Redirection Module in MPI-IO

I/O redirection module redirects data accesses on the original files to the replications. Usually an application issues a data request with three parameters: the identifier of the original file, the data offset, and the request size. After locating the replication file according to these three I/O parameters, runtime information and the metadata in replication catalog, the redirection module translates the filename and offset between original file and the replication and fulfils the request using the replication. The performance of contiguous data access is higher than that of non-contiguous data access.

F. Replication Catalog

The replication catalog is used to store metadata for replications. It manages metadata about the relationships among data replications, original files, and the data access patterns, including which original file a replication's data comes from and based on which access pattern a replication is created. Its implementation also uses Berkeley DB configured as a hash table; the key is the patternID (the same key in Pattern Database) and the value is the path to the replication file based on the corresponding data access pattern. The above in-depth evaluation with IOR already demonstrates the effectiveness of PDLA in improving I/O performance. The overall performance improvement with IOR is 84% to 970%. To convince the IOR testing is representative, we have extended the evaluation to PIO-Bench and MPITile- IO benchmarks.

V. RESULTS



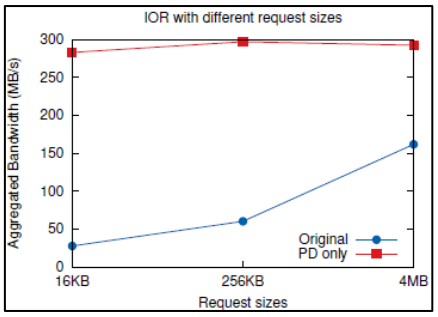


Fig. 3: Results

We run PIO-Bench with a nested-stride access pattern and MPI-Tile-IO with its default access pattern. MPI-Tile- IO treated the entire data file as a 2-D matrix and divides it into $n \times n$ tiles (n rows by n columns). Given $n2$ processes, each process accesses the data in one tile, with fixed-stride access pattern. The data of n tiles in the same row are nested together. Therefore, MPI-Tile-IO's data access pattern is also nested-stride.

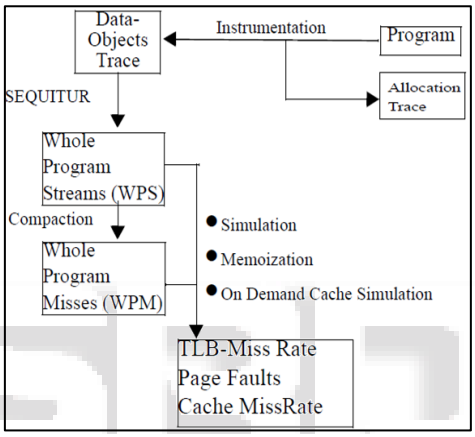


Fig. 5: Trace the data objects and Allocation

Heterogeneity-aware selective data layout scheme for parallel file systems with both HDD and SSD-based servers. We alleviate the inter-server load imbalance by varying the file stripe sizes or the number of files on different servers based on their storage performance. In addition, we selects the optimal static data layout from three types of candidates for applications with specific access patterns.

Moreover, to adapt to dynamic changes of access behaviours in some complex applications, we also developed a dynamic data layout strategy that stores file with multiple copies, each using a different layout policy, and selects the proper copy with the lowest cost to serve I/O requests.

Generally, a large number of copies for the dynamic layout policy would lead to a better performance, but also come at a higher cost. In principle, HAS improves hybrid parallel file system performance by matching data layout with both application characteristics and storage capabilities. We have developed and presented the proposed layout optimization scheme under MPICH2 and OrangeFS. Experimental results show that HAS improves the I/O performance by up to 292.7 percent over the existing file data layout schemes.

VI. COMPARISON

HAS with two other data layout schemes: The application-aware scheme (ADL) and the storage aware scheme (SDL).				
S. No	Data layout Scheme	Description	Pros	Cons

1	ADL The application-aware scheme (ADL)	In ADL, file data is placed across the hybrid file servers with one of the three policies according to the application's access pattern, but each server is assigned a fixed-size file stripe.	Since ADL and SDL have shown considerable performance improvements over the default data layout scheme namely the fixed-size striping	Accuracy Less throughput
2	SDL Storage aware scheme	In SDL file stripe sizes for the hybrid servers are determined by the server performance but only 1–DH policy is chosen, without fully considering application access patterns.	Fixed sized Strops	High Accuracy Less Read Performance
3	HAS Heterogeneity-aware selective data layout scheme	We use the popular benchmark IOR, BTIO, HPIO and a real application to test the performance. We first show the efficiency of the selective static data layout scheme for a specific access pattern, then we show the efficiency of the dynamic data layout for mixed access patterns.	Mixed access Patterns and Various Data Layout Schemes	High Throughput High Read Performance

Table 1: Comparisons of ADL, SDL, HAS

VII. Experimental Scenario

Experimental results show the efficiency of proposed algorithm. From analysis we can compare the results between Improved Autonomous Power Control and Least Degree for-K.

A. Application

Audio and streaming video are encoded to packet data first and then send to network through User Datagram Protocol (UDP) transport. Packets arrival to receiver after travel a several time in network and are decoded by decoder.

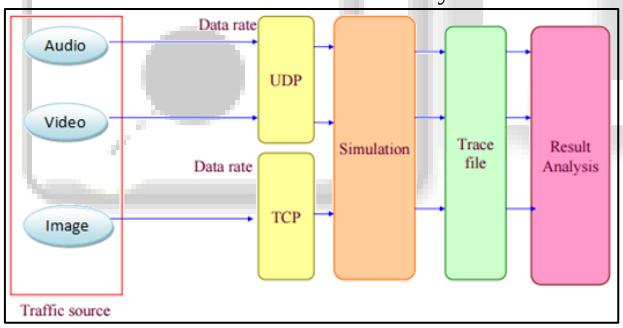


Fig. 5: Transfer data packets through protocol

The protocols are evaluated for packet delivery ratio, throughput, and average end-to-end delay.

B. Throughput Comparisons

We know that throughput increases when connectivity is better. It can be observed that the performance of the IAPCMAC reduces drastically while Least Degree for-K is slightly better among the existing protocols.

C. Throughput

It is defined as the total number of packets delivered over the total simulation time. The throughput comparison shows that the two algorithms performance margins are very close under traffic load of 50 and 100 nodes in scenario and have large margins when number of nodes increases to 200.
Mathematically, it can be defined as:

D. Throughput= N/1000

Where N is the number of bits received successfully by all destinations.

E. Packet Drop

Packet drop is defined as the total number of packets dropped during the simulation. Mathematically, it can be defined as:
Packet lost = Number of packet send – Number of packet received

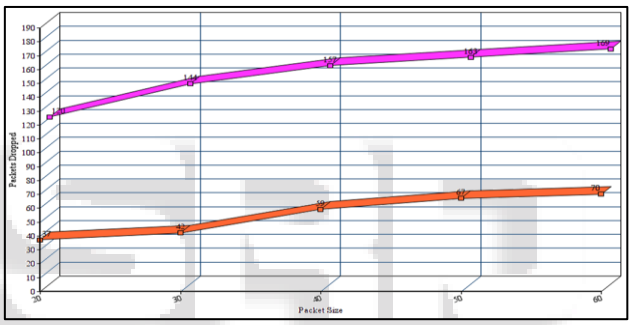


Fig. 6: Packet Drop

F. Packet Drop Ratio

Packet drop is defined as the ratio of total number of packets dropped during the simulation. Mathematically, it can be defined as: Packet Drop Ratio = Number of packet send / Number of packet received

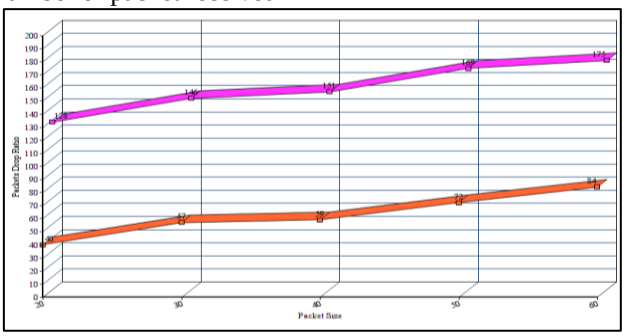


Fig. 7: Packet Drop Ratio

G. Packet delivery ratio

Packet delivery ratio is defined as the ratio of data packets received by the destinations to those generated by the sources.

Mathematically, it can be defined as: PDR= S1÷ S2

Where, S1 is the sum of data packets received by the each destination and S2 is the sum of data packets generated by the each source.

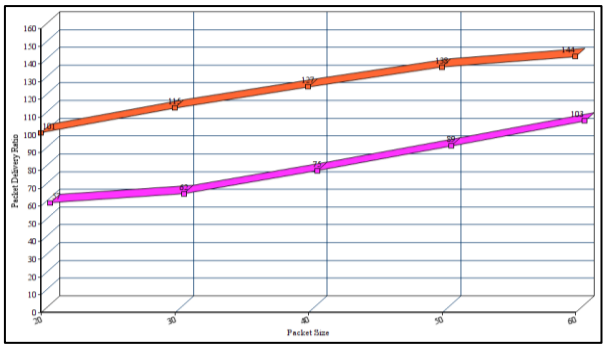


Fig. 8: Packet Delivery Ratio

Graphs show the fraction of data packets that are successfully delivered during simulations time versus the number of nodes. Performance of the IAPCMAC is reducing regularly while the Least Degree for-K is increasing. Least Degree for-K is better among the IAPCMAC protocol.

VIII. CONCLUSION

We alleviate the inter-server load imbalance by varying the file stripe sizes or the number of files on different servers based on their storage performance. In addition, we select the optimal static data layout from three types of candidates for applications with specific access patterns. Moreover, to adapt to dynamic changes of access behaviours in some complex applications, we also developed a dynamic data layout strategy that stores file with multiple copies, each using a different layout policy, and selects the proper copy with the lowest cost to serve I/O requests. Generally, a large number of copies for the dynamic layout policy would lead to a better performance, but also come at a higher cost. In principle, HAS improves hybrid parallel file system performance by matching data layout with both application characteristics and storage capabilities. We have developed and presented the proposed layout optimization scheme under MPICH2 and OrangeFS. Experimental results show that HAS improves the I/O performance by up to 292.7 percent over the existing file data layout schemes

REFERENCES

[1] Marti, Sergio, Thomas J. Giuli, Kevin Lai, and Mary Baker. "Mitigating routing misbehavior in mobile ad hoc networks." In Proceedings of the 6th annual international conference on Mobile computing and networking, pp. 255-265. ACM, 2000.

[2] Liu, Kejun, Jing Deng, Pramod K. Varshney, and Kashyap Balakrishnan. "An acknowledgment-based approach for the detection of routing misbehavior in MANETs." Mobile Computing, IEEE Transactions on 6, no. 5 (2007): 536-550

[3] Sheltami, Tarek, Anas Al-Roubaiey, Elhadi Shakshuki, and Ashraf Mahmoud. "Video transmission enhancement in presence of misbehaving nodes in MANETs." Multimedia systems 15, no. 5 (2009): 273-282

[4] E. M. Shakshuki , N. Kang and T. R. Sheltami "EAACK—A secure intrusion detection system for MANETs", IEEE Trans. Ind. Electron., vol. 60, no. 3, pp.1089 -1098 2013

[5] D. Dong, X. Liao, Y. Liu, C. Shen and X. Wang, "Edge Self-Monitoring for Wireless Sensor Networks," IEEE Transactions on Parallel and Distributed Systems," vol. 22, no. 3, March 2011, pp. 514-527.

[6] I. Khalil, S. Bagchi and N. B. Shroff, "SLAM: Sleep-Wake Aware Local Monitoring in Sensor Networks," Proc. 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2007 (DSN 2007), 565-574.

[7] T. Hoang Hai and E-N. Huh, "Optimal Selection and Activation of Intrusion Detection Agents for Wireless Sensor Networks," Proc. Future Generation Communication and Networking (FGCN 2007), vol.1, no., pp.350-355, 6-8 Dec. 2007.

[8] S. M. Fitaci, K. Jaffres-Runser and C. Comaniciu, "On modeling energy security trade-offs for distributed monitoring in wireless ad hoc networks," Proc. Military Communications Conference, 2008. MILCOM 2008. IEEE, vol., no., pp.1-7, 16-19 Nov. 2008.

[9] N. Tsikoudis, A. Papadogiannakis and E. P. Markatos, "LEoNIDS: a Low-latency and Energy-efficient Network-level Intrusion Detection System," IEEE Transactions on Emerging Topics in Computing, Vol. PP, no. 99, 2014.

