# LambdaRAM: A High-Performance, Multi-Dimensional, Distributed Cache Over Ultra-High Speed Networks

BY

VENKATRAM VISHWANATH
B.E., University of Mumbai, 1999
M.S., University of Illinois at Chicago, 2003

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2009

Chicago, Illinois

## ACKNOWLEDGEMENTS

I owe a huge debt of gratitude to my committee, who have also been my mentors over the years: Jason Leigh, Larry Smarr, Bob Grossman, Andy Johnson, and Mike Seablom.

I would like to thank Maxine Brown and Lenore Zuck for being great mentors.

I would like to thank Rob Burns for all his support with the dissertation. I would also like to thank NASA GSFC, the OptIPuter community and the ON-Vector community for their support.

I would like to thank the staff members and students at EVL who have supported me throughout my tenure at EVL.

Finally I will always be grateful to my family for all their encouragement and unconditional support.

VV

## TABLE OF CONTENTS

## TABLE OF CONTENTS (Continued)

## LIST OF TABLES

## LIST OF FIGURES

## LIST OF FIGURES (Continued)

## LIST OF ABBREVIATIONS

| | |
|---|---|
| API | Application Programming Interface |
| CPU | Central Processing Unit |
| EVL | Electronic Visualization Laboratory |
| GPU | Graphics Processing Unit |
| GSFC | Goddard Space Flight Center |
| HDF | Hierarchical Data Format |
| HPC | High Performance Computing |
| IP | Internet Protocol |
| LAN | Local Area Network |
| LRAM | LambdaRAM |
| MAN | Metropolitan Area Network |
| MAP | Modeling Analysis and Prediction Program |
| MERRA | Modern Era Retrospective Re-Analysis for Research and Applications |
| NASA | National Aeronautics and Space Administration |
| NSF | National Science Foundation |
| NUMA | Non Uniform Memory Access |
| PVFS | Parallel Virtual File System |
| RBUDP | Reliable Blast User Datagram Protocol |
| RTK | Rails Toolkit |
| TCP | Transmission Control Protocol |
| UDP | User Datagram Protocol |
| UIC | University Illinois at Chicago |
| WAN | Wide Area Network |

## SUMMARY

Interactive, real-time exploration and correlation of multi-terabyte and petabyte datasets from multiple sources are critical to advancing scientific discovery in many disciplines, including climate modeling and prediction, biomedical imaging, geosciences, high-energy physics and homeland security. These data-intensive applications are now being enabled by the OptIPuter, a new paradigm that relies on multi-gigabit photonic networks to interconnect distributed storage, computing and visualization resources, thereby creating a planetary-scale supercomputer. Critical performance bottlenecks have been the access latencies associated with reading and/or writing data from/to storage systems and clusters, whether connected via local or wide-area networks. In the case of climate modeling and analysis, the NASA climate Modeling, Analysis, and Prediction (MAP) project noted that these latencies result in "…the supercomputers remaining idle for 25-50% of the execution time during the analysis phase." Reducing I/O latency would allow researchers to run more complex models during the same timeframe, resulting in faster and more accurate weather prediction.

LambdaRAM is a high-performance, multi-dimensional, distributed cache that harnesses the memory of cluster nodes in one or more clusters that are interconnected by ultra-high-speed networks, providing data-intensive applications with rapid access to both local and remote data. LambdaRAM's memory and data management enables applications to rapidly stride over multi-dimensional multi-terabyte scientific datasets.

It employs novel proactive latency mitigation heuristics, including presending and prefetching, based on the access patterns of an application, and data transfer protocols designed for high-bandwidth networks to mitigate data access latencies. Using LambdaRAM, climate analysis applications to compute wind shear, surface temperature and ozone thickness are able to rapidly stride through multi-terabyte datasets over both local and wide-area networks and achieve up to 20-fold improvement in performance.

# 1. INTRODUCTION

Interactive real-time exploration and correlation of multi-terabyte and petabyte datasets from multiple sources has been identified as a critical enabler for scientists to glean new insights in a variety of disciplines critical for national security, including climate modeling and prediction, biomedical imaging, geosciences, and high-energy physics [Newman03]. The critical performance bottlenecks in such data-intensive applications are the access latencies associated with storage systems and remote data access. In the case of climate modeling and analysis, the NASA climate Modeling, Analysis, and Prediction (MAP) project noted that these latencies result in "…the supercomputers remaining idle for 25-50% of the execution time during the analysis phase" [Seablom08]. Reducing I/O latency would allow researchers to run more complex models during the same timeframe, resulting in faster and more accurate weather prediction. Additionally, researchers copy terabyte-sized datasets located at other NASA centers or other organizations to local systems in order to run the models, which also incurs some overhead. If the remote datasets could be accessed quickly and as transparently as possible from the remote location, the overhead costs associated with copying and storing the datasets can be significantly reduced. This would enable real-time remote data analysis, reduce errors associated with replicated data and mitigate the cost of the storage systems for these replicas. A key challenge in NASA's weather simulations is to couple multiple models, including atmospheric and ocean models, and, requires sharing of data, in real-time, between the various models that may be running on geographically distributed clusters. Model coupling would enable higher resolution and accurate weather prediction.

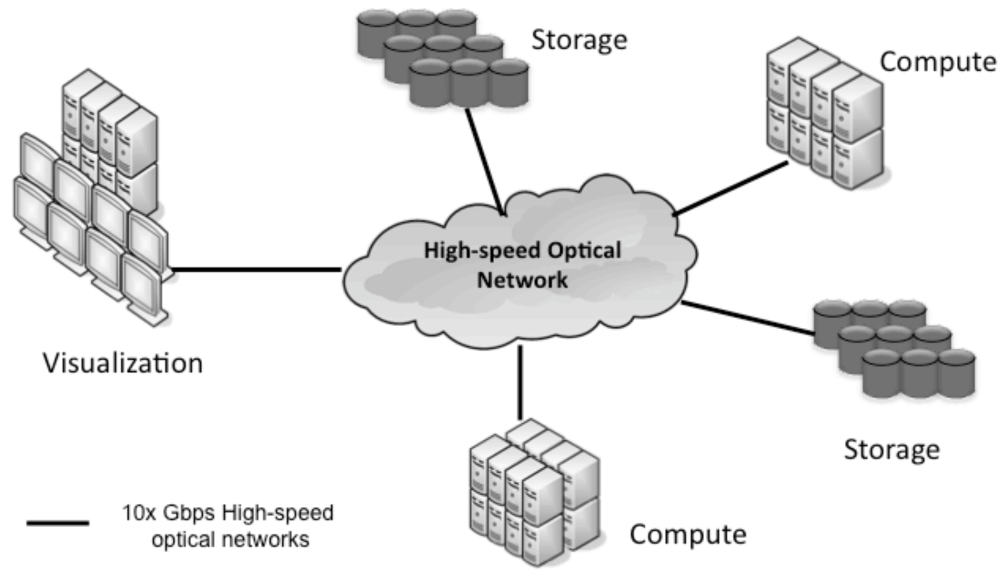
Storage
Compute
High-speed Optical Network
Visualization
Storage
Compute
10x Gbps High-speed optical networks

Figure 1: The OptIPuter approach for Data-Intensive Distributed HPC

These data-intensive real-time experiments are now being enabled by the OptIPuter, a new paradigm in data-intensive distributed computing, funded by the National Science Foundation (NSF), to build a real-time planetary-scale supercomputer by interconnecting distributed storage, computing, and visualization resources over ultra-high speed photonic networks at tens of gigabits per second, thereby creating a planetary-scale supercomputer. This is also known as a LambdaGrid and is depicted in Figure 1. LambdaGrids are becoming increasingly prevalent in data-intensive science as they enable real-time data access and collaboration, and, are being deployed for applications including High Energy Physics, Astronomy, Meta-genomics and weather prediction. LambdaGrids serve as the system-bus of the meta-computer enabling data-intensive science. However, novel solutions are needed to fully exploit the large network bandwidth and overcome the latency associated with the planetary scale interconnects and provide real-time data access and data collaboration.

We are also witnessing a trend where the network bandwidth to remote memory is extremely large and far exceeds the bandwidth to the local node. This trend is depicted in. Data-Intensive HPC will need to fully exploit this large network bandwidth in order to scale their performance to petascale and future exascale architectures.

Table 1: Bandwidth Trends In End-Systems

| | **2008** | **2010** |
|---|---|---|
| Number of cores | 4 | 12 |
| DRAM BW (Gbps) | 80 | 125 |
| Memory BW / core (Gbps) | 20 | 10 |
| Expected memory BW / core (Gbps) (half the cores access memory concurrently) | 40 | 20 |
| BW to remote on-board memory (Gbps) | 80 | 160 |
| BW to remote node memory (Gbps) | 10 | 100 |

In this dissertation, we present LambdaRAM, the memory sub-system for LambdaGrids. LambdaRAM is a high-performance, multi-dimensional, distributed cache that harnesses the memory of cluster nodes in one or more clusters that are interconnected by ultra-high-speed networks, providing data-intensive applications with rapid access to both local and remote data. LambdaRAM's memory and data management enables applications to rapidly stride over multi-dimensional multi-terabyte scientific datasets. It employs novel proactive latency mitigation heuristics, including presending and prefetching, based on the access patterns of an application, and data transfer protocols designed for high-bandwidth networks to mitigate data access latencies.

The novel achievements of the dissertation include:

- Enabling distributed data-intensive computing by exploiting the large network bandwidth of optical networks interconnecting storage, compute and visualization resources.
- A memory harnessing framework encompassing the memory of one or more clusters, spanning a gamut of ultra-high-speed networks including Local Area Networks (LAN), Metropolitan Area Networks (MAN) and Wide Area Networks (WAN). This enables seamless access to geographically distributed data.
- A novel latency mitigation framework, incorporating pull-based, push-based and hybrid heuristics, to address the latency needs of Data-Intensive applications over ultra-high-speed LAN, MAN and WAN.
- Scalable memory management heuristics and multi-dimensional distributed data structures addressing the multi-terabyte datasets of data-intensive applications.
- Modular and extensible design together with intuitive interfaces enabling high productivity and high performance computing.
- To the best of our knowledge, our work is the first to explore and use formal verification techniques in data-intensive high performance computing. This ensures reliable deployment in mission-critical environments including real-time weather prediction.

- Using LambdaRAM, climate analysis applications to compute wind shear, surface temperature and ozone thickness are able to rapidly stride through multi-terabyte datasets over both local and wide-area networks and achieve up to 20-fold improvement in performance.
- Demonstrated with diverse real-world applications, including data visualization of bioscience and geosciences data, and, NASA's climate analysis applications.
- End-system, topology-aware, resource abstractions to enable applications efficiently utilize current and future high-end systems. This helps data-intensive middleware scale their performance to the end-system architecture of petascale and future exascale systems.

The dissertation is organized as follows: In Chapter 2, we discuss the requirements of data intensive applications. We examine the related work in Chapter 3. We discuss the design and implementation of LambdaRAM in Chapter 4. In Chapter 5, we evaluate the performance of LambdaRAM with micro-benchmarks and application-benchmarks using NASA's climate analysis applications. We present an initial formal verification of our design in Chapter 6, and, discuss the rails toolkit that enables middleware to fully exploit future architectures in chapter 7. Finally, We conclude in Chapter 8.

# 2. REQUIREMENTS OF DATA-INTENSIVE HIGH PERFORMANCE COMPUTING

In this chapter, we elucidate the needs of data-intensive high performance computing applications. We first categorize the I/O classes found in the various HPC domains in Section 2.1 and present the requirements of data-driven science in Section 2.2.

## 2.1. Data Intensive HPC I/O Classes

Figure 2 depicts the various data-intensive HPC application classes categorized based on their IO patterns. They can be broadly classified as:

- Compute centric applications
- Data reduction applications
- Data production applications
- Hybrid combinations of the above

Compute centric applications have low data requirements and spend majority of the time on the computation. They typically need a small amount of data to initialize the computation and produce a small output. The bulk of the data requirements are due to the checkpoint and restart data associated with the computations. The compute-intensive parts of the applications have strict data consistency requirements while the checkpoint and restart could be categorized as Write-Once Read-Many data access.

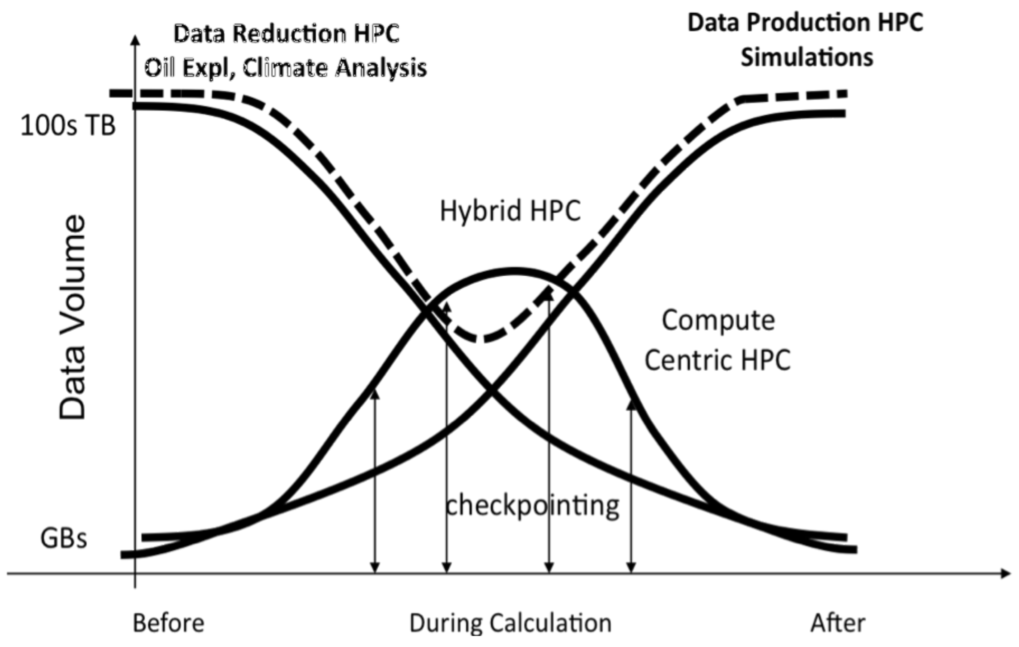

Figure 2: The Data-intensive HPC application classes

Data reduction applications represent the class of applications that need access to large amount of data at the beginning of their computation and the data requirements decrease as the computation progresses. Oil exploration analysis is one example where the applications need to access and analyze large multi-dimensional ocean and atmospheric datasets to decide if oil is present at the well and if it's worth investing millions of dollars to drill at the location. These applications typically need Read-Only data access. Most data analysis and data mining are examples of data reduction applications.

Data production applications represent the class of applications such as outputs of simulations, sensors, etc which produce large amounts of data for storage and later analysis. The initial data requirements of these applications are low. However, by the end of the computation, these applications tend to produce multiple Terabytes of data. These applications are typically Write-Once consistency mode.

Hybrid combinations of Data reduction, compute centric and data production applications are common in Data intensive computing. Climate Analysis and modeling consists of reading in historical data files and latest weather forecast data, applying compute-intensive models to the data and finally producing new forecast that are written to storage. This behavior is also found in various other fields including computation biology and astrophysics. These applications tend have distinct stages comprising of Read-Only, Write-Many, Write-Once Read-Many access patterns.

## 2.2. Requirements of Data-Intensive HPC Domain sciences

In earth sciences, the climate analysis and prediction applications routinely require real-time access to geographically distributed data repositories, data from sensor and satellites, data from simulations, etc., in-order to make more accurate weather predictions. The datasets are typically multi-dimensional and climate applications require to rapidly stride through local and remote datasets. Coupling multiple earth science models, such as atmospheric and ocean models, will result in a more accurate weather prediction. Model coupling requires coupling of data between multiple simulations. During a hurricane, an effective and timely evacuation requires to access and couple data from multiple sources and even domains, including real-time weather simulations, high-resolution terrain and aerial maps, population data and current traffic information. Other domains sciences including astrophysics, computational chemistry, biosciences, oil exploration, data visualization, have similar requirements as listed above. This is also depicted in Table 2.

The needs of data-intensive HPC applications include:

- Low latency access to data. The data could be located on a local storage system or geographically distributed.
- Access to multiple remote repositories.
- Ability to access multi-terabyte and petabyte sized datasets.
- Rapidly stride over multi-dimensional scientific data.
- Ability to interface with multiple storage systems.

Table 2: The Requirements of Data-Intensive Domain Sciences

| **Domain** | **Data sizes** | **Dimensions** | **Remote Data Access** | **Data Coupling Needs** |
|---|---|---|---|---|
| Climate Analysis | 10 PB | 2D, 3D, 4D | Y | Y |
| Astrophysics | 10 PB | 3D, 4D | Y | Y |
| Oil Exploration | 100 TB | 3D, 4D | Y | Y |
| Computational Chemistry | 100 TB | 3D, 4D | Y | Y |
| Biosciences | 100 TB | 2D, 3D | Y | Y |

# 3. RELATED WORK

In this chapter, we examine relevant related work and compare them with LambdaRAM. The dissertation is influenced by research in distributed shared memory, parallel filesystems, distributed storage and memory servers are relevant to LambdaRAM. We examine each of them in detail and compare them with LambdaRAM in Table 3.

**Distributed Shared Memory:**

Global Arrays [Nieplocha06] and Charm++ [DeSouza04] are two commonly used distributed shared memory implementations in HPC. Similar to LambdaRAM, they work on multi-dimensional datasets. While these systems support only local cluster operation, LambdaRAM leverages our expertise in ultra-high-speed wide-area networking and low-latency reliable high-speed transport protocols [He02] [Xiong05] [Vishwanath06] to enable it to extend over multiple geographically distributed clusters in both local and wide-area networks.

**Parallel Filesystem:**

Parallel File systems, including PVFS2 [PVFS2], GPFS [Schmuck02] [Liao05] and Network File systems including NFS [NFS], support client-side file caching and work on a local cluster scale and at a file-level. This is complementary to LambdaRAM that works at a multi-dimensional dataset-granularity spanning multiple files. In fact, we are using LambdaRAM as a caching layer for scientific datasets stored in PVFS2-based storage systems and will work towards a closer integration of the two.

**Distributed File-systems and P2P Storage:**

Distributed File systems, including, Storage Resource Broker [Moore01], GASS, Storage Resource Manager [Shoshani02] and IBP [Bassi03], and P2P storage including Sector [Gu06] provide efficient access to files over wide-area networks. While these file systems work on a per-file granularity, LambdaRAM works at the granularity of multi-dimensional scientific datasets spread over multiple files. In fact one can even use LambdaRAM on top of these other file systems to enhance their performance by providing low-latency access to multi-dimensional scientific data for parallel applications.

**Memory Servers:**

Memory servers based on Kernel-level implementations, including, Global Memory Service (GMS)[Feeley95], Anemone [Hines06] and dRamDisk [Roussev06], and, user-level implementations, including, Dodo [Koussih99], NetRAM [Anderson94], cooperative caching and JumboMem [Pakin07], have been proposed. However, these implementations support only local area networks.

The key innovation in LambdaRAM is that it extends memory caches in clusters regardless of whether the clusters are located over a LAN, MAN or WAN. LambdaRAM leverages the changing technology landscape whereby bandwidth is becoming cheaper than computing to bring about increases in performance and capability that was not previously possible. To achieve its high data throughput over WANs, LambdaRAM uses high-speed transport protocols including a reliable UDP-based streaming protocol to fully exploit the available bandwidth in LambdaGrids.

Table 3: Comparison of LambdaRAM with related research work

| System | Write-Many Semantic | N Dim | Global Address Space | Cache | Multi-Cluster Hierarchy Config | Wide Area, High Speed Ntwk | Mem QoS | In-Mem Data Coupling betw. Apps |
|---|---|---|---|---|---|---|---|---|
| Parallel File System | **Y** | - | - | - | - | - | - | - |
| Distributed File System & P2P Storage | **Y** | - | - | - | - | **Y** | - | - |
| Cooperative Caching | - | - | - | **Y** | - | - | - | - |
| Dist. Shared Memory | **Y** | **Y** | **Y** | - | - | - | - | - |
| Scientific Data Management | **Y** | **Y** | - | - | - | - | - | - |
| LambdaRAM | - | **Y** | **Y** | **Y** | **Y** | **Y** | **Y** | **Y** |

# 4. DESIGN

In this chapter, we first describe the various physical architectural modes in LambdaRAM to encompass the memory of one or more clusters. In Section 4.2, we describe the functional architecture of LambdaRAM, and, discuss the data access mechanism in Section 4.3. We finally present sample pseudo-code of serial and parallel applications using LambdaRAM in Section 4.4.

## 4.1. Physical Configurations supported in LambdaRAM

The physical architectural configurations supported by LambdaRAM include: a local cluster mode where LambdaRAM spans the memory of nodes of a single cluster; a client-server cluster mode where LambdaRAM spans memory of two clusters; and, a hierarchical cluster mode, wherein, LambdaRAM spans the memory of multiple clusters. This is also known as the client-peer-server mode. We now discuss each in detail.

### 4.1.1 Local Cluster Mode

In the local cluster mode, LambdaRAM harnesses the memory of the nodes of the entire cluster. As depicted in Figure 3, LambdaRAM spans the entire cluster memory of the NASA

GSFC cluster. It manages access to the 700TB multi dimensional MERRA dataset [MERRA] and enables an application, such as the forecast model, to seamlessly stride through the MERRA dataset.

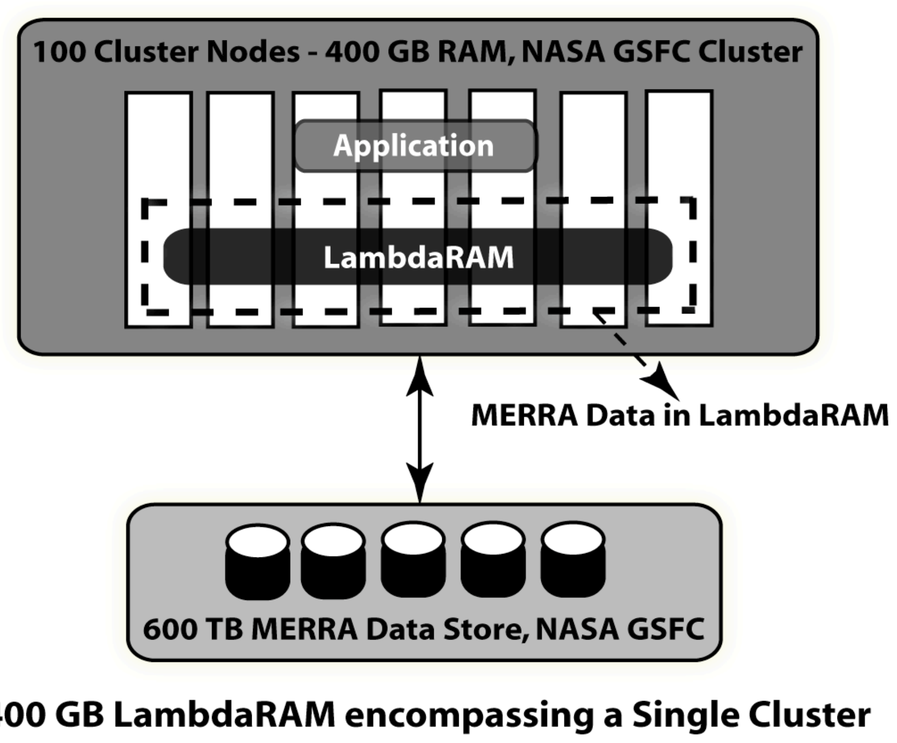
100 Cluster Nodes - 400 GB RAM, NASA GSFC Cluster
Application
LambdaRAM
MERRA Data in LambdaRAM
600 TB MERRA Data Store, NASA GSFC
400 GB LambdaRAM encompassing a Single Cluster

Figure 3: LambdaRAM encompassing the memory of nodes of a single cluster

### 4.1.2 Client-Server Cluster Mode

The client-server cluster mode enables an application to seamlessly access remote data and is the most commonly used configuration. In this case, LambdaRAM encompasses the memory of the cluster where the application runs (client cluster) and the cluster storing the data repository (server cluster). A typical scenario is depicted in Figure 4 wherein a 256-node NASA Ames cluster in California, USA and a 100-node NASA GSFC cluster in Maryland, USA are interconnected by dynamically provisionable ultra-fast high-speed optical networks over the National Lambda Rail (NLR). In the figure, a parallel weather prediction application running on the NASA Ames cluster routinely needs to access the MERRA dataset located at NASA GSFC in Maryland. In this case, LambdaRAM encompasses the combined memory of the two clusters, caches upto 1.2TB of the MERRA data in memory at any given time, and, manages the data access to the remote multi-terabyte MERRA data repository for the application.

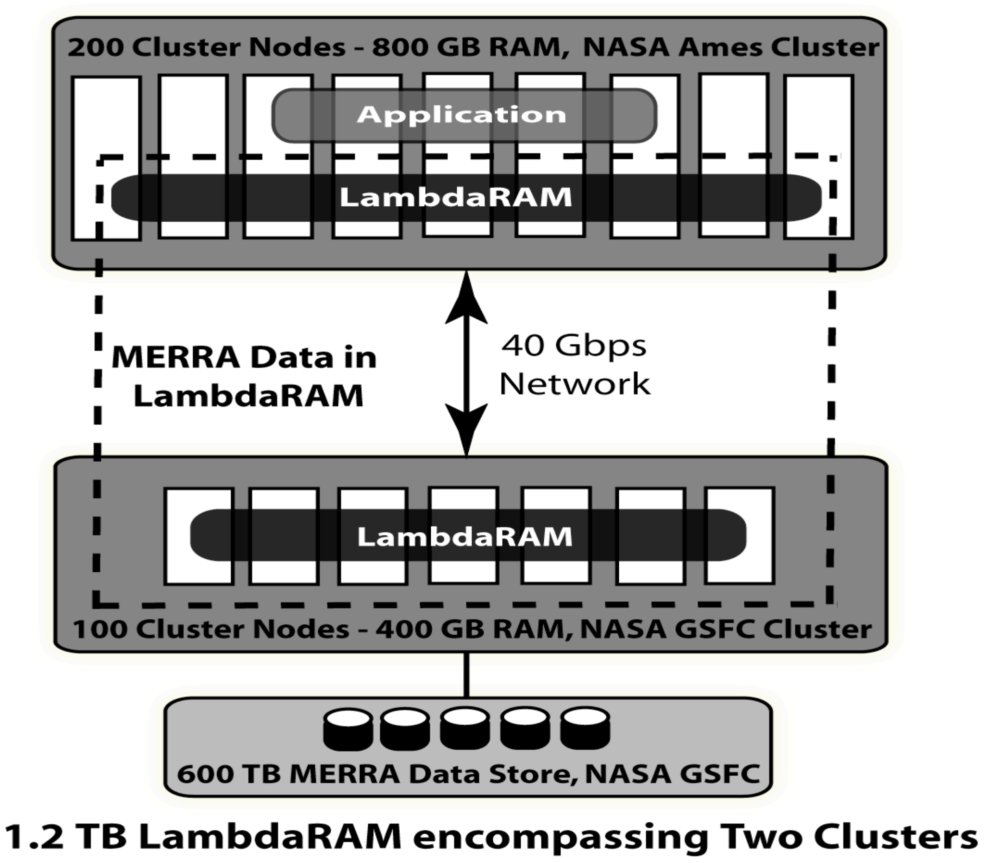
200 Cluster Nodes - 800 GB RAM, NASA Ames Cluster
Application
LambdaRAM
MERRA Data in LambdaRAM
40 Gbps Network
LambdaRAM
100 Cluster Nodes - 400 GB RAM, NASA GSFC Cluster
600 TB MERRA Data Store, NASA GSFC
1.2 TB LambdaRAM encompassing Two Clusters

Figure 4: Client-Server LambdaRAM Configuration

### 4.1.3 Hierarchical Cluster Mode

LambdaRAM supports a hierarchical cluster configuration wherein it can encompass the memory of multiple clusters. This is also known as the Client-Peer-Server mode. This is typically used when there are available clusters along the network path between the application cluster and the data repository cluster. As depicted in Figure 5, LambdaRAM, for the MERRA application, harnesses the memory of clusters at NASA GSFC, TeraGrid Chicago and NASA Ames connected via high-speed optical networks, and, manages the application's data requests to the MERRA data repository. For the MERRA application, the client LambdaRA runs on the NASA Ames cluster, Peer LambdaRAM on the TeraGrid cluster and the Server LambdaRAM on the NASA Goddard cluster.

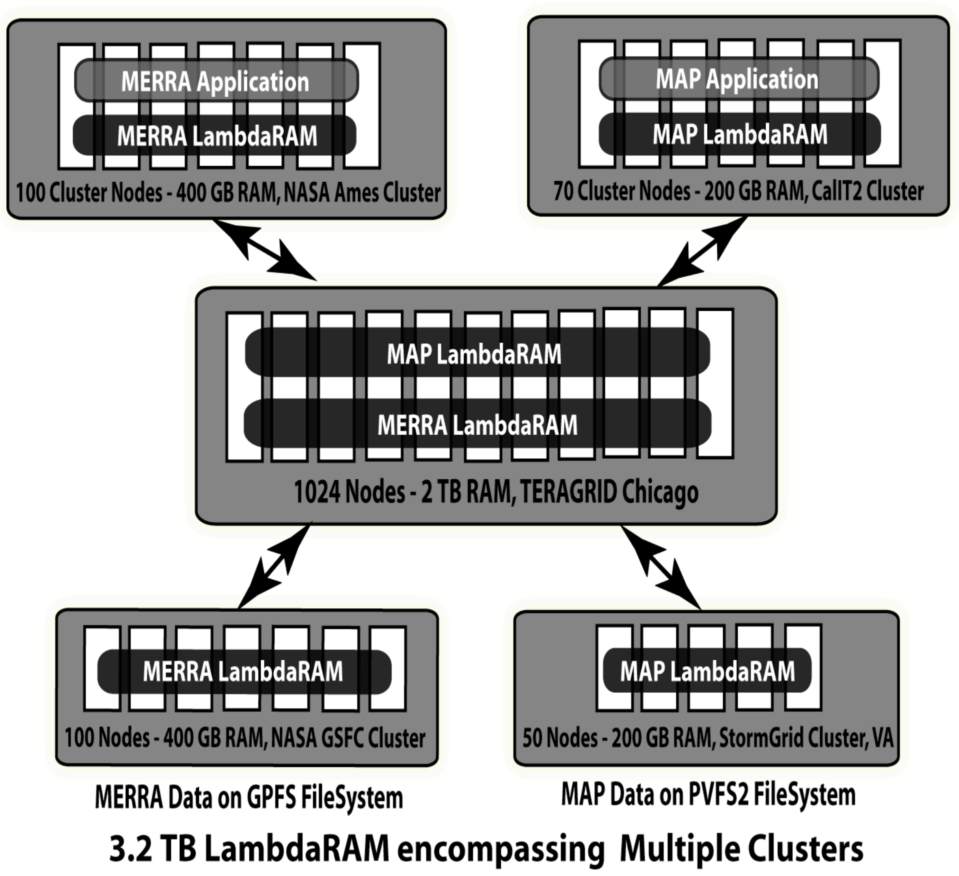
MERRA Application
MERRA LambdaRAM
100 Cluster Nodes - 400 GB RAM, NASA Ames Cluster
MAP Application
MAP LambdaRAM
70 Cluster Nodes - 200 GB RAM, CalIT2 Cluster
MAP LambdaRAM
MERRA LambdaRAM
1024 Nodes - 2 TB RAM, TERAGRID Chicago
MERRA LambdaRAM
100 Nodes - 400 GB RAM, NASA GSFC Cluster
MAP LambdaRAM
50 Nodes - 200 GB RAM, StormGrid Cluster, VA
MERRA Data on GPFS FileSystem
MAP Data on PVFS2 FileSystem
3.2 TB LambdaRAM encompassing Multiple Clusters

Figure 5: Multi-Cluster Hierarchical LambdaRAM Configuration

## 4.2. Architecture

The functional architecture of LambdaRAM is depicted in Figure 6. The data access layer satisfies an application's data requests. It interacts with the distributed data cache to satisfy these data requests. The distributed data cache spans the memory of cluster nodes on one or more clusters. If the requested data is not present in the distributed data cache, it fetches the data from the storage system using the I/O abstraction layer. We now describe each subsystem in detail.

### 4.2.1 Data Access Layer

An application can access data in LambdaRAM using either the data access API or the interposer Layer. The data access API presents and intuitive set of interfaces to access and stride through multi-dimensional datasets while the Interposer layer enables applications to use LambdaRAM without any modifications to the implementation. We discuss each in detail.

#### 4.2.1.1 Data Access API

The Data Access API enables manipulation of a multi-dimensional dataset, distributed over several files, as single multi-dimensional array. This enables a scientist to focus on the science rather than spending time on the data management and access issues. The API has evolved over time based on collaborations and interactions with Data visualization applications, Bioscience applications and Climate analysis and modeling applications. It presents an application with an intuitive multi-dimensional array-like interface to access scientific datasets. The current implementation supports 1D, 2D, 3D and 4D arrays. It can be easily extended to support additional dimensions. The API is aimed at new applications being developed as well as applications that can change their implementations to use the API. It also supports multi-dimensional strided access.

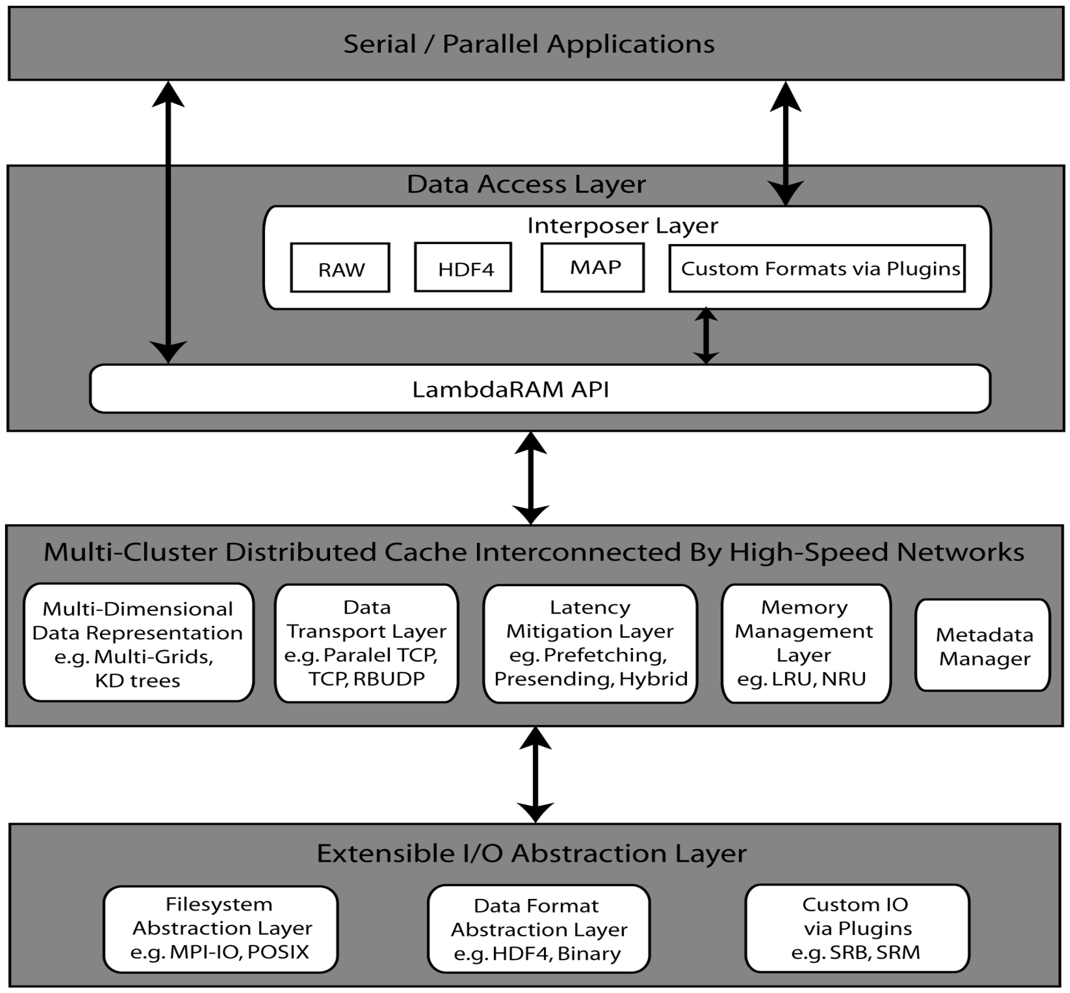
Serial / Parallel Applications
Data Access Layer
Interposer Layer
RAW
HDF4
MAP
Custom Formats via Plugins
LambdaRAM API
Multi-Cluster Distributed Cache Interconnected By High-Speed Networks
Multi-Dimensional Data Representation e.g. Multi-Grids, KD trees
Data Transport Layer e.g. Paralel TCP, TCP, RBUDP
Latency Mitigation Layer eg. Prefetching, Presending, Hybrid
Memory Management Layer eg. LRU, NRU
Metadata Manager
Extensible I/O Abstraction Layer
Filesystem Abstraction Layer e.g. MPI-IO, POSIX
Data Format Abstraction Layer e.g. HDF4, Binary
Custom IO via Plugins e.g. SRB, SRM

Figure 6: LambdaRAM Architecture

#### 4.2.1.2 Interposer Layer

The interposer layer enables an application to use LambdaRAM without modifying a single line of code. This enables easy integration of LambdaRAM with existing scientific applications. Applications accessing data using interfaces, including POSIX I/O, HDF4, etc., can seamlessly use LambdaRAM without any code modifications. The interposer layer works by intercepting the data access calls of an application and mapping them onto relevant LambdaRAM data access API calls. To use the interposer layer, one needs to set the LD_PRELOAD environment variable to the appropriate LambdaRAM interposer library. The LambdaRAM interposer library takes as input a list of files that it should manage. IO requests to files not managed by LambdaRAM are forwarded to system IO or the relevant library. LambdaRAM currently supports seamless integration with applications using the HDF4, Binary and Raw data.

**POSIX I/O Interposer:**

Figure 7 depicts the POSIX File I/O Interposer design of LambdaRAM. This interposer enables applications accessing data with the glibc file I/O library to use LambdaRAM without any code changes. The interposer library intercepts all system IO calls and maps them onto relevant LambdaRAM data access API calls. The interposer layer also enables an application to seamlessly access remote geographically distributed data as if they were locally present without any code modification.

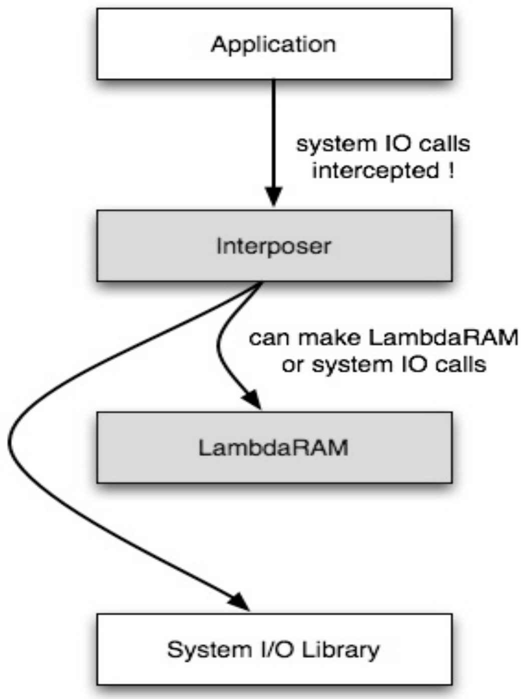
Application
system IO calls intercepted !
Interposer
can make LambdaRAM or system IO calls
LambdaRAM
System I/O Library

Figure 7: POSIX I/O Interposer

**HDF4 Library Interposer:**

HDF4 is a hierarchical data format library developed by the HDF group at the University of Illinois at Urbana Champaign. It is a commonly used data format in earth and atmospheric sciences. NASA uses HDF4 in numerous projects including the MERRA project to store approximately 700TB of data. A HDF4 interposer library would enable NASA's existing applications to use LambdaRAM without modifying a single line of code. This would also enable these applications to access remote HDF4 datasets and rapidly stride though them in multiple dimensions using LambdaRAM. We have prototyped an interposer library to map relevant HDF4 Scientific Dataset (SD) I/O calls to LambdaRAM calls. SD is a commonly used set of HDF4 functions for accessing scientific datasets. In the current prototype, we mapped only the most commonly used HDF4 SD calls typically used in Earth science applications to appropriate LambdaRAM calls.

### 4.2.2 Multi-Dimensional Distributed Cache Layer

The multi-dimensional distributed cache enables efficient access to local and geographically distributed multi-dimensional data. It distributes the data among the various clusters via the multi-dimensional data representation layer and manages the caches using the memory management layer. The latency mitigation layer helps overcome the data access latencies by proactively fetching data just before an application needs it. The data is transferred efficiently between the various levels of caches using the data transport layer. The I/O abstraction layer enables one to interface with various data formats present on various filesystems and storage solutions. We now describe each in detail.

#### 4.2.2.1 Multi-Dimensional Data Representation Layer

Scientific datasets are typically multi-dimensional. Earth science datasets typically consists of 3-D and 4-D datasets. The multi-dimensional data representation layer enables support for multi-dimensional scientific datasets. The data representation is a distributed data structure spread among the various nodes participating in LambdaRAM at a given cluster or level. The current implementation supports regular structured scientific 1D, 2D, 3D and 4D datasets. This enables support for most earth science datasets, 2-D, 3-D and 4D data visualization. This layer can be extended to support additional representations, including unstructured grids and KD-trees, using plug-ins. An application can tailor this layer based on the structural characteristics of its data.

We now discuss the design of the multi-grid data structure in LambdaRAM to support multi-dimensional structured data. Multi-grid data are used in scientific domains, including Astrophysics and Earth Sciences. Figure 8 depicts the multi-grid data representation for a 2-D dataset. The dataset is partitioned into blocks and are distributed among the nodes participating in LambdaRAM. Blocks are further clustered into partitions. Thus, Partitions are also multi-dimensional. Even-though, a block represents multi-dimensional data, it is stored as 1D data in-order to optimize its data transfer. The parameters such as number of partitions, number of blocks per partitions and block size can be set in the LambdaRAM configuration files and are useful for memory management. In the current implementation, the data distribution is static. The data structure is replicated at each level of hierarchy in LambdaRAM.

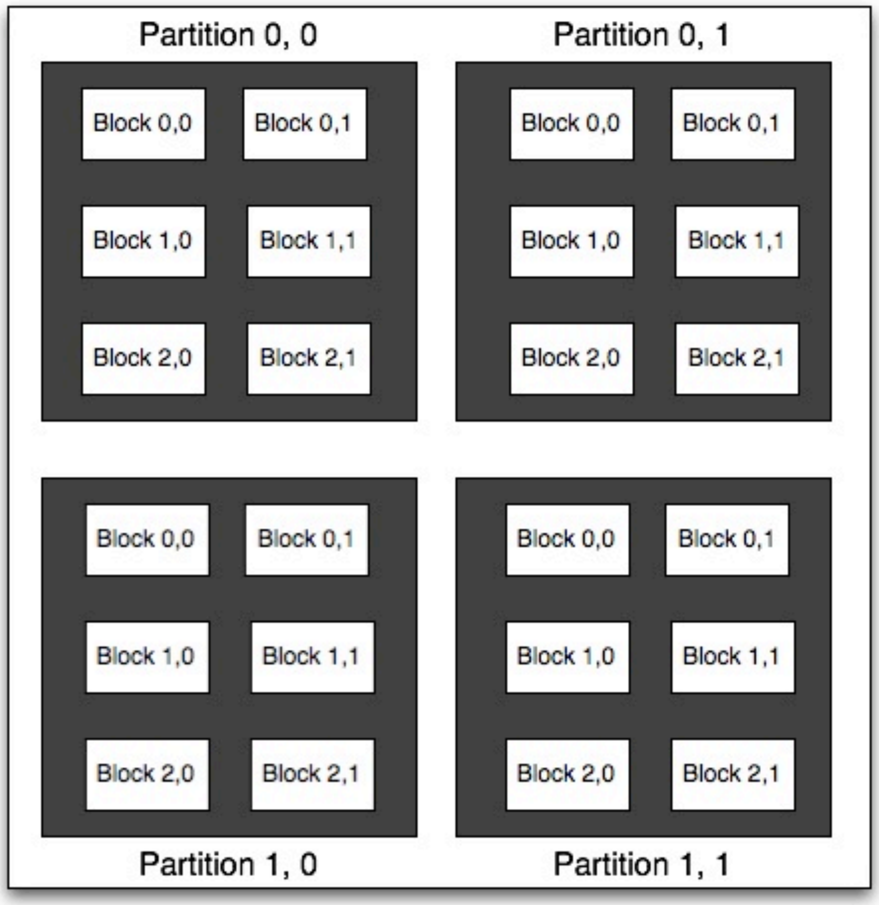

Figure 8: A 2-D Multi-Grid Data Structure

#### 4.2.2.2 Memory Management Layer

Data-intensive HPC applications need to routinely access multi-terabyte sized datasets. The datasets are usually much larger than the combined memory of all the nodes and clusters combined. The memory management layer design needs to be scalable to these handle large datasets.

The memory management heuristics can be either global i.e. it takes the memory state of all the nodes in a cluster into account, or, a simpler local scheme which considers only the usage at the node. Local schemes are useful for applications with high I/O rates such as data striding applications where the latency of making global memory management decisions could adversely affect the performance. Global schemes are better suited to applications with lower I/O data rates. Additionally, in case of the Server and Peer modes of LambdaRAM, local schemes are more appropriate as these clusters could potentially be servicing multiple clients, and, a global scheme would be complex, as it needs to account for each client. The memory management a particular cluster or level in LambdaRAM is independent of the memory management at other levels in the LambdaRAM cache hierarchy. Applications have diverse memory management requirements and we designed this layer to be extensible in order to tune the performance to the needs of an application. The memory management heuristics currently supported includes Least Recently Used (LRU), Most Recently Used (MRU) and LRU augmented with future usage hints. Each cache block has a reference identifier representing its last access. This is used by the memory management heuristics. The memory management employs a sweep based garbage collection scheme wherein a thread periodically checks if a block can be reclaimed based on the heuristics employed. Instead of maintaining a list of blocks sorted based on their last access, we maintain a list of partitions last accessed. This leads to improved concurrency as it reduces contention on accessing the cache blocks. Based on the memory used as a percentage of the maximum memory available to LambdaRAM (an input parameter), we define the following five memory management levels as shown in Table 4. As the level increases based on the memory used, the heuristics dynamically become more aggressive regarding reclaiming cache blocks. We have found this scheme to perform well in practice.

Table 4: Memory Heuristic Levels Based on Usage

| **Memory Level** | **Memory Usage as a percentage of Total Memory** |
|---|---|
| Level 0 | > 0% <= 50% |
| Level 1 | > 50% <= 65% |
| Level 2 | > 60% <= 80% |
| Level 3 | > 80% <= 90% |
| Level 4 | > 90% |

Figure 9 depicts the state transition of a cache block in LambdaRAM. The data access thread (fetch), pre-fetch thread, pre-send thread and the garbage collector can concurrently access a cache block. The design has been optimized to minimize lock contention.

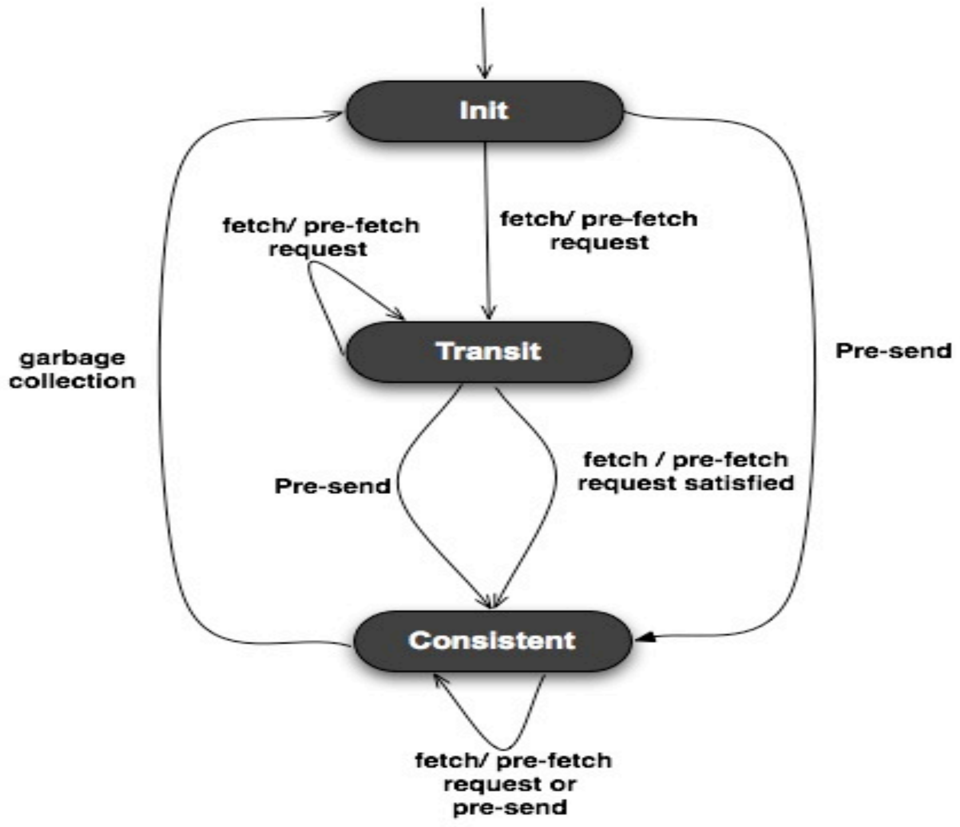

Figure 9: State Transition diagram of a LambdaRAM Cache Block

#### 4.2.2.3 Latency Mitigation Layer

Latency mitigation heuristics are essential to mitigate the access latencies associated with storage systems and wide-area networking. The goal of this layer is to predict and proactively fetch future data blocks, and thus help improve the cache hit ratio for data accesses. LambdaRAM employs two heuristics, namely data pre-fetching and data pre-sending, to proactively fetch data just before an application requests for it.

**Data Pre-fetching**

Pre-fetching is a pull-based scheme initiated by the clients. In this case, the clients explicitly send a pre-fetch request to the server for data blocks. The server replies back with the relevant data blocks. In our design, we also incorporate an opportunistic pre-fetching heuristic wherein the client makes a request for future data blocks and the server fetches the blocks into it's cache if the blocks are uncached. The server does not send the blocks back to the client. This, heuristics tries to ensure that the future blocks to be accessed by the client are at-least cached in-memory at the server end. This is useful when the memory usage of the clients is pretty high and pre-fetching would adversely affect the performance of the application.

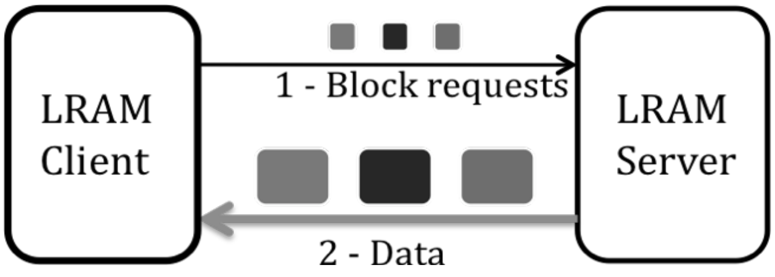

Figure 10: Data Pre-fetching in LambdaRAM

**<u>Data Pre-sending</u>**

In pre-sending, the servers speculate the future access patterns of the clients and push the relevant data blocks onto them. Pre-sending also effectively reduces the latency associated with prefetching by half as it does not incur the request overhead associated with pre-fetching. It improves the bandwidth usage and is useful in high-speed networks. In presending, the data flows, from the server cluster to the client cluster, are regulated based on feedback from the clients. This prevents the presending data flows from overwhelming the clients.

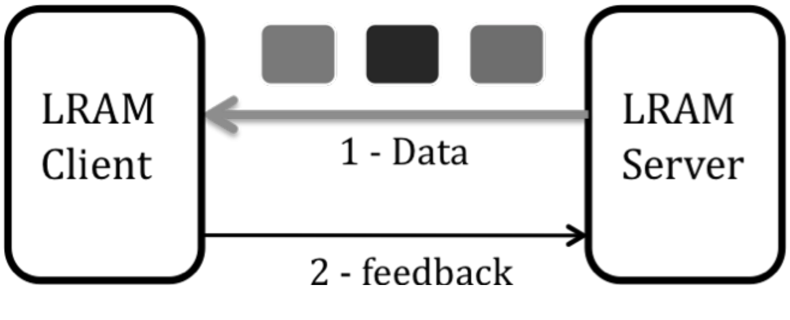

Figure 11: Data Pre-sending in LambdaRAM

Pre-fetching has an inherent request latency i.e. network latency, associated with every request. This is negligible for Local and Metropolitan area networks where the latency is in the order of micro-seconds to 1 milli-second. However, this may This is especially true for applications with large IO rates.

Additionally, one could design a hybrid heuristic combining presending and prefetching where the presending can be augmented with client feedback about the access patterns to enable only relevant data blocks to be pre-sent.

## Design Parameters of the Latency Mitigation Heuristics

The Latency Mitigation layer is extensible and the heuristics can be tuned to the requirements of an application in order to improve the accuracy of prediction. The heuristics needs to take into account a number of parameters to achieve high performance. These include:

- **Application's data access characteristics**

The data access patterns prevalent in most HPC scientific applications include sequential, apriori, strided and random accesses. In the case of climate analysis applications of NASA SIVO, the datasets are typically multi-dimensional and the main access pattern is multi-dimensional data striding. In NASA's wind shear computation for hurricane prediction, the access pattern is data  striding over 4-D data over time and geographical co-ordinates. In case of interactive data visualization, the main access pattern involves accessing regions surrounding the current point of interest.  Apriori data accesses, i.e. the access patterns are known in advance, are used in the reruns of simulations with update or blended datasets. Other data access patterns commonly used include sequential and random access patterns. Random access patterns are used in interactive visualization of weather data while sequential patterns are used in applications including data transfers. In addition to the Data patterns, the I/O request characteristics have diverse request sizes and request rates.  A prediction heuristic needs to account for these patterns and applications in order to improve the cache hit ratio.

- **I/O and Computational Characteristics of Data Intensive applications**

Applications computing zonal averages are primarily I/O bound as they access large data with low computational requirements. In case of I/O bound applications, the heuristics can be complex and thus consume additional CPU cycles and make a more informed prediction. Applications such as Empirical Orthogonal functions computation involve both I/O intensive and compute intensive operations. A key requirement in this case is low CPU utilization of LambdaRAM. Thus, the prediction heuristics need to simple and consume as few CPU  cycles as possible. These are characteristic patterns in several HPC domain sciences including Astrophysics, Metagenomics and Biosciences.

- **Working set size and available memory:**

As described earlier, the memory level regulates how aggressive the latency heuristics are. A lower working set size makes the scheme very conservative while a larger working set size enables one to be more aggressive and pre-fetch/pre-send more cache blocks.

- **Available network bandwidth:**

With network bandwidth becoming abundant and cheap, the heuristics need to fully exploit the available network bandwidth by being aggressive. Additionally, in bandwidth-constrained deployment, the heuristics should be conservative to reduce the network usage.

- **Network latencies:**

Mitigating network latency is a key requirement of the heuristics. The heuristics need to scale to LAN, MAN and WAN range networks. The prediction heuristics need to account for the network latency between the node and the remote peer/server.

- **Presence of Intermediate Cache Clusters:**

LambdaRAM supports the notion of intermediate or hierarchical caches between the data and the application. This is similar to cache hierarchy found in a processor. Intermediate cache clusters present near the client and / or servers are useful for reducing the load on the clients and servers. Thus, one can offload complex and aggressive heuristics onto the intermediate cluster and have simpler schemes that consuming less memory and CPU cycles on the clients and server clusters.

#### 4.2.2.4 Data Transport Layer

The data transport layer is responsible for high performance data transfers within a cluster and between clusters over local and wide-area networks. It uses a threaded all-to-all connection model where it creates a threaded connection between a client node and every server node. The design of this layer is modular and enables one to experiment with various data transport protocols. In the case of intra-cluster communication, one could use TCP or protocols optimized for the interconnect networks such as MX for Myrinet and IB for Infiniband. LambdaRAM creates a thread between a node and to the other nodes of a cluster. For inter-cluster communication over high-bandwidth ultra-fast optical networks, LambdaRAM supports TCP and advanced data transport protocols including Celeritas and Parallel TCP. Additional protocols, including UDT and RBUDP, can be easily supported with the help of plug-ins. The plug-in has a well-defined interface and this enables support for other novel protocols.

We now elucidate the Celeritas protocol designed to support high-performance streaming of data blocks over ultra-high speed networks. A critical requirement for high-performance over ultra-high speed networks and wide-area is protocols that can fully utilize the ultra-high speed networks. In high-bandwidth wide-area networks, TCP - the ubiquitous protocol used for transferring data in the Internet, is unable to efficiently utilize the available bandwidth and thus does not scale to the high bandwidth networks. We have designed Celeritas, a high-speed data transfer protocol, for high-performance reliable data streaming over high-bandwidth networks. Celeritas is an application-level rate-based, UDP-based reliable data transport protocol. Celeritas builds on our prior work done in Reliable Blast UDP (RBUDP), an UDP-based transport protocol for data transfers over dedicated networks. Similar to RBUDP, Celeritas is a user-space protocol. The user-space implementation makes it easier to deploy on production systems. It has been successfully tested on Linux and Mac OSX platforms. A key goal of Celeritas is to scale reliable data-streaming over dedicated WAN to 10 Gbps and beyond for multiple streams.

The key differences between RBUDP and Celeritas are:

- **Error Correction**

Similar to RBUDP, Celeritas uses UDP for transferring data and TCP for transferring control information. A key difference is that, in RBUDP the error correction occurs after the main data transfer. This keeps the link idle for RTT/2 for each message that is very high for many data streaming applications which consists of large number of small (in 10’s of MB) messages. Celeritas overcomes this by having the sender periodically update the receiver with the error list rather than at the end of receiving the message.

- **<u>Transfer list of buffers (vectors)</u>**

Transferring list of blocks is necessary in LambdaRAM. High performance transport protocols such as RBUDP and UDT do not support the ability to transfer a list of vectors i.e. support the scatter-gather operation. This is supported by Celeritas.

- **<u>Flow Control</u>**

Flow control is necessary in order to prevent a sender from overwhelming a receiver. Celeritas's flow control mechanism involves maintaining the number of unacknowledged packets at the sender. If number of unacknowledged packets exceeds the MAX_UNACKNOWLEDGED_PACKETS, progress is halted (i.e. sender stops sending additional data) till it receives acknowledgement for data from the receiver. RBUDP does not incorporate a flow control mechanism and a sender can easily overwhelm a receiver.

- **<u>Congestion Control</u>**

RBUDP was primarily designed for use in dedicated environments together with admissions control. In RBUDP, a key assumption is that an application generates only a single stream and the allocated network bandwidth is for this stream. However, almost all applications in LambdaGrids have multiple data streams requiring efficient co-operative sharing of the bandwidth. Additionally, streams have diverse throughput requirements and congestion control schemes taking this into account are lacking in any current data transfer protocol. In Celeritas, we have a manager daemon for each site to which each connection registers. One can incorporate multiple congestion control algorithms to co-operatively regulate the rates of the multiple streams. The current scheme is simple and uses a rate pre-computed based on the number of flows. In LambdaBridge, we have proposed several schemes that can be incorporated in this framework.

**Additional features of Celeritas include:**

- Supports zero-copy using header prediction.
- Message-based data transfer. This enables support for asynchronous communication and enables one to preserve message boundaries.
- Per-flow rate control capability
- Automatic path MTU discovery
- Performance monitoring

**Performance of Celeritas over Wide Area Networks:**

We evaluate the performance of Celeritas over a 10 Gbps wide-area network between EVL, Chicago and NASA, Goddard. The network comprised of the CAVEWave network between Chicago and Mclean, Virginia, and the DRAGON network between McLean, VA and NASA Goddard. The round trip latency (RTT) for this network is around 19ms. The node at Chicago is a dual-core dual-processor AMD opteron with 4GB RAM and 10GE PCI-e based

Myricom NIC. The node at NASA Goddard is also a dual-core  dual-processor AMD Opteron with 4GB RAM. This machine has a 10GE PCI-X based Intel SR network interface. The maximum achievable throughput is around 7.5Gbps due to the PCI-X bandwidth.  Figure 5 compares the performance of single stream data transfer using TCP, Parallel TCP and Celeritas over the 10 Gbps network. In this experiment, the benchmark consisted of transferring 20MB of data repeated Celeritas is able to achieve a 12-fold performance improvement over TCP and Parallel TCP.  We would like to stress that the TCP bandwidth achieved is the maximum attained with all possible window and socket buffer sizes. Thus, Celeritas is able to exploit high-bandwidth networks and helps achieve performance over wide-area high bandwidth networks.

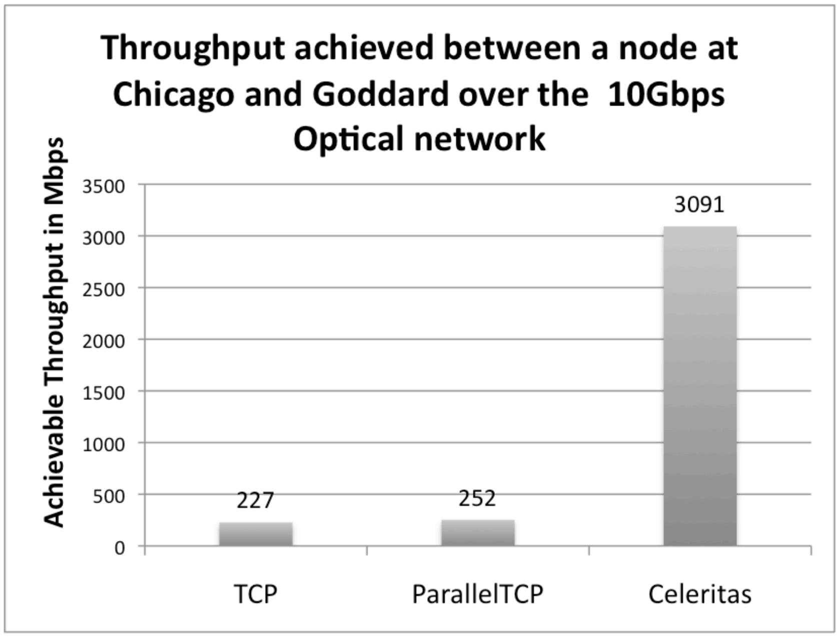

Figure 12: Performance comparison between TCP, Parallel  TCP and Celeritas

#### 4.2.2.5 Metadata Manager

The metadata manager maintains information about the datasets managed by LambdaRAM. It keeps track of information including the state of the datasets, associated dataset files and the state of the nodes. There is a single Metadata manager for each cluster in the LambdaRAM hierarchy, which runs on the node designated as the master in the LambdaRAM configuration. On the server cluster, the metadata manager maintains information about the relevant files that constitutes a dataset. This is provided as input via a XML configuration file. This helps to logically stitch together a dataset composed of several files. Thus, a scientist can manipulate a dataset as a single unit and need not have to deal with managing individual files that constitute a dataset. This enhances productivity as the scientist can focus his energies on the problem instead of data management. The information about logically stitching together a dataset from the individual files is cached on each node in the server cluster to improve data access latency.

### 4.2.3 I/O Abstraction Layer

The I/O abstraction layer enables efficient access to datasets in scientific data formats residing on storage systems. The design is extensible enables integration of LambdaRAM to support multiple data formats on various filesystems. This layer is composed of the filesystem abstraction layer and the data format abstraction layer.

The data format abstraction layer enables support for accessing datasets present in scientific data formats. Currently, the data format abstraction layer supports HDF4 [10], MAP-Binary (a custom data format for storing the data and metadata for the NASA MAP project), raw and binary data formats. Additional data formats, including NetCDF and HDF5, can be support with the help of plug-ins.

The filesystem abstraction layer enables the use of various file IO interfaces, including POSIX-based and high-performance parallel interfaces such as MPI-IO, to efficiently access files in high-performance filesystems including PVFS2 [PVFS] and GPFS [GPFS]. In the case of POSIX-IO, the implementation supports a thread pool to support parallel access to local data. This helps in improving the performance by taking advantage of disk parallelism prevalent in high performance storage solutions. Plugins for other distributed file-systems, including storage resource manager (SRM) and storage resource broker (SRB) could be designed in-order to interface LambdaRAM with these systems and access the data managed by these storage systems.

### 4.2.4 Additional LambdaRAM Features

#### Support for Multiple Datasets and Applications

LambdaRAM can manage multiple datasets in the cache simultaneously. This is important for data analysis applications that need to access several datasets to make more informed decisions. This is also important for applications such as remote visualization applications using multi-resolution data wherein each resolution is a unique dataset. Multiple applications can simultaneously use LambdaRAM. Figure XXXX depicts LambdaRAM supporting multiple concurrent applications and datasets.

### Memory Quality of Service (MemQoS)

LambdaRAM provides an application with Memory Quality of Service (MemQoS). An application can specify the amount of memory LambdaRAM can use on each node and for each dataset. MemQoS is useful for assigning priorities among datasets and caching frequently accessed datasets in memory. This is of importance to datacenters in-order to provide QoS assurances.

## 4.3. Typical Data Access in LambdaRAM

Figure 13 depicts how LambdaRAM satisfies an application's request for multi-dimensional data. It first maps the request onto relevant data blocks and checks to see if the blocks are cached. For all the uncached blocks, it computes the remote node from which it needs to fetch the block. It forwards this request to the remote node and waits for the blocks. LambdaRAM servers and peers satisfy a remote client's request as shown in Figure 14. We would like to note that the blocks could be pre-sent or being pre-fetched. Once, all the blocks are cached, it copies the relevant data into the application buffer.

In LambdaRAM, we have two methods to compute the home node of a block.

- The client is aware of the block distribution at the remote server and uses this to compute the home node for the block. This is the default heuristic in LambdaRAM. This could be augmented for fault-tolerance by having a dedicated node on the server end to query for the home node of a particular block. This scheme might help for local area networks, however, the inherent query latency would adversely affect the performance over wide-area.

- The client queries a local cluster node designated as the home node of the block. If the block is not present then the query gets forwarded to the block's home node at the remote server. This is called the local-caching mode in LambdaRAM and can be set in the LambdaRAM configuration file. This scheme could also be augmented by having a client broadcast the request to all nodes in the local cluster or by having a distributed hash table that keeps track of nodes caching a particular block.

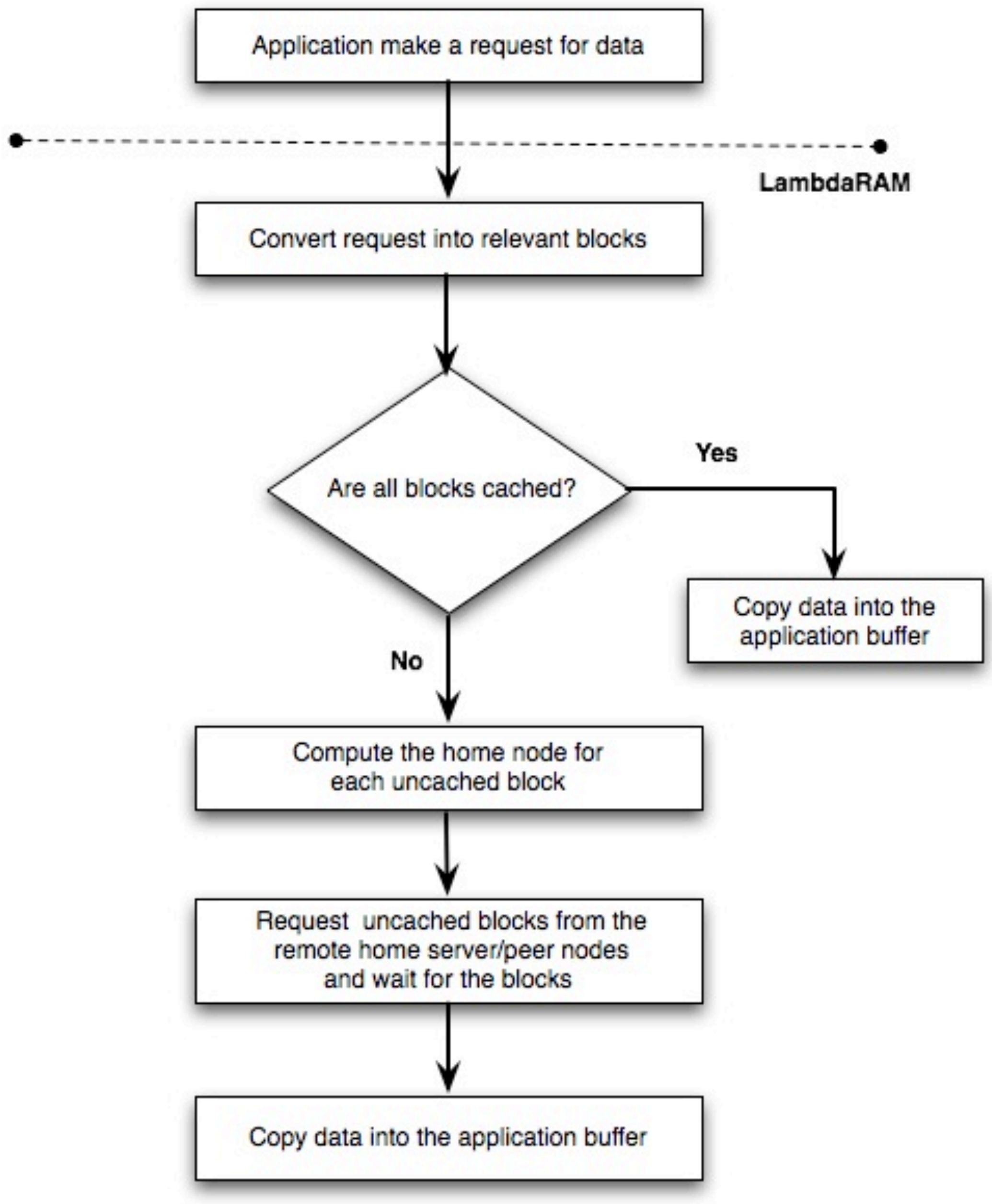
Application make a request for data
LambdaRAM
Convert request into relevant blocks
Are all blocks cached?
Yes
Copy data into the application buffer
No
Compute the home node for each uncached block
Request uncached blocks from the remote home server/peer nodes and wait for the blocks
Copy data into the application buffer

Figure 13: Application's Request Processing in LambdaRAM

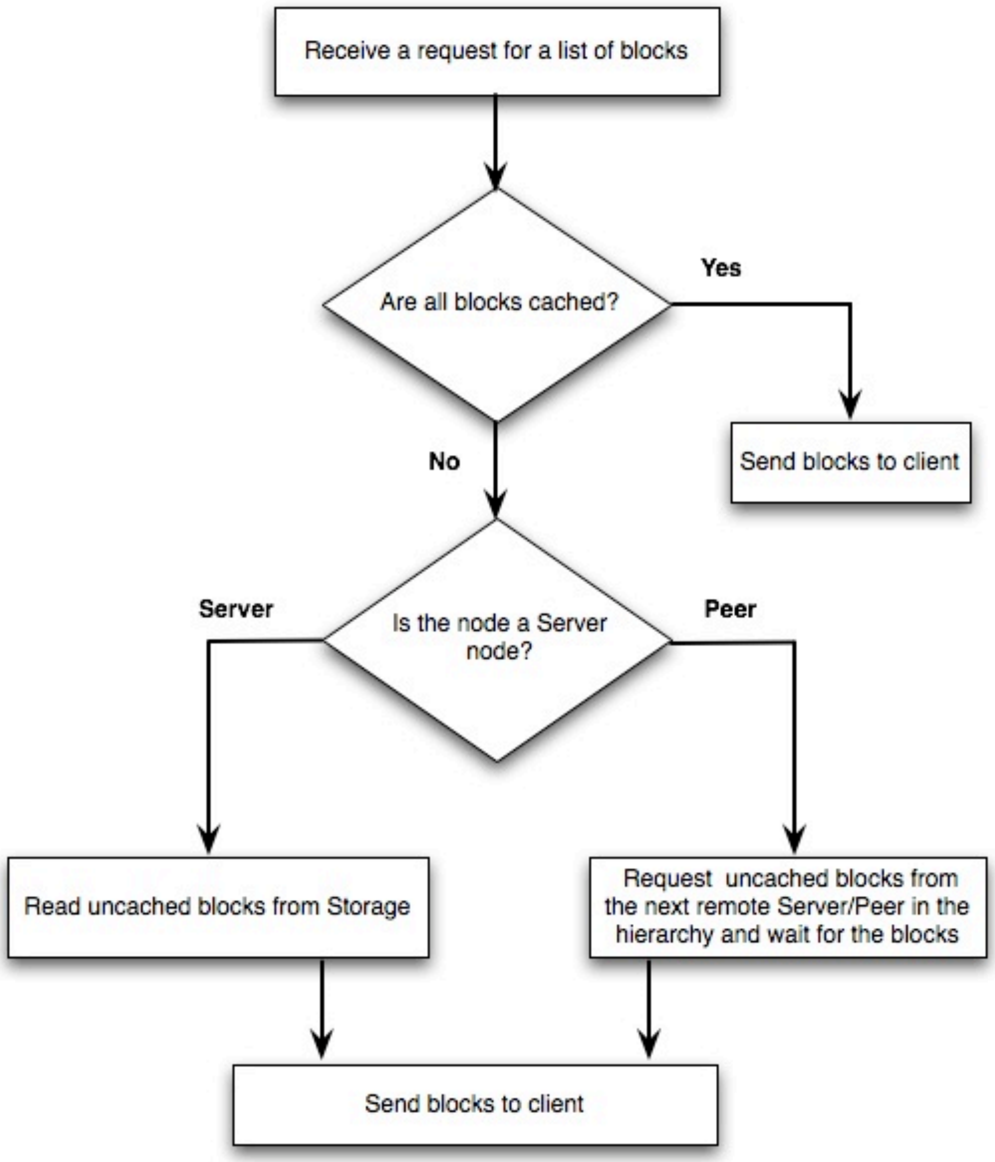
Receive a request for a list of blocks
Are all blocks cached?
Yes
Send blocks to client
No
Is the node a Server node?
Server
Peer
Read uncached blocks from Storage
Request uncached blocks from the next remote Server/Peer in the hierarchy and wait for the blocks
Send blocks to client

Figure 14: Processing a Remote Client's Request by LambdaRAM Servers

## 4.4. Putting it all together

LambdaRAM can be used by serial applications as well as parallel applications written using MPI. Figure 15 depicts a sample program using the LambdaRAM data access API and a parallel application in MPI using LambdaRAM for data access is depicted in Figure 16. These applications can use LambdaRAM to seamlessly access geographically distributed data.

```
#include "LRAM.h"
using namespace LRAM;

int main (int argc, char** argv)
{

        // Initializing LambdaRAM with relevant configuration
        LRAM::initialize (lram_conf_file))

        // Open a Dataset with relevant dataset meta-data
        int datasetID = LRAM::open (data_config_file);

        // Read the multi-dimensional array
        // In this case, we are reading a 3D data with starting extents
        // from st_ext with a length of len in each dimension into
        // a 1D buffer buffer
        long long st_ext[3], len[3];
        unsigned char* buffer;

        nread = LRAM::read (dsid, st_ext, len, buffer);

        // Close the Dataset
        LRAM::close(datasetID);

        // Finalize LambdaRAM
        LRAM::finalize())

}
```

Figure 15: A Typical Program using the LambdaRAM API

## Sample Parallel Program using LambdaRAM

```
#include "LRAM.h"
#include "mpi.h"
using namespace LRAM;

int main (int argc, char** argv)
{
        // Initialize MPI
        MPI_Init();

        // Initializing LambdaRAM with the relevant configuration file
        LRAM::initialize(lram_conf_file);

        // Open a Dataset with relevant dataset configuration file
        int datasetID = LRAM::open(dataset_conf_file);

        // Read the multi-dimensional array
        // In this case, we are reading a 3D data with starting extents
        // from st_ext with a length of len in each dimension into
        // a 1D buffer buffer
        long long st_ext[3], len[3];
        unsigned char* buffer;

        nread = LRAM::read (dsid, st_ext, len, buffer);

        // Close the Dataset
        LRAM::close(datasetID);

        // Finalize LambdaRAM
        LRAM::finalize())

        // Finalize MPI
        MPI_finalize();

        return 0;

}
```

Figure 16 A Parallel MPI Program using LambdaRAM API

# 5. EVALUATION

In this chapter, we evaluate the performance of LambdaRAM using micro-benchmarks and application-level benchmarks over local-area, metropolitan area, and wide-area networks. We present micro-benchmarks in Section 5.1 and application-level benchmarks using NASA's climate analysis applications together with large datasets in Section 5.2.

## 5.1. <u>Micro-Benchmarks</u>

The commonly used data access patterns in data-intensive computing are sequential, consecutive and striding patterns, and, are depicted in Figure 17, Figure 18 and Figure 19 respectively. Sequential data access is commonly used in applications including data analysis, visualization and data transfers. In sequential access pattern, in the $i^{th}$ iteration, node 0 reads data starting from the location where node n-1 (assuming n nodes) finished reading in the iteration $i-1$. Consecutive data access is commonly used in data visualization when a user pans around the data. In consecutive access pattern, in the $i^{th}$ iteration, node 0 reads the data node 1 (assuming N nodes) read in iteration $i-1$. Data striding is commonly used in data-intensive HPC applications, including climate analysis and astrophysics.

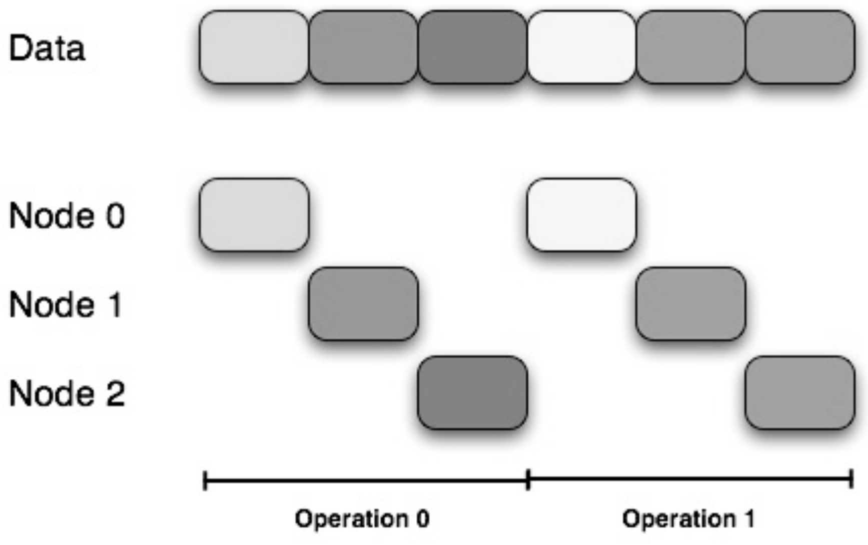

Figure 17: Sequential Data Access Pattern for Parallel Applications

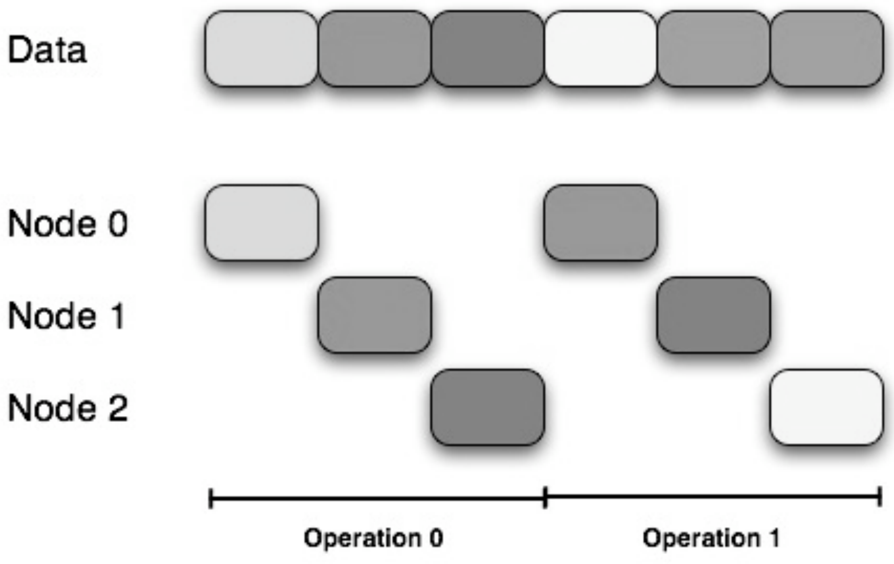

Figure 18: Consecutive Data Access Pattern for Parallel Applications

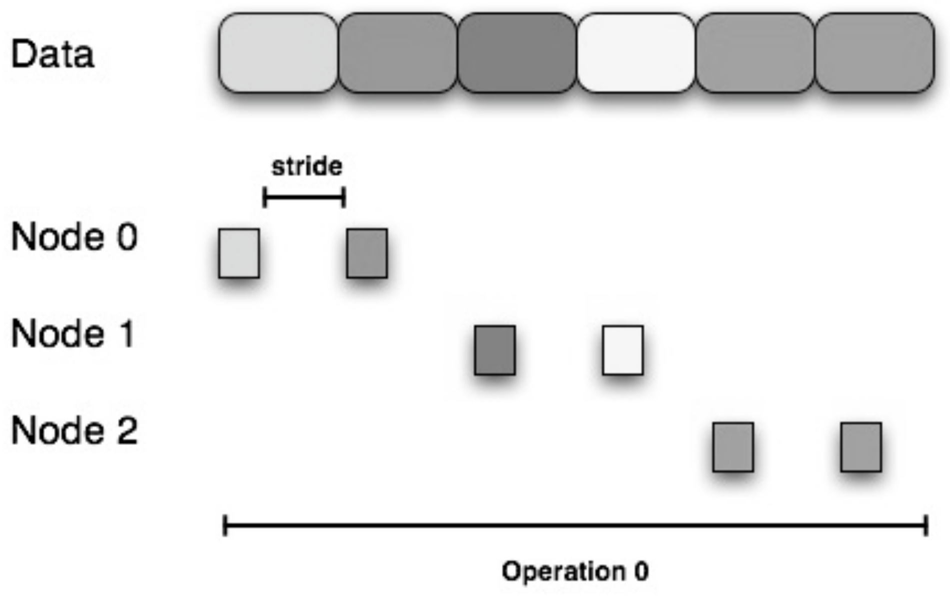

Figure 19: Strided Data Access Pattern for Parallel Applications

We designed a micro-benchmark to evaluate the performance of LambdaRAM for applications exhibiting the above access patterns. The benchmark is a parallel application designed using MPI. It supports multi-dimensional sequential, strided and consecutive data access patterns and has multiple configurable parameters to enable us evaluate the performance of LambdaRAM in various scenarios. In case of sequential and consecutive accesses, the configurable parameters include the amount of data read per iteration, the number of iterations and number of nodes. In case of strided accesses, one can configure the stride length, the amount of data read, number of iterations and number of nodes.

For our experiments, we used a 200GB 3D dataset comprising of 20 files that were 10GB each. The dataset was stored on a 17 TB PVFS2 version 2.6 parallel filesystem across 28 nodes of a 30-node "Yorda" cluster at EVL, UIC. The cluster nodes consisted of 64-bit dual processor 2.4Ghz AMD Opterons with 4GB RAM. We dedicated a gigabit Ethernet network interface card on each node for the PVFS2 traffic. The nodes were interconnected via a high-performance Cisco 3750 switch with 96 Gbps bisection bandwidth. The dataset was also replicated on a 250GB SATA II disk on each node of the cluster configured with XFS filesystem. The benchmark application was run on 8 nodes.

We compare the following scenarios for accessing data:

- The parallel benchmark using POSIX I/O with scatter-gather optimization to access data on local storage. We would like to remind that in this case the data is replicated on a disk on each node. Replicated storage is not a scalable solution. However, it gives us an estimate of the achievable performance with raw storage.

- The parallel benchmark using MPI-IO to access data on the PVFS2 partition distributed across the 28 nodes of the cluster.

- The parallel benchmark using Local LambdaRAM to access data. Local LambdaRAM refers to the case where the application and data are co-located on a cluster, and, LambdaRAM spans the memory of the cluster nodes on which the application runs and manages the data accesses for the benchmark. In this case, LambdaRAM accesses the data in following two ways:

1. Using POSIX I/O to access data replicated on local storage on each node.
2. Using MPI-IO to access data on the PVFS2 partition.

- The parallel benchmark access data using the Client-Server configuration of LambdaRAM to access data. Client-Server LambdaRAM refers to the case of the application and data residing on different clusters with the application using LambdaRAM to access the remote data. LambdaRAM spans the memory of both clusters. In this case, the application and Client LambdaRAM was run on a 8 node 32-bit dual-processor Intel Xeon cluster with 1 Gigabit Ethernet NIC. The Server LambdaRAM was run on the Yorda cluster. The Client and Server clusters are co-located in a machine room and interconnected using a 2x10Gbps optical link. The Server LambdaRAM accesses the data, similar to the Local LambdaRAM, in following two ways:

    1. Using POSIX I/O to access data replicated on local storage on each node.
    2. Using MPI-IO to access data on the PVFS2 partition.

### 5.1.1 Sequentially Accessing a 3D Dataset over Local Area Networks

We evaluate the performance of the above-mentioned scenarios to sequentially access the 200GB dataset. The benchmark application was configured to access 20MB per request. LambdaRAM was configured with an apriori prefetching heuristic. From the Figure 20, we see that replicated storage, as expected, performs better than PVFS2. Both Local LambdaRAM and Client-Server LambdaRAM improve on the performance of PVFS2 and Replicated Storage. LambdaRAM achieves upto a four-fold improvement in performance over PVFS2 and close to a two-fold improvement over local storage. This improvement is due to the multi-dimensional data management, caching and prefetching heuristics of LambdaRAM.

The Client-Server LambdaRAM demonstrate an improvement of 10.2% and 19.4% over Local LambdaRAM for PVFS2 and replicated storage cases respectively. The performance increase is for the very same application running on an identical number of nodes. This improvement is due to the fact that in case of Client-Server LambdaRAM the working-set size is larger than Local LambdaRAM as more nodes are involved. Thus, having dedicated Server cache nodes (or Peer cache nodes) reduces the load on the client nodes where the application is running. Additionally, the complex latency mitigation heuristics can be offloaded onto the Server keeping the overhead of LambdaRAM on the clients low.

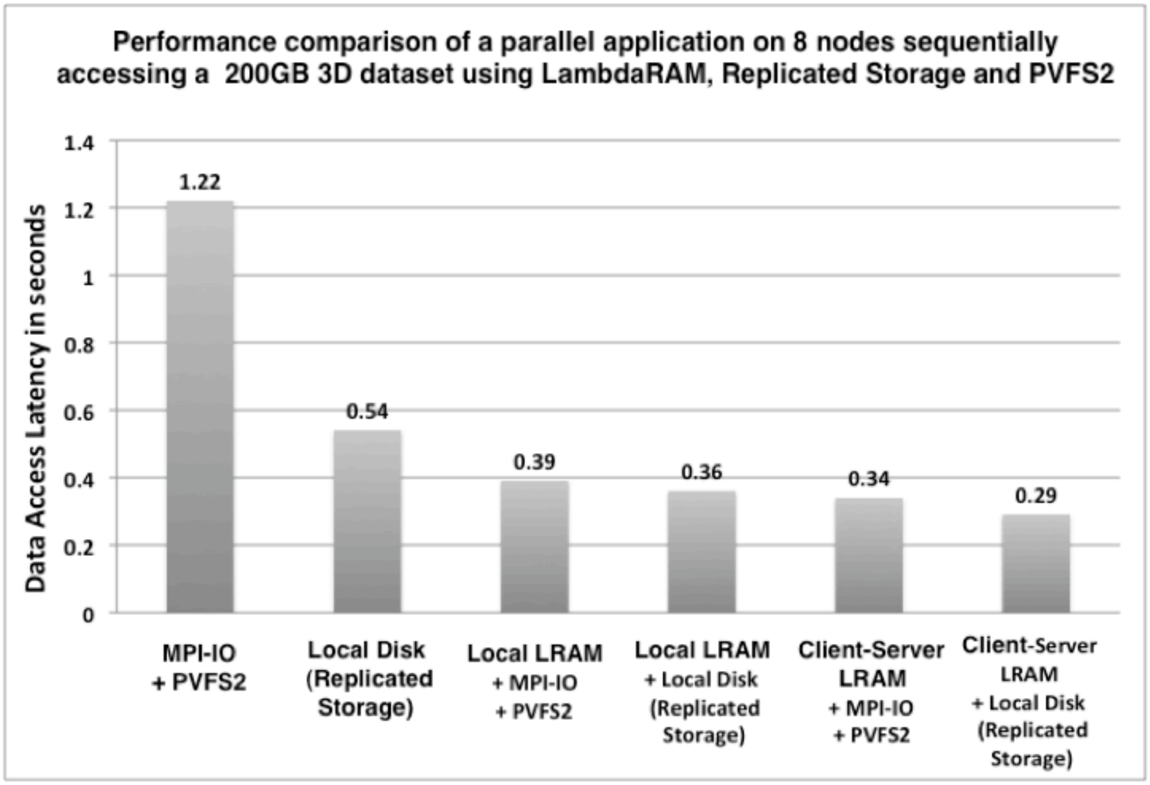

Figure 20: Performance Evaluation of LambdaRAM sequentially accessing a 3D dataset

### 5.1.2 Consecutive Data Access of a 3D dataset over Local Area Networks

We evaluate the performance of the above-mentioned scenarios to consecutively access the 200GB dataset. The benchmark application was configured to access 20MB per request. LambdaRAM was configured with an apriori prefetching heuristic. From the Figure 21, we see that replicated storage, as expected, performs better than PVFS2. Both Local LambdaRAM and Client-Server LambdaRAM improve on the performance of PVFS2 and Replicated Storage. LambdaRAM achieves upto a four-fold improvement in performance over PVFS2 and a two-fold improvement over local storage. This improvement is due to the multi-dimensional data management, caching and prefetching heuristics of LambdaRAM.

The Client-Server LambdaRAM demonstrate an improvement of 17.6% and 22.6% over Local LambdaRAM for PVFS2 and replicated storage cases respectively. The performance increase is for the very same application running on an identical number of nodes. This improvement is due to the fact that in case of Client-Server LambdaRAM the working-set size is larger than Local LambdaRAM as more nodes are involved. Thus, having dedicated Server cache nodes (or Peer cache nodes) reduces the load on the client nodes where the application is running. Additionally, the complex latency mitigation heuristics can be offloaded onto the Server keeping the overhead of LambdaRAM on the clients low.

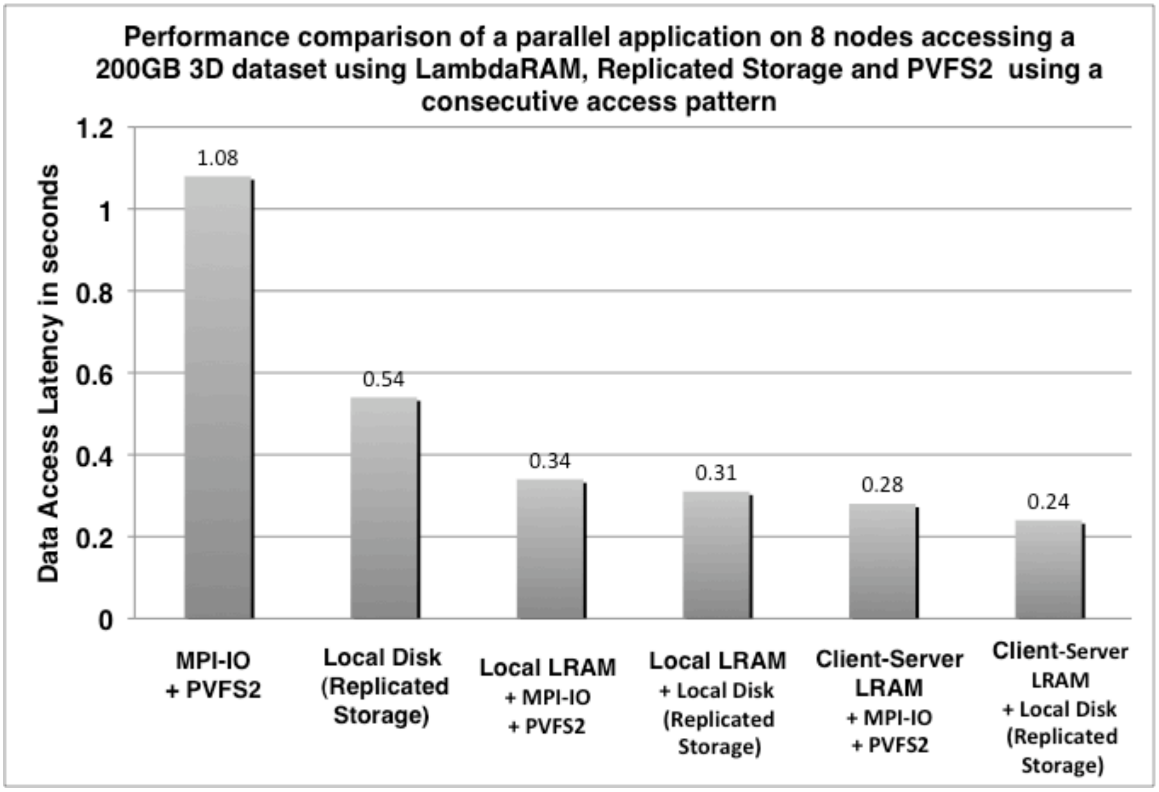

Figure 21: Performance Evaluation of LambdaRAM consecutively accessing a 3D Dataset

### 5.1.3 Striding Through a 3D dataset over Local Area Networks

We evaluate the performance of the above-mentioned scenarios to rapidly stride through the 200GB dataset. The benchmark application was configured to access 20MB per request. LambdaRAM was configured with an apriori prefetching heuristic. From the Figure 22, we see that replicated storage, as expected, performs better than PVFS2. Both Local LambdaRAM and Client-Server LambdaRAM improve on the performance of PVFS2 and Replicated Storage. LambdaRAM achieves upto a four-fold improvement in performance over PVFS2 and a two-fold improvement over local storage. This improvement is due to the multi-dimensional data management, caching and prefetching heuristics of LambdaRAM.

The Client-Server LambdaRAM demonstrate an improvement of 2% and 23% over Local LambdaRAM for PVFS2 and replicated storage cases respectively. The performance increase is for the very same application running on an identical number of nodes. This improvement is due to the fact that in case of Client-Server LambdaRAM the working-set size is larger than Local LambdaRAM as more nodes are involved. Thus, having dedicated Server cache nodes (or Peer cache nodes) reduces the load on the client nodes where the application is running. Additionally, the complex latency mitigation heuristics can be offloaded onto the Server keeping the overhead of LambdaRAM on the clients low. We notice a lower performance decrease with PVFS2 to rapidly stride through multi-dimensional datasets. We believe, as the IO-rates are high, the overhead of collective operations affects the performance.

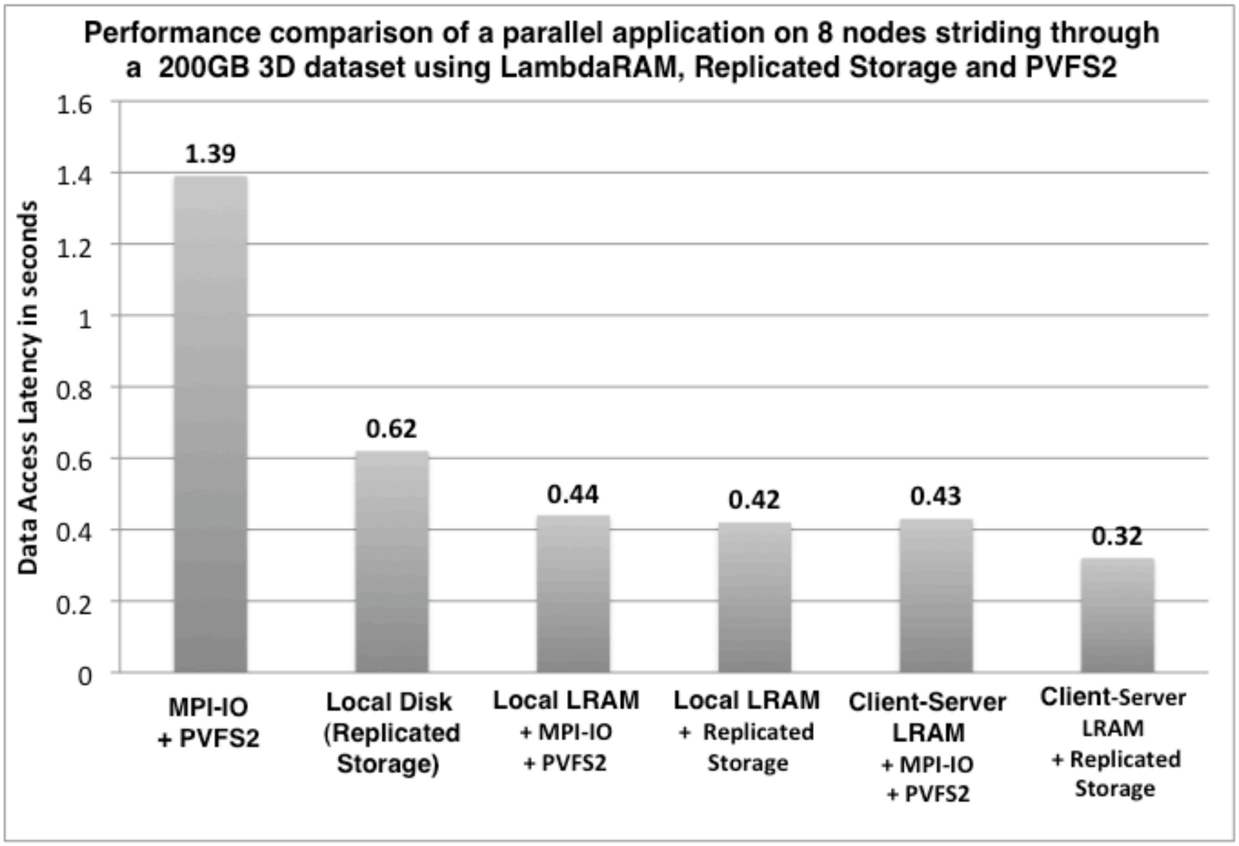

Figure 22: Performance Evaluation of LambdaRAM Rapidly Striding Through a 3D Dataset

## **Efficacy of the prefetching to proactively fetch data**

Scientific applications typically exhibit periodic computation and I/O phases. Thus, A key goal of LambdaRAM is to prefetch data while the application is performing the computations. This would help reduce access latencies and in case of NASA's climate simulations keep the computation busy by proactively fetching data for the simulations before it is needed. In this experiment, we introduced a sleep of 1 sec between consecutive data IO operations to simulate the compute phase. From the Figure 23, we see significant reductions in data access latency using LambdaRAM for sequential, consecutive and strided access patterns. Thus, prefetching is very useful in proactively accessing data while a computation is taking place and helps reduce data-access latency.

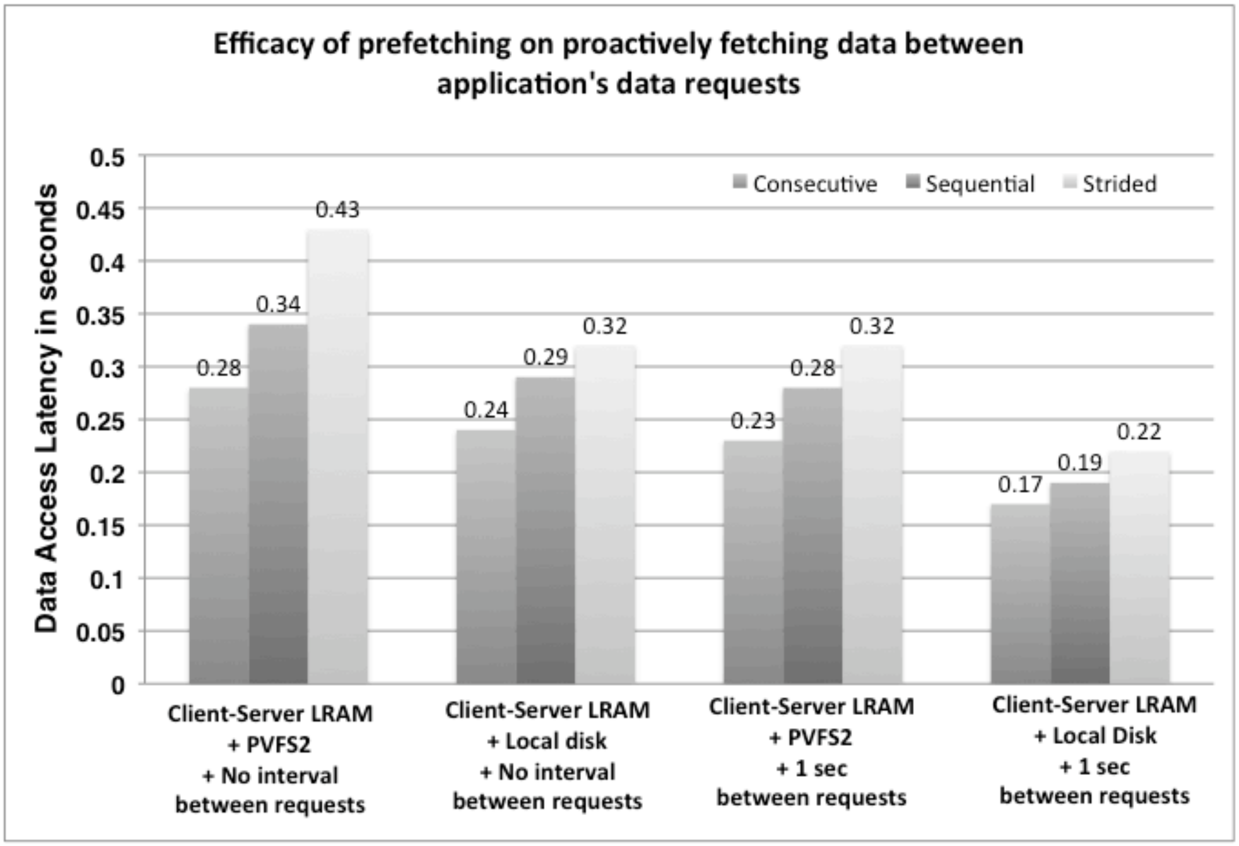

Figure 23: Efficacy of Data Pre-fetching in LambdaRAM

## 5.2. Application-Level Benchmarks

We evaluate the efficacy of LambdaRAM with climate analysis applications over local-area, metropolitan area and wide-are networks. In section 5.2.1, we evaluate the performance of LambdaRAM to compute wind shear for NASA's MAP 2006 project. We present performance evaluation of climate analysis applications to compute average ozone thickness and surface temperature using for NASA MERRA data using LambdaRAM over wide-area networks in section 5.2.2.

### 5.2.1 Wind Shear Computation of NASA's Modeling Analysis and Prediction (MAP) 2006 Project Data Over LAN, MAN and WAN

In the summer of 2006, The Earth-Sun Exploration Division of Goddard Space Flight Center (GSFC) and the Science and Mission Systems Office at Marshall Space Flight Center brought together resources from NASA and from corporate partners to study tropical cyclones. The primary objective of MAP '06 was the application of NASA's advanced satellite remote sensing technologies and earth system modeling capabilities to improve the understanding and prediction of tropical cyclones that develop in and move across the Atlantic basin. This project began in the early portion of the 2006 hurricane season and continued through late autumn. MAP '06 implemented the Goddard Earth Observing System (GEOS5), the fifth-generation global atmospheric model and the Grid point Statistical Interpolation (GSI) data analysis system. In addition, the ability of GEOS5 to initialize the Weather Research and Forecast (WRF) regional model was evaluated. The data from the model runs were used to analyze cyclone and hurricane formation with the goal of improving future hurricane forecast systems. A critical factor affecting the formation and destruction of hurricanes and cyclones is wind shear. Wind shear could be defined as the vector difference between the wind velocities at 850mb and 200mb pressure heights in the atmosphere. In the case of hurricanes, wind shear is important primarily in the vertical direction and gives us a deep insight into the strength of a hurricane. A high wind shear value results in increased latent heat dissipation that reduces the strength of a hurricane over time. A lower wind shear results in a hurricane of higher intensity. Rapidly striding over remote MAP '06 datasets to compute wind shear would enable earth scientists to efficiently analyze and predict tropical cyclones and hurricanes.

The wind shear computation application is written in C++. It runs on a single node and strides over the MAP'06 data to compute the wind shear. The MAP'06 datasets are multi-dimensional and stored in a MAP-Binary format. The MAP-Binary is a customized format for MAP data and stores the data along with the associated metadata in a big-endian binary format. MAP'06 datasets from August 15 2006 to August 31 2006 were used for the experiments. This dataset was approximately 250 GB and consisted of 21 files for each day and a total of 357 files for the 17 days. The datasets are 4 dimensional with the dimensions being time, pressure levels, latitude and longitude. The wind shear computation application uses POSIX IO to read the MAP-Binary dataset. We also modified this application to use the LambdaRAM API to access the data. We would like to note that LambdaRAM supports MAP-Binary format via a data format abstraction layer plug-in.

We evaluate the performance of striding and computing the wind shear of the 4D MAP’06 dataset using LambdaRAM over high-speed networks. The experiments included striding over the entire world data, a critical component for global models, and regional striding such as striding over the Atlantic basin, critical for analysis of tropical cyclones and hurricanes developing in the Atlantic basin. We present experimental evaluation of LambdaRAM to stride based on time and based on geographical co-ordinates to compute wind shear for the MAP’06 data.

#### 5.2.1.1 Striding and computing wind shear using LambdaRAM and Parallel Filesystem

In this experiment, the 250 GB MAP’06 dataset was stored on a 17 TB PVFS2 parallel filesystem across 28 nodes of a 30-node cluster at EVL, UIC. The cluster nodes consisted of 64-bit dual processor 2.4Ghz AMD Opterons with 4GB RAM and a 1 GigE network interface card (NIC). The nodes were interconnected via a high-performance Cisco 3750 switch with 96 Gbps bisection bandwidth. The wind shear computation application was run on one node of the cluster. We compare the performance of computing wind shear by striding over the data residing in PVFS2 to striding over the same data using LambdaRAM. We would like to note that in our experiments we read large unrelated datasets between each experimental run to mitigate the effects of the filesystem cache on the results. We first discuss the performance of striding, based on time, to compute the wind shear for the entire earth. Subsequently, we present results on striding based on geographical co-ordinates and time to compute the wind shear for the Atlantic basin.

## **Striding and computing the wind shear for the entire earth**

We evaluate the performance of striding, based on time, to compute the wind shear for the entire earth. From Figure 24, we see that a single LambdaRAM server yields a 100% speedup over the performance of striding using PVFS2. This is mainly due to the multi-dimensional data management and prefetching based on the application's access patterns in LambdaRAM. In case of PVFS2, the filesystem deals with individual files while LambdaRAM treats the entire set of files a single dataset and manages it collectively. We observe a linear speedup as we increase the number of LambdaRAM servers from a single server to four servers. This is due to the increase in the available memory for caching data as we increase the number of servers. We observe a five-fold performance improvement using four servers. Increasing the number of LambdaRAM servers beyond four servers does not result in an increased speed-up due to the saturation of the 1 Gbps network bandwidth of the client node. Thus, LambdaRAM, with its multi-dimensional data management and latency mitigation heuristics, can help improve the performance of scientific applications accessing data residing in a parallel filesystem such as PVFS2.

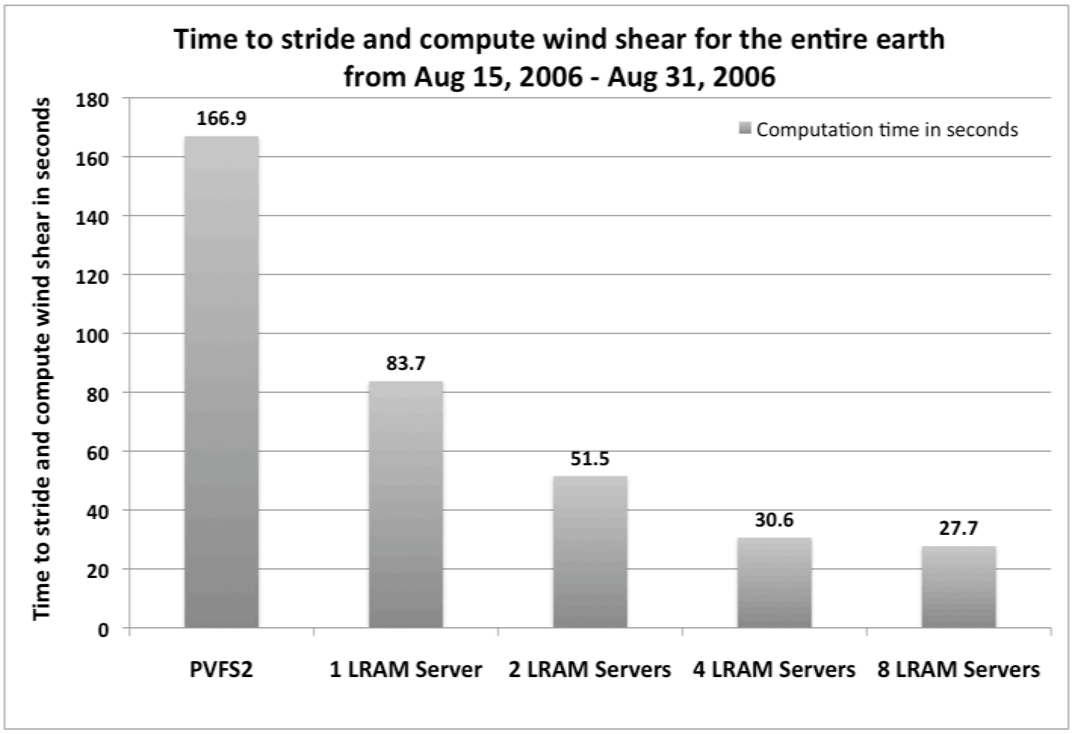

Figure 24: Wind shear computation for the entire world using LambdaRAM on a LAN
Lower computation time indicates improved performance. Results show a linear speedup as we scale the number of LambdaRAM servers to four. The bottleneck beyond four servers is due to the saturation of the 1Gbps link of the client node.

## Striding and computing the wind shear for the Atlantic basin

We evaluate the performance of striding, based on geographical co-ordinates and time, to compute the wind shear for the Atlantic basin. We note that striding using PVFS2 through the 4D MAP'06 dataset for the Atlantic basin involves accessing multiple noncontiguous regions, which incurs a lot of overhead. Hence, even though the Atlantic basin is a subset of the entire earth, we observe from Figures 25 that with PVFS2 it takes more time to compute the wind shear for the Atlantic basin than the entire earth. In case of LambdaRAM, as the data is cached in the memory of the servers and it takes less time to compute the wind shear for the Atlantic basin than the entire earth as we access less data from remote memory. Additionally, in PVFS2, the filesystem deals with individual files while LambdaRAM treats the entire set of files a single dataset and manages it collectively. From Figure 25, we see that a single LambdaRAM server yields a five-fold speedup over the performance of striding using PVFS2. With 4 servers, we are able to achieve a twenty-fold performance improvement. Increasing the number of LambdaRAM servers beyond four servers does not result in an increased speed-up due to the saturation of the 1 Gbps network bandwidth of the client node. Thus, LambdaRAM, with its multi-dimensional data management and latency mitigation heuristics, can help improve the performance of scientific applications striding through data over many dimensions in a parallel filesystem such as PVFS2.

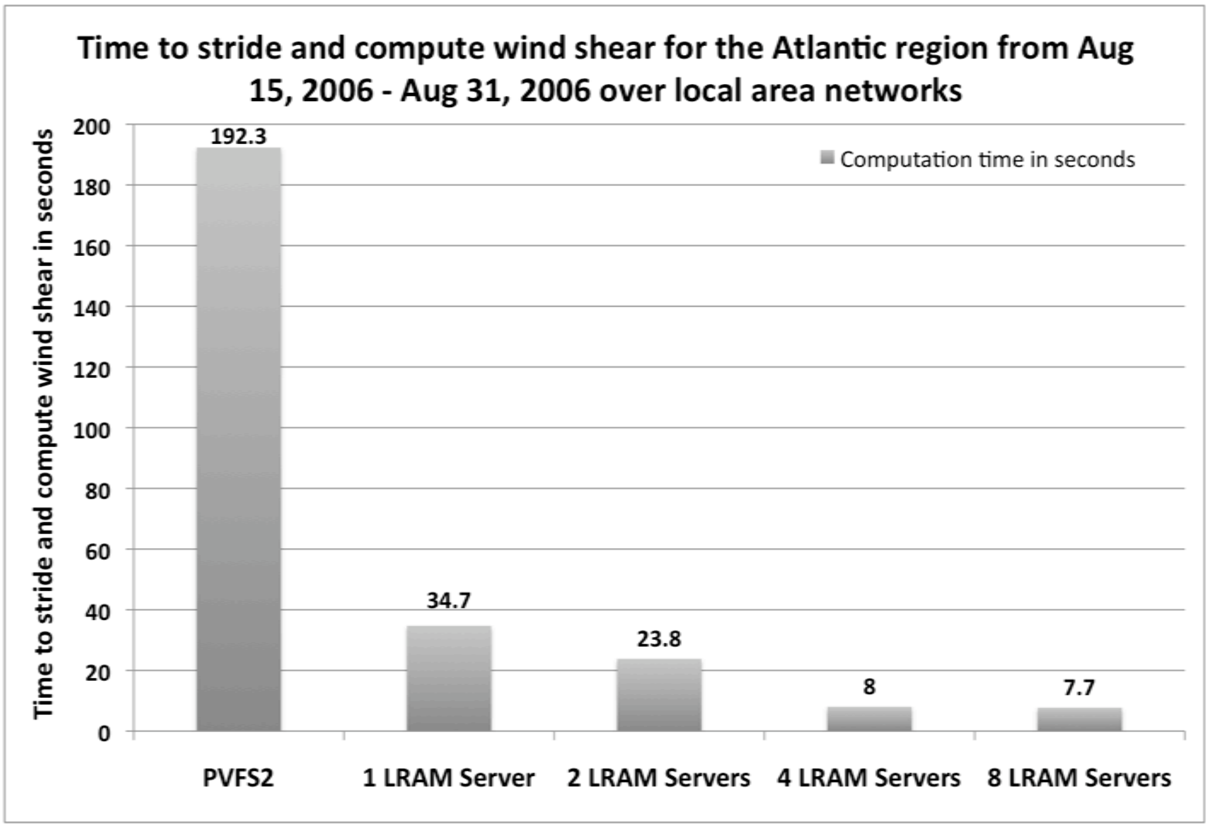

Figure 25: Wind Shear Computation for the Atlantic Basin using LambdaRAM on a LAN

Lower computation time indicates improved performance. Results show a speedup of twenty-fold as we scale the number of LambdaRAM servers to four. The bottleneck beyond four servers is due to the saturation of the 1Gbps link of the client node.

#### 5.2.1.2 Performance comparison between LambdaRAM over metropolitan area high-speed network (MAN) and an ultra-fast storage system

We evaluate the performance of striding and computing wind shear for the 250 GB MAP’06 data stored locally on an ultra-fast “state-of-the-art” storage system with accessing this data remotely, using LambdaRAM, over a high-speed optical network. The MAP’06 datasets were stored on an ultra-fast multi-terabyte storage system located at the Starlight facility [12] in downtown Chicago. The storage system consists of quad processor Intel 3.2Ghz Xeon with 24 SATA-1 disks configured in RAID5 using three hardware PCI-X RAID cards and XFS filesystem. In the LambdaRAM case, a client-server cluster configuration was used. The LambdaRAM server configuration was run on the ultra-fast storage system at Starlight while the client configuration and the wind-shear computation application were run on a cluster node at EVL. The cluster node consisted of 64-bit dual processor 2.4 GHz AMD Opterons with 4 GB RAM and a 1 GigE NIC. EVL is connected to Starlight via 2 x 10 Gbps Optical Network; however, the effective bandwidth between the client node and the Storage server was limited to 1 Gbps due to the 1 GigE NIC on each system. In the “ultra-fast storage system” case, the wind shear application is run locally on the storage system and accesses data using POSIX IO. We would like to note that in our experiments we read large unrelated datasets between each experimental run to mitigate the effects of the filesystem cache on the results. We first discuss the performance of striding, based on time, to compute the wind shear for the entire earth. Next, we present results on striding, based on geographical co-ordinates and time, to compute the wind shear for the Atlantic basin.

**Striding and computing the wind shear for the entire earth**

In Figure 26, we compare the performance of striding, based on time, to compute the wind shear on the ultra-fast storage system with the same using LambdaRAM over ultra-fast networks. We observe that it takes less time to remotely stride and compute the wind shear using LambdaRAM than computing it locally on the ultra-fast storage system. The increase in performance using LambdaRAM is due its efficient multi-dimensional data caching, management and latency mitigation heuristics to mitigate access latencies. The current bottleneck in LambdaRAM's performance is due to the saturation of the 1 Gbps network bandwidth of the client node. With 10GE NIC becoming increasingly ubiquitous, the performance of computing wind shear with LambdaRAM for remote data will most likely increase.

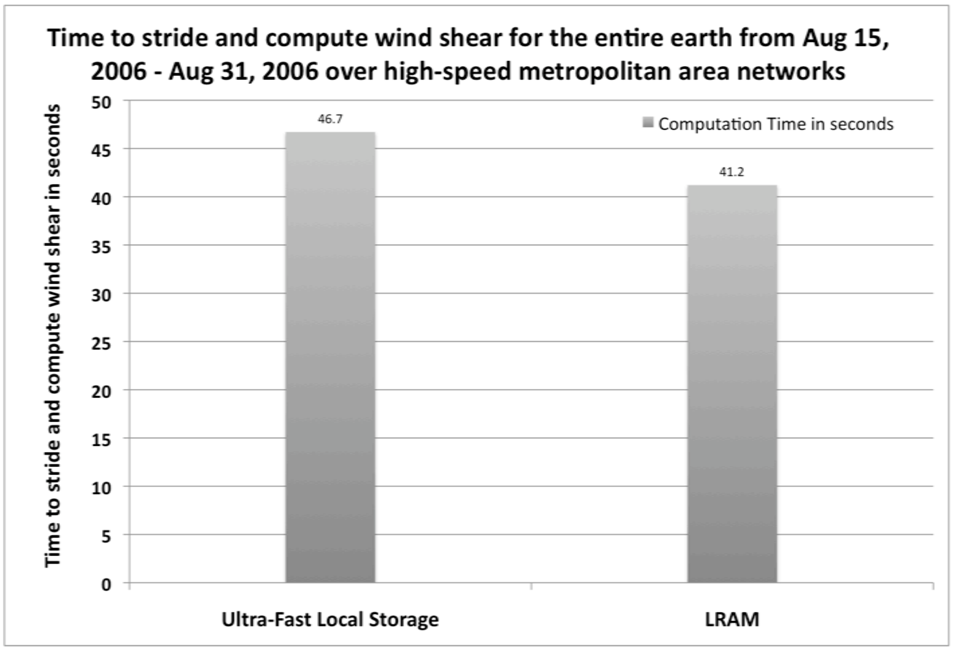

Figure 26: Wind Shear Computation for the Entire World using LambdaRAM on a MAN
Wind shear computation for the entire world on an ultra-fast storage system and using LambdaRAM between EVL and Starlight over 1Gbps Networks. Lower computation time indicates improved performance.

**Striding and computing the wind shear for the Atlantic basin**

We compare the performance of striding, based on geographical co-ordinates and time, to compute the wind shear locally on the ultra-fast storage system with the same using LambdaRAM over ultra-fast networks. We observe from Figures 26 and 27 that in case of the ultra-fast storage system it takes longer to compute the wind shear for the Atlantic basin than the entire earth. This is due to the fact that striding through the 4D MAP’06 dataset over the Atlantic basin involves accessing multiple noncontiguous regions on disk that incurs a lot of overhead. From Figure 27, we observe a two-fold speed up with LambdaRAM in comparison to the ultra-fast storage system. The increase in performance using LambdaRAM is due its efficient multi-dimensional data caching, management and prefetching data to overcome access latencies. Thus, we observe that it takes less time to remotely stride over multiple dimensions and compute the wind shear using LambdaRAM than computing it locally on the ultra-fast storage node.

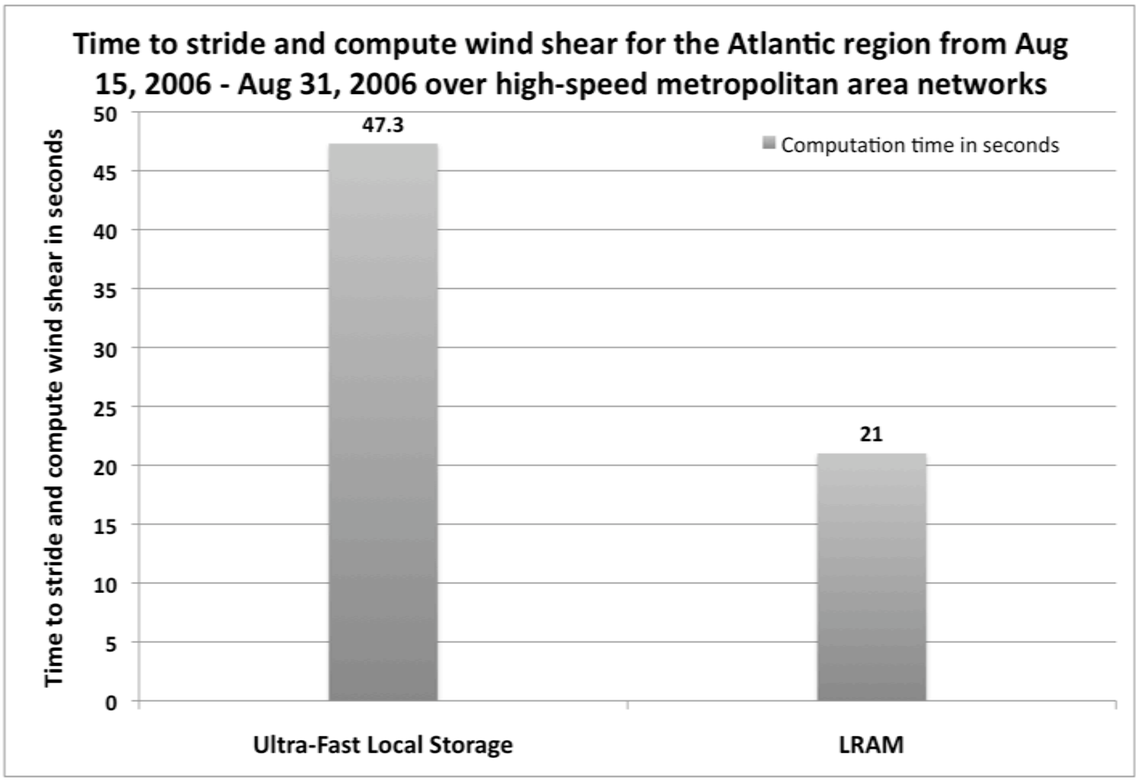

Figure 27: Wind Shear Computation for the Atlantic Basin using LambdaRAM on a MAN
Wind shear computation for the Atlantic basin on an ultra-fast storage system and using LambdaRAM between EVL and Starlight over 1Gbps Networks. Lower computation time indicates improved

### 5.2.2 Climate data analysis of NASA's Modern era retrospective analysis for research and applications (MERRA) dataset Over LAN, MAN and WAN

Retrospective-analysis of weather data is a key element in understanding climate variability. The Modern Era Retrospective-analysis for Research and Applications (MERRA) was developed to support NASA's Earth science objectives, by applying the state-of-the-art global modeling and assimilation office (GMAO) data assimilation system that includes many modern observing systems in a climate framework. Reanalysis blends the continuity and breadth of output data of a numerical model with the constraint of vast quantities of observational data. The result is a long-term continuous data record. The MERRA time period covers the modern era of remotely sensed data for the entire earth, from 1979 through the present. MERRA data is stored at NASA GSFC and NASA hopes to make this available to researchers in 2009. The data is published in HDF4 scientific data format, and, is currently around 700TB. MERRA data consists of multi-dimensional 2D, 3D, 4D variables. Given the size of the dataset, replicating this dataset at other sites incurs a heavy cost. Additionally, researchers need to modify this dataset and replicating this data leads to data consistency issues. NASA would like to enable researchers at remote sites seamlessly access this data located at Goddard and use the data for their weather analysis. LambdaRAM would enable applications of NASA and their collaborations at various sites to seamlessly access the remote MERRA data. Rapidly striding over the MERRA data using LambdaRAM would enable earth scientists to make timely and informed tropical cyclones and hurricane prediction.

We evaluate the performance of a weather analysis application accessing MERRA data using LambdaRAM over wide-area networks. We transferred MERRA data for 1979 from the repository at NASA Goddard to a storage system at EVL, Chicago. This dataset consists of 6 hour reading for each day of the year. The dataset was around **1TB** comprising of 365 * 4 = 1440 data files in HDF4 format. Each file consists of 16 variables including wind shear, ozone thickness and surface temperature. The dataset was stored on an ultra-fast storage system at EVL Chicago. The storage system consists of a dual-processor dual-core AMD Opteron system with 4GB of RAM, 2TB of storage and a 10GE Myricom PCI-e based NIC. The 2TB storage was configured using eight 300 GB SATA – II disks on a PCI-e based 8-port 3ware RAID controller using RAID 0. The node connects to the wide-area using the CAVEWave and TeraFlow network at 10Gbps. The analysis was performed on a node at NASA Goddard consists of a dual-core dual-processor AMD Opteron with 4GB RAM. This machine has a 10GE PCI-X based Intel SR network interface. The maximum achievable throughput is approximately 7.5Gbps due to the PCI-X bandwidth limitation. A dedicated 10 Gbps network was provisioned for the experiment. The network comprised of the CAVEWave network from Chicago to McLean, Virginia, the DRAGON network between McLean, VA and NASA Goddard, and the Teraflow network from McLean to Chicago. The round trip latency (RTT) for this network is around 19ms. Figure 28 depicts the experimental setup where the data was located in Chicago, the analysis application was run at NASA Goddard and the results of the analysis was streamed and visualized in real-time at Chicago.

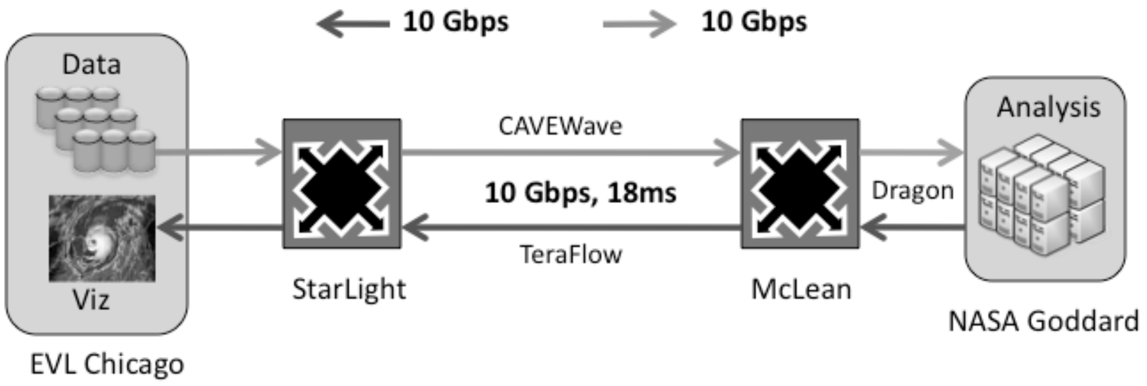

Figure 28: Wide Area Experimental Testbed

Experimental Testbed to evaluate the performance of rapidly striding and analyzing remote MERRA data using LambdaRAM over a 10Gbps Wide Area Network

## **Striding and computing the average ozone thickness for MERRA data using LambdaRAM over wide-area networks**

Climate scientists routinely compute the average ozone thickness. Accelerating this computation is critical for timely hurricane and tornado prediction. Ozone thickness is a 4D dataset consisting of a time-series of floating point values for the various atmospheric levels for each earth grid point. Computing the average surface temperature involves multi-dimensional striding over 4-D datasets. The ozone thickness computation application is written in C++ and runs on a single node We compare the performance of striding and computing the average ozone thickness for the MERRA data for 1979 locally on the ultra-fast storage system with striding over the data using LambdaRAM over wide-area networks between EVL Chicago and NASA Goddard.

From Figure 29, we observe upto a 40% improvement in performance using LambdaRAM with Celeritas as the data transport protocol over the ultra-fast local storage system. On the Local storage, striding through multi-dimensional data involves accessing multiple noncontiguous regions on disk and incurs a lot of overhead and leads to performance degradation. Pre-sending yields a better performance than pre-fetching as the data striding access patterns, as it does not incur the request latency of prefetching.  Additionally, Celeritas plays a key role in achieving high-performance over the Wide-area networks. Thus, we observe that it takes less time to remotely stride over multiple dimensions and compute the average ozone thickness using LambdaRAM than computing it locally on the ultra-fast storage node. Celeritas and latency mitigation heuristics enable LambdaRAM to achieve high performance over wide-area networks.

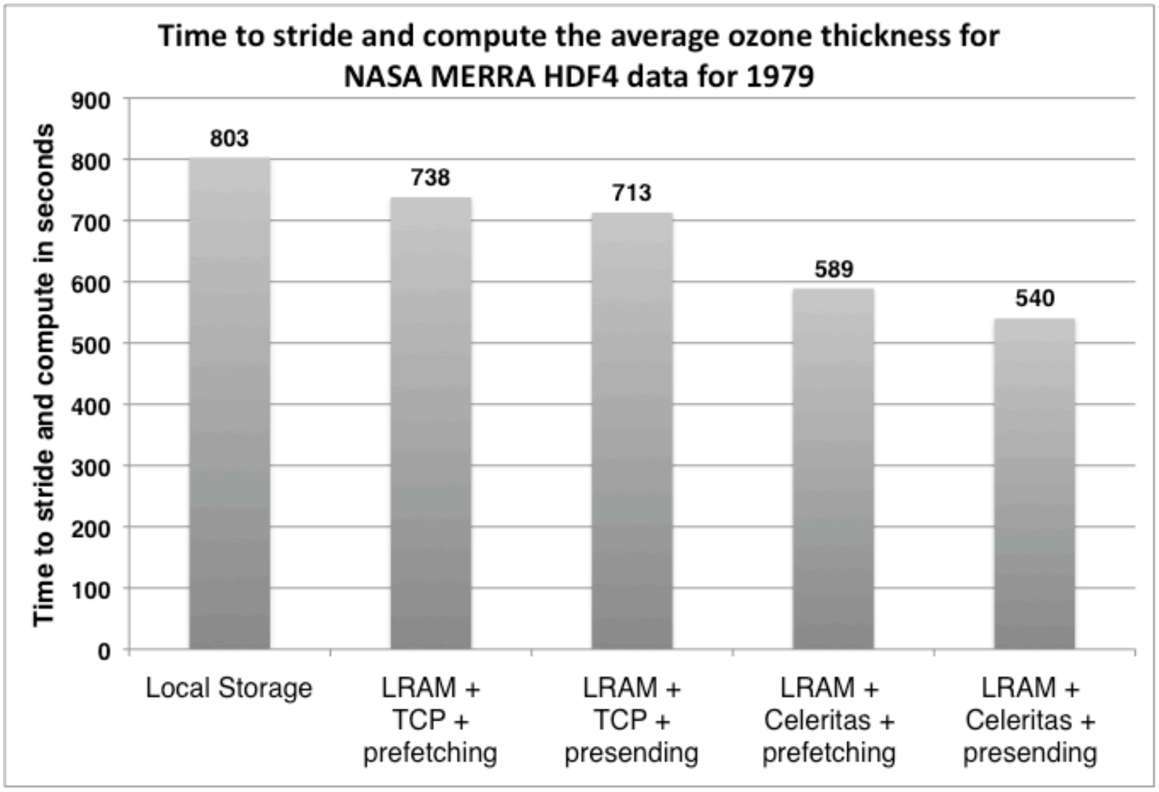

Figure 29: Ozone Thickness Computation using LambdaRAM over WAN

Striding and computing average ozone thickness for NASA MERRA data for 1979 using LambdaRAM over 10 Gbps Networks between Chicago and Goddard. Lower computation time indicates improved performance.

**Efficacy of adding data caching servers**

We evaluate the efficacy of adding server LambdaRAM nodes on the performance of striding and computing over wide-area networks. We replicated the 1TB MERRA data on a ultra-fast storage system at Starlight. The storage system consists of quad processor Intel 3.2Ghz Xeon, PCI-X based Neterion 10GE NIC, and, 24 SATA-1 disks configured in RAID5 using three hardware PCI-X RAID cards and XFS filesystem. This is connected using at 10Gbps to the nodes at NASA Goddard and EVL Chicago over the CAVEWave and Teraflow networks. The Server LambdaRAM was configured to encompass the node at EVL and at Starlight.

From the Figure 30, we see a two-fold improvement in performance using two servers in LambdaRAM. The addition of nodes helps distribute the load and data management on the Server side. With a single Server, the average bandwidth used was around 2.1Gbps. With 2 Server nodes, the average bandwidth utilized over the wide area network was approximately 3.5 Gbps. In the current experiments, the number of servers accessible over the wide-area was limited. It would be useful to evaluate the scaling as we increase the available bandwidth and the number of nodes.

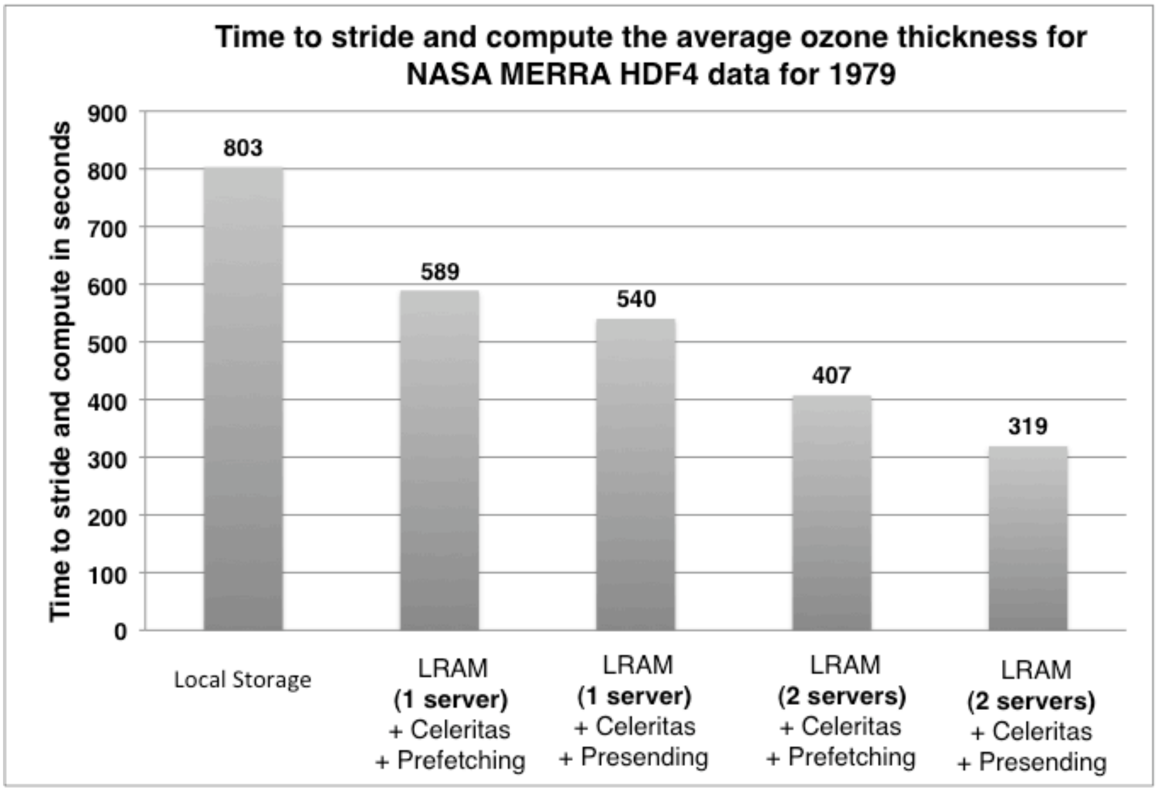

Figure 30: Efficacy of adding Data Servers in LambdaRAM

Effect of adding servers in LambdaRAM on striding and computing average ozone thickness for the entire world for 1979 over wide-area networks

## **Striding and computing the average surface temperature for MERRA data using LambdaRAM over wide-area networks**

Climate scientists routinely compute the average surface temperature. Accelerating this computation is critical for timely hurricane and tornado prediction. Surface temperature is a 3D dataset consisting of a time-series of floating point values for each earth grid point. Computing the average surface temperature involves multi-dimensional striding over a time-series of 2D datasets. The surface temperature computation application is written in C++. It runs on a single node and strides over the MERRA data to compute the average surface temperature. We compare the performance of striding and computing the surface temperature for the MERRA data for 1979 locally on the ultra-fast storage system with striding over the data using LambdaRAM over wide-area networks between EVL Chicago and NASA Goddard.

From Figure 31, we observe a 40% improvement in performance using LambdaRAM with Celeritas as the data transport protocol over the ultra-fast local storage system. On the Local storage, striding through multi-dimensional data involves accessing multiple noncontiguous regions on disk and incurs a lot of overhead and leads to performance degradation. Pre-sending yields a better performance than pre-fetching as the data striding access patterns, as it does not incur the request latency of prefetching. Additionally, Celeritas plays a key role in achieving high-performance over the Wide-area networks. Thus, we observe that it takes less time to remotely stride over multiple dimensions and compute the average surface temperature using LambdaRAM than computing it locally on the ultra-fast storage node. Celeritas and latency mitigation heuristics enable LambdaRAM to achieve high performance over wide-area networks.

We visualized the surface temperature computation at Goddard in real-time at Chicago. The visualization was designed using VTK and QT. The computed average surface temperature was streamed in real-time from NASA Goddard to Chicago over the 10Gbps network.

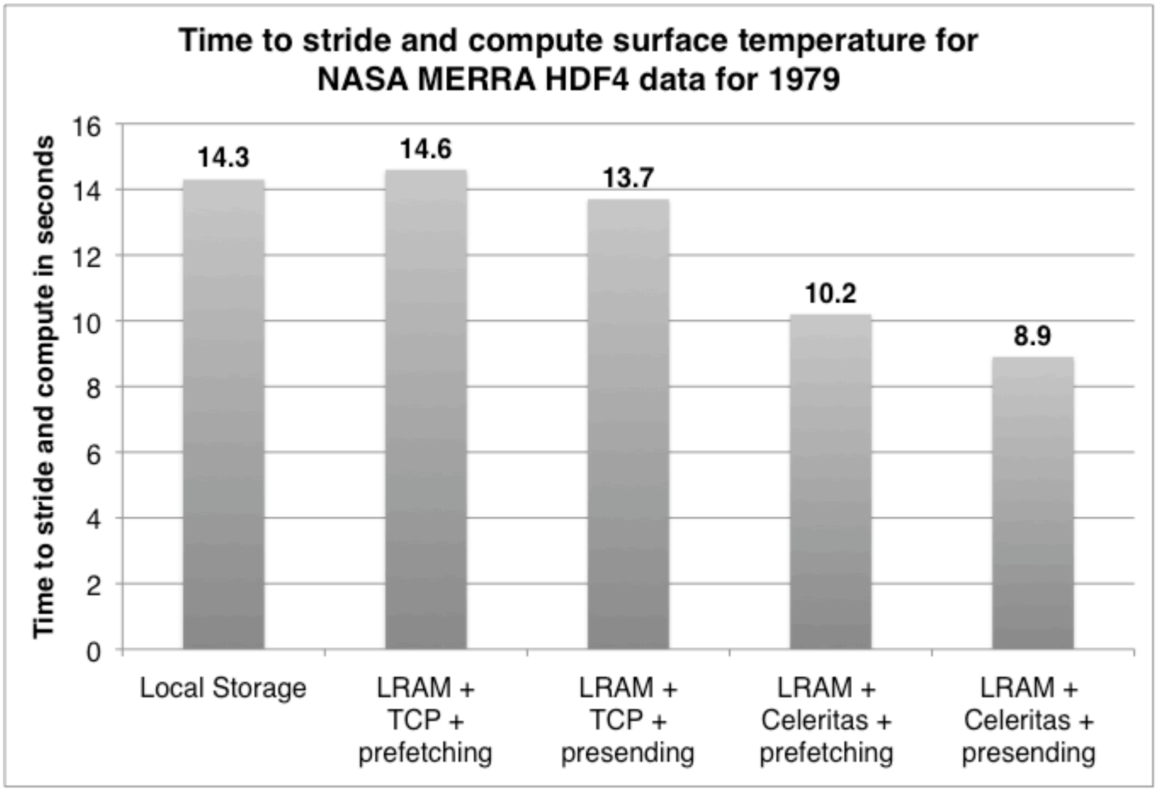

Figure 31: Computing Average Surface Temperature using LambdaRAM over WAN

Striding and computing the average surface temperature for the entire world for 1979. Lower the time to compute and stride, better the performance. The application running at NASA Goddard uses LambdaRAM to access data located in Chicago.

# 6. FORMAL SPECIFICATION AND VERIFICATION OF LAMBDARAM

Formal verification enables reliable deployment of LambdaRAM in safety-critical environments such as NASA's real-time climate analysis and forecasting applications. Additionally, this is important as we scale the protocol to Petascale systems wherein software testing is no longer sufficient to. We first present an abstraction of LambdaRAM in Section 6.1. We verify this abstraction for safety and progress properties using formal verification tools in Section 6.2. Formal verification helped identify a bug in the memory management heuristic of LambdaRAM.

## 6.1. The Abstraction Phase

In this section, we present our abstraction for the Read-Only consistency mode of LambdaRAM. As mentioned earlier, this mode is sufficient for most data-intensive HPC applications. To abstract LambdaRAM, we made the following assumptions:

1. HPC clusters are typically heterogeneous. Modeling heterogeneous cluster configuration drastically increases the number of parameters in the model. We have currently assumed homogeneous cluster configuration wherein all the servers and clients have the same
2. High-speed optical networks are point-to-point networks. Thus, they do not facilitate all-to-all communication needed for cluster-to-cluster communication. Aggregation technologies in Layer 2 (Ethernet grooming), Layer 3 (Routers), etc., are used to achieve

all-to-all communication. Aggregation technologies and the multiple networks paths between any source-destination pair in optical networks result in re-ordering of messages. We have currently not considered message re-orderings in our model.

3. Data-intensive applications are inherently parallel, where each data request is decomposed into requests of data blocks from several nodes. If a node request exceeds the node's memory in a LambdaRAM computation, the request is further decomposed into a sequence of data blocks, each fitting the node's memory. Here, we assume that requests do not exceed the maximum memory available on a single node and bypass the need to model a sequence of requests.
4. LambdaRAM can encompass the memory of multiple clusters interconnected by high-speed networks. We restrict our attention to a two cluster, client-cluster server-cluster configuration, which is one of the common configurations of LambdaRAM.

With these assumptions, we worked to formulate a higher-level abstraction of LambdaRAM.

### 6.1.1 Initial Phase

The Initial abstraction of an application running on two machines is as shown in Figure 32. The client cluster of LambdaRAM, on which the application typically executes, is composed of the following modules:

- **Data Access Module (DA):** responsible for satisfying an application's request for data blocks. DA first checks if the data block is locally cached, and sends a request to an appropriate client to fetch the block if is not cached.

- **Client Module (CLIENT):** satisfies the DA's request for uncached blocks from remote servers. It consists of a client connection to each server.
- **Garbage Collector (GC):** aids the memory management of the Local LambdaRAM by employing various heuristics including (e.g. LRU, and MRU).
- **Local LambdaRAM Cache (LRAM):** a shared data structure on each node, which is part of the global LambdaRAM Cache.

The server cluster of LambdaRAM is composed of the following modules:

Local LambdaRAM Cache (LRAM), Garbage Collector (GC) and the Server Module (SERVER). The LRAM and GC are similar to the client-cluster case, and, the SERVER is responsible for satisfying the data requests from the clients.

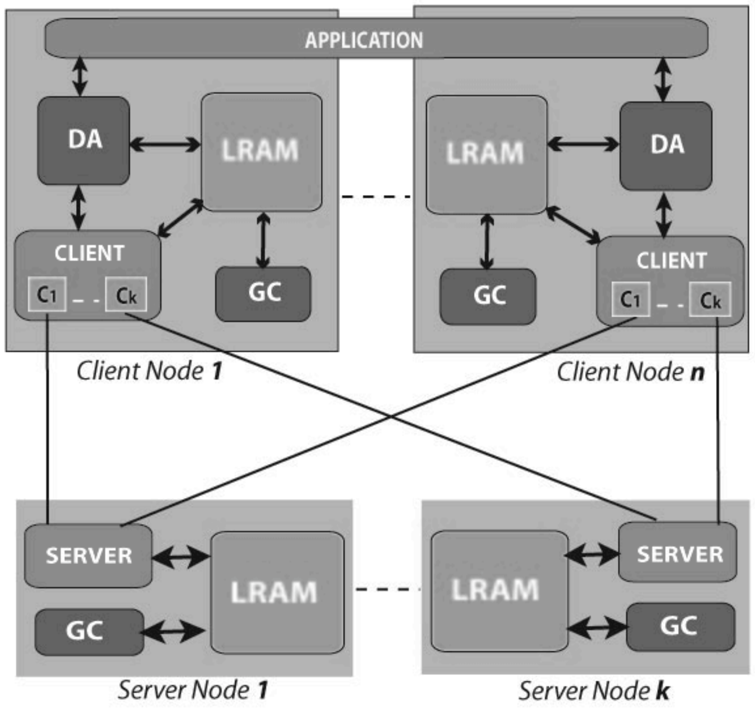

Figure 32: Initial Abstraction of LambdaRAM

### 6.1.2 Second Phase

We simplified the initial abstraction by assuming that the datasets fit into the combined memory of the LambdaRAM server nodes. This enabled us to eliminate the garbage collector on the server nodes and simplify the server nodes to a single server process servicing clients' requests. The garbage collector on the client's side could not be similarly abstracted since assuming the datasets fit into the memory at a client is not realistic. The resulting system is shown in Figure 33.

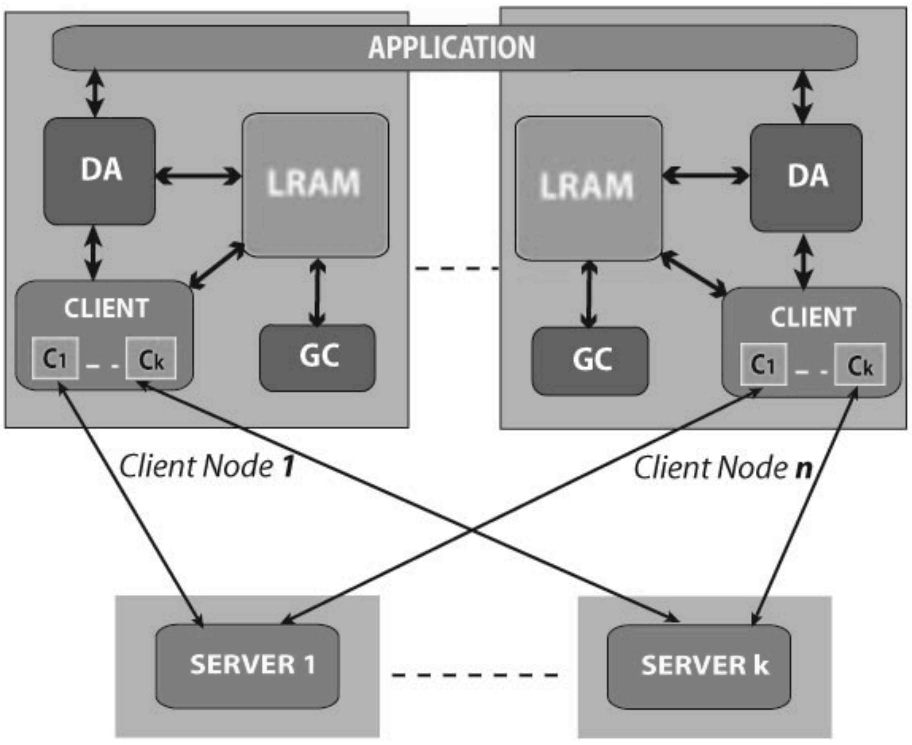

Figure 33: Abstraction with elimination of server-side memory management

### 6.1.3 Third Phase

LambdaRAM uses reliable data transport protocols and we assume reliable communication between clients and servers. We combine the client and server modules into a single client-server pair module, shown in Figure 34 as they exhibit a symmetric behavior for the Read-Only case.

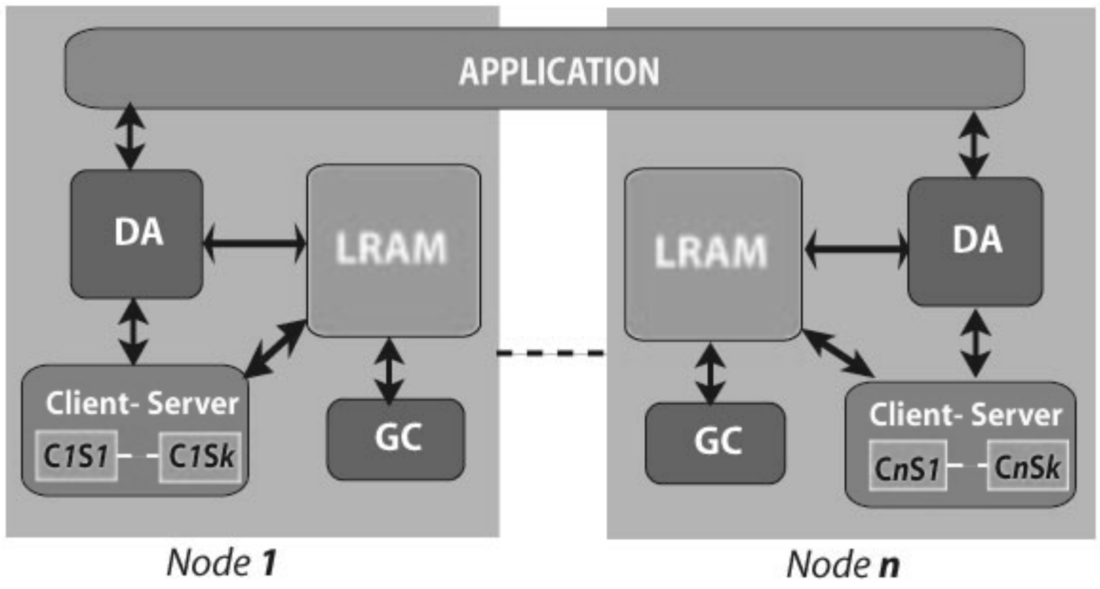

Figure 34: Client-Server Pair abstraction

### 6.1.4 Final Abstraction

Applications that use LambdaRAM are typically data-parallel applications and usually exhibit a symmetric behavior on each node. The simplified abstraction, taking advantage of this symmetric behavior, is as shown in Figure 35.

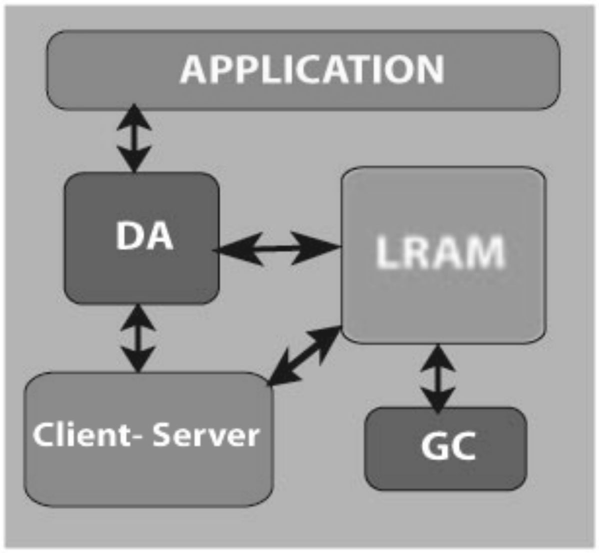

Figure 35: Abstracting incorporating the symmetric property of parallel systems

Figure 36 and Figure 37describe the message sequence charts of the main events in the system – Figure 36 describes the events from the time application requests data blocks until it receives them, and Figure 37 describes the concurrent (and independent) activity of the garbage collector.

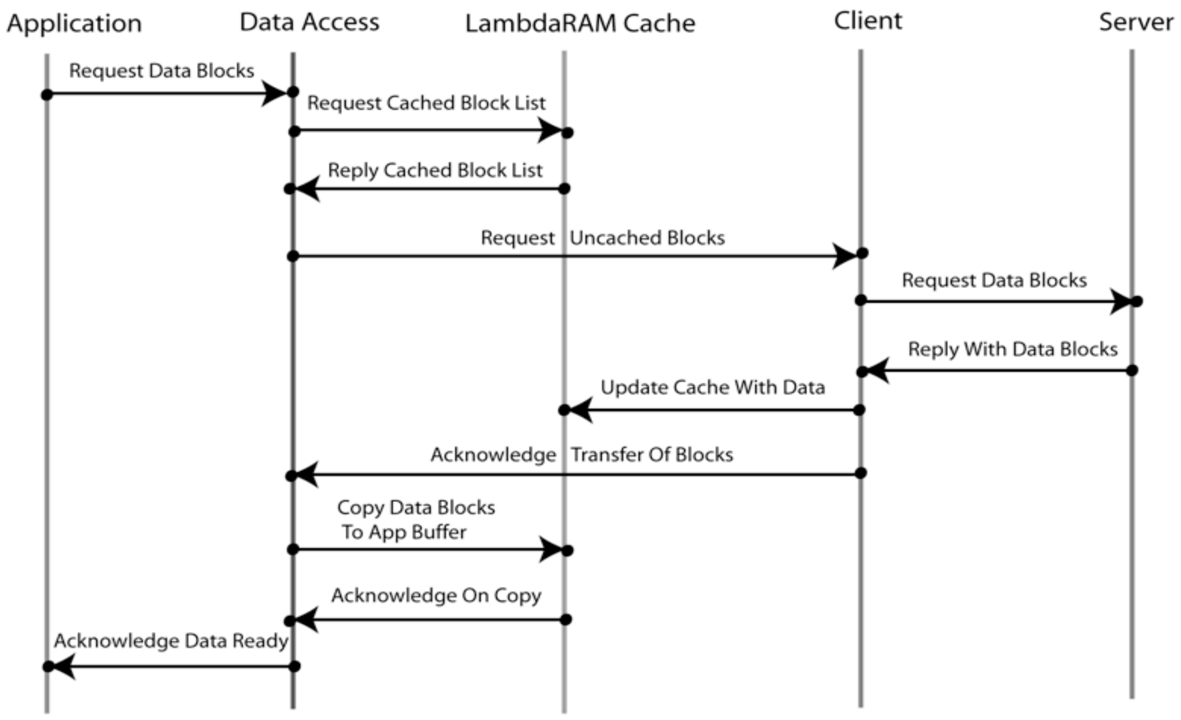

Figure 36: Message Sequence Chart for satisfying application requests

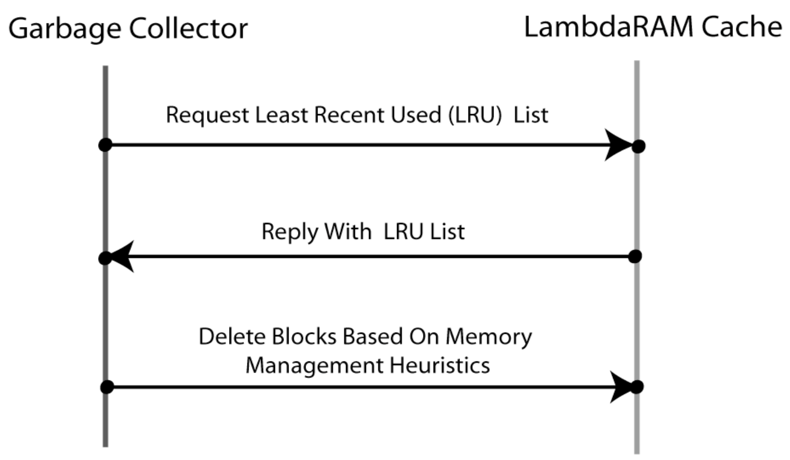
Garbage Collector
LambdaRAM Cache
Request Least Recent Used (LRU) List
Reply With LRU List
Delete Blocks Based On Memory Management Heuristics

Figure 37: Message Sequence Chart for garbage collection

## 6.2. Formal Techniques and Tools

We present a brief overview of the formal techniques and tools we used. There are two properties we were requested to verify: a safety property (of the type [] p where [] is the temporal operator ``always'' and *p* is a state assertion, i.e., an assertion whose truth depends only on the state it is interpreted on), and a *progress* property of the type $p \Rightarrow \langle\rangle q$, where $\langle\rangle$ is the temporal operator ``eventually'' and both *p* and *q* are state assertion. This property read as ``every *p*-state is eventually followed by a *q*-state.'' Since the system is *parameterized* (by, for example, the size of the memory, the bound on the size of cached memory, etc.), each assignment of values to the parameters defines an *instantiation* of the system. Verification of such a system implies verification of *every* instantiation, which, in general, is undecidable. There are, however, several techniques one can use that are sound, that is, if one succeeds verifying the system with these techniques, every instantiation of the system satisfies the properties.

### 6.2.1 Initial Steps

To make the verification task more manageable we made some simplifying assumptions. See section 6.3.1 for details. With the simplifications, we obtained a single parameter system, the parameter being the size of the memory. We coded the resulting system in SMV, which allows considering it as a *bounded just transition system* (BJTS) -- a transition system with justice (weak fairness) assumptions. Having the system expressed as a BJTS allows for analyzing it with several formal techniques as well as to apply some existing symbolic model checking tools on it.

### 6.2.2 Verification Techniques

To prove safety ( [] p) we employed the *Invisible Invariant* methodology [Pneuli01][Arons01][Zuck04] which allows for automatic verification such properties for a parameterized system.

To prove progress, we use a simplification of the method of [Fang06] : Suppose we want to show that a system that is composed of some parallel modules satisfies a progress property $p \Rightarrow [] \, q$ i.e., every $p$-state is eventually followed by a $q$-state.

Let $P_1, \ldots\ldots, P_n$ be a sequence of all the system's modules, $k > 0$ be a constant, and assume that $1 \le \ell \le n$. Let $r$ be the property: " Once $p$ becomes true, if $P_1$ takes $k$ steps, then ….., then $P_\ell$ takes $k$ steps, then a $q-$state must be reached in this duration". Then, obviously, $r$ implies the progress property $p \Rightarrow \langle\rangle \, q$.

More formally, let $M_0$ be a new module described below where *active* and *counter* are fresh variables, $active \in \{0,1..l\}$ and $counter : [1..l] \mapsto [0..k]$

```
If $(active = 0 \wedge p \wedge \neg q)$
then $active := 1;$ for all j $\in [1...\ell], counter[j] := 0$
$Elseif\,(q)$
then $active := 0;$
$Elseif\,((1 \le active \le \ell) \wedge (counter[active] = k))$
then $active := active + 1$
```

Similarly, for each $j = 1,....,\ell$, let *Mi* be the module described below.

If $(active\ = j \wedge p \wedge \ \neg q)$
then $counter\ [j]\ := counter\ [j] + 1$

We then have the following theorem, whose proof follows from a similar one in [Fang06]:

**Theorem 1**: *If system*

$$(M_0 \ ||| \ ||_{\mathrm{j=1}}^{\ell} (P_j \ ||| \ M_j)) \ || \ P_{\ell+1} \ || \ldots || \ P_n$$

*satisfies*

$$[]\ (active \ \rightarrow \bigvee_{j=1}^{\ell} counter[j] < k)$$

*then the system* $||_{i=1}^{n} P_i$ *satisfies* $p \Rightarrow \ <> q$

Note that we assumed that each module can always take an idle step. We, however, don't wish to count idle steps when non-idle ones are enabled (which will allow the counters to grow indefinitely, violating justice). In practical terms, we have a single ``Idle'' module that performs all the idle steps, and our processes, once scheduled, idle only if they have no other option. Note also that the method described applies to proofs where $l$ does not depend on the system parameter $N$. Hence, this is a simpler situation than the one described in [Fang06].

Theorem 1 demonstrates how progress properties can be transformed into safety properties. While $l$ is independent of $N$, the progress property may be parameterized, hence, we may need to verify the safety (implied by progress) of the new system using parameterized verification techniques.

### 6.2.3 Tools

We use Temporal Logic Verifier (TLV) [Pnueli96] for model checking. TLV is a Binary Decision Diagram (BDD) based model checker that uses SMV for its input language, and has interactive scripting capabilities that make it especially suitable for our purposes.

## 6.3. Verification Phase

LambdaRAM is implemented in C++; the code base is currently 30K lines, which renders it impossible to formally verify by existing automated tool. The abstraction described in Section 6.1 identified five modules. There were two properties that needed to be formally verified:

**Safety :** The number of non-empty cache blocks never exceeds the maximal memory that can be cached (which is given as a parameter)

**Liveness :** Every requested block is eventually is granted

These two properties seem like the typical toy properties given in basic formal verification texts, however, in the case of the LambdaRAM code, there are several factors that renders their verification considerably harder: The memory is multi-dimensional, a ``block'' consists of a list of hyper-boxes ``chunks'' of the memory, the number of applications, the shape of the memory, the maximal amount of memory that can be cached at a given time, the number of requests an application can issue, as well as numerous other parameters, can all vary, and formal verification should prove (1) and (2) regardless of the value of the parameters.

### 6.3.1 Simplifying Assumptions

We opted to make some simplifying assumptions in order to obtain an initial formal verification, and then to remove the assumptions. The assumption were chosen so as to be independent of one another with respect to verification of (1) and (2). The main assumptions are:

- **The memory is a linear array**

  While the complex structure of both memory and requests are an inherent part of the protocol, for proving (1) and (2) it suffices to assume that the requests can be translated to sequences of memory addresses, and that the latter can be represented as absolute addresses over *N*. At some later point, it may be necessary to verify this translation between the hyper-boxes into a sequence of addresses, but this is irrelevant to the properties we are aiming to verify.

- **Most parameters can be assumed to be small constants**

  The parameters that are relevant to proving (1) and (2) are the bound on the maximal number of memory cells that can be cached at a given time *(*MaxMemory*),* the number of application threads, the number of memory blocks *(*N*)*, and the bound on the size of requests. Obviously, MaxMemory should be larger than the maximal request size. However, as our abstraction of the memory implies, it suffices to assume that the request size is small. To simplify matters, we chose the request size to be 1. For sanity checks, we also verified the protocol with larger request sizes, and, as expected, obtained no new behaviors. Similarly, we chose MaxMemory to be some multiple of the request size. Again, we experimented with several values, and settled on 2 for the presentation here.

As we note in the future work section, we are currently working on obtaining the automatic verification with general parameters, or on formally proving that small values we chose indeed suffice.

- **Module Abstraction**

  We chose to (manually) abstract some modules, to verify the system with the abstracted modules, and to separately verify that the abstraction is correct. The latter was accomplished by methods similar to [Abadi91]. Since we are using a model checker, we could not prove the abstraction for arbitrary instantiations of parameters, however, we did obtain successful model checking runs with non-trivial instantiations, and a deductive proof that we are now ``guiding'' the tool TLPVS to generate.

- **Atomicity Assumptions**

  As is common in this type of parameterized verification, we assumed that some tests are performed atomically while in any reasonable implementation this is not a realistic assumption. We are currently working on applying some of the new methodologies (e.g., [Abdulla08]) to remove such atomicity assumptions.

## 6.3.2 Proving Safety

The safety property we wish to prove is that the number of cache blocks that are cached or are in transit never exceeds the maximal memory that can be cached. For each memory block \$i\$, the variable CacheBlock_State[i] denotes the state of the $i^{\text{th}}$ memory block, and it is neither cached nor in transit when it equals EMPTY. Hence, the safety property we want to verify is that for every instantiation *N*,

$$[](\sum_{i=1}^{N}(\text{CacheBlock\_State}[i] \neq \text{EMPTY}) \leq \text{MaxMemory})$$

To prove the property, we employed the method of Invisible Invariants using total number of blocks as the single parameter. We chose RequestSize to be 1 and MaxMemory to be 2. The transition relation is of the form $\exists i.\forall j.\rho(i,j)$ where i and j range over 1..TotBlocks and $\rho(i,j)$ refers to two free index variables. Suppose we are seeking an invariant of the form $\forall i,j.\phi(i,j)$. Using invisible invariants, we can use instantiation of size $N_0 = 4$. In fact, we chose a larger $N_0$ and succeeded in generating inductive invariants for shapes that have a 2- and a 3- universally quantified. We approached the problem in two directions -- in one, we went the usual invisible invariant way, starting with the set of reachable states, projecting on two (or three) processes, and generalizing onto the others. We also attempted to produce the invariant by starting with $\Theta$ the initial assertion, and iteratively projecting and generalizing it, until a fix point is reached. Surprisingly, both methods produced the same inductive invariants, only the latter (starting with $\Theta$ and reaching a fix point) took considerably more time. This is contrary to prior simpler experiences where both methods produced the same invariants and the latter method converged much faster. It's hard to draw conclusion from this, and as much as we can, we'll continue to use both methods simultaneously (if for nothing else, it proved to be a very efficient debugging tool) and attempt to gauge their relative merits. The results are shown in Table 5.

We ran the experiments on 2.2GHz Intel Core 2 Duo MacBook Pro with 2GB of 667Mhz DDR2 SDRAM. The Darwin Kernel Version running on the MacBook Pro was 8.10.1. TLV 4.18.4 was used for model checking.

Table 5: Model Checking Run-Time Results

| $N_0$ | invariant shape | from reachable | from $\Theta$ |
|---|---|---|---|
| 4 | $\forall i, j.\phi(i, j)$ | 0.45 sec | 2.9 sec |
| 5 | $\forall i, j.\phi(i, j)$ | 1.12 sec | 7.08 sec |
| 6 | $\forall i, j.\phi(i, j)$ | 2.06 sec | 14.66 sec |

### 6.3.3 Proving Liveness

The liveness property we wish to establish is that every requested block is eventually granted. A block $i$ is requested when RequestBlockList[i] is set, and is granted when RequestBlockList[i] is reset. Hence, the liveness property is that for every $i \in [1..\text{TotBlocks}]$,

$$\text{RequestBlockList[i]} \Rightarrow \langle\rangle\neg\text{RequestBlockList[i]}$$

Since all the blocks are treated symmetrically, it suffices to establish the property for a *representative* block, say $i = 2$. Hence, we focus on verifying:

$$\text{RequestBlockList[2]} \Rightarrow \langle\rangle\neg\text{RequestBlockList[2]}$$

As described earlier, we arranged the modules where $P_1$ is Application (App), $P_2$ is DataAccess (DA), $P_3$ is ClientServer (CS), and $P_4$ is GarbageCollector (GC). Thus $n = 4$. We also choose $\ell$ to be 3 and $k$ (the counter bound) to be 3.

With $\text{p} = \text{RequestBlockList[2]}$ and $\text{q} = \neg\text{RequestBlockList[2]}$, we verified the system

$$(M_0 \;|||\; ||_{j=1}^{\ell} (P_j \;|||\; M_j)) \,||\, P_{\ell+1} \,||\, \ldots \,||\, P_n$$

against the safety property

$$\forall N.\ [\,]\ (\mathit{active} \rightarrow \bigvee_{j=1}^{\ell} \mathit{counter}[j] < k)$$

using the method of invisible invariants (taking the same $N_0$ as before). From Theorem 1, it now follows that the original system satisfies

$$\text{RequestBlockList}[2] \Rightarrow \langle\rangle\neg\text{RequestBlockList}[2]$$

In fact, before the successful verification, we obtained error traces, which allowed each module to take infinitely many idle steps. **A more careful inspection revealed a bug** -- the garbage collector (GC) was always allowed to change the observable behavior of the system even when there were no changes in the memory since its last pass. This enabled a scheduler that scheduled other modules only when the garbage collector prevented them from taking a productive step. This was fixed to make sure the garbage collector doesn't perform unnecessary work, and we could prove the liveness property.

# 7. RAILS TOOLKIT (RTK) - ENABLING TOPOLOGY-AWARE HIGH-END COMPUTING

Data-intensive middleware have demonstrated scalable performance using today's cyberinfrastructure architectures. However, future architectures, as listed in Table 6, will require today's applications and middleware to scale their performance in ways previously unexplored. Future cyberinfrastructure will be characterized by deep and complex memory, processor and interconnect hierarchies with inherent parallelism in the various subsystems, and the large bandwidth available to remote memory. Thus, a critical component for scalable performance for middleware, including LambdaRAM, is novel techniques for efficient utilization of end-system architectures and resources.

Typically, e-Science applications and middleware scale their performance to end-systems by optimizing their implementations for the end-system architecture. However, as end-system architectures evolve and become more complex, solutions that aid in the design of evolvable software are of paramount importance. One way to achieve this would be to develop abstractions of the various subsystems. These abstractions can help e-Science programmers design efficient and deployable middleware and applications. We present the Rails Toolkit (RTK), an approach towards enabling e-Science applications and middleware to effectively exploit the potential of these architectural trends. RTK abstracts end-system topology for applications and middleware, and enables co-scheduling of CPU cores, GPUs, memory and network resources within multi- and many-core computer systems. We define a "rail" as the co-scheduling of two or more of these resources. Using RTK, application

Table 6: Cyberinfrastructure Architectural Trends

| **Subsystem** | **Currently Deployed Architecture** | **Future Architectural Trends** |
|---|---|---|
| Processor | Dual and Quad core | Multi- and Many-cores with a Multi-dimensional topology |
| Memory | SMP, NUMA (typically 2 memory banks) | SMP, NUMA, Hybrid combination of SMP and NUMA, Multi-dimensional (2D and 3D) memory topology |
| Graphical Processing Unit (GPU) | PCIe based GPU (typically with 128 processors) | Multiple GPUs with 256 to 800 processors per GPU (potentially on-core GPU design) |
| System Interconnects | Shared Bus, PCIe Gen 1 (2.5 Gbps) | Multi-lane PCIe Gen 2 and 3, Quick processor interconnect (QPI), HyperTransport, (HT) DWDM-based optical interconnects |
| Network Interconnects | 10 GE Ethernet, Infiniband, Myrinet, etc. | 40 Gbps – 100 Gbps Multi-lane Ethernet, Infiniband interconnects, Multi-lane DWDM based interconnects |
| Wide-Area Network | 1-10 Gbps networks | DWDM-based Multi-10 Gbps optical networks |

developers can create one or more rails over which their data-intensive computations and data retrievals can be accelerated with minimal interference from other rails or applications, and thus dramatically improve program performance. RTK is an open source toolkit and presents an intuitive API for applications and middleware to efficiently utilize end-system architectures. RTK can be used to improve the performance of high-performance computing applications, high-speed data delivery applications, and high-resolution graphics and video streaming. Figure 38 depicts a network rail which is a software abstraction of a processor core connected to a lane on a network interface card (NIC) via a dedicated interconnect. A network rail helps in improving the achievable throughput and reducing the message latency by reducing cache pollution and lowering memory access latency. The rails approach enables pipelining of multiple subsystems to compose hybrid rails. The RTK API can be used to pipeline GPU rails and network rails, and thus improve the performance of graphics streaming for remote visualization by reducing cache pollution, exploiting memory locality to reduce latency and reducing system bus contention. This is critical for future cyberinfrastructures where GPUs are an integral component. RTK enables allocation of parallel rails, which, facilitates exploitation of system topology and the parallelism inherent in current (and future) system architectures. A parallel four-rail network rail system is depicted in Figure 38, each rail consists of a processor core with dedicated memory connected to a lane on a NIC via a dedicated interconnect. The parallel rails approach can be expanded to exploit parallelism in other sub-systems. RTK is implemented in C++ and is distributed under GNU Public License (GPL) version 2.1. It works under Linux and has been tested on SMP-based Intel architectures, NUMA-based AMD Opterons and IBM Cell architectures. We describe the Rails toolkit architecture in Section 7.1, and evaluate the performance of RTK on micro-benchmarks and application-level benchmarks in Section 7.2.

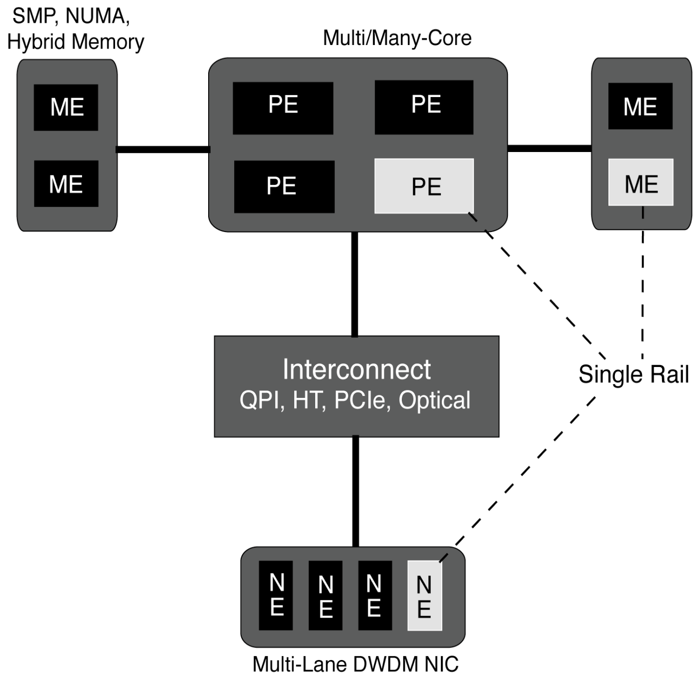

Figure 38: The Rails Approach

This figure depicts a network rail wherein co-allocation of memory elements (ME), processor elements (PE) and networks resources (NE) help achieve improved performance

## 7.1. Rails Toolkit Architecture

Figure 39 depicts the Rails Toolkit Architecture, which consists of the Resource Abstraction Layer, Resource Allocation Layer and the Rail Allocation Layer. The **Resource Abstraction Layer** abstracts the end-system topology and deals with the low-level resource bindings. The **Resource Topology Database** maintains relevant information including the topological configuration of the available processors, cores, memory nodes and IO devices. This database is populated during initialization by probing the system resources and using input configuration files. The **Resource Binding Layer** is responsible for binding interrupts to processor(s), threads to processor(s), the memory policy and allocation over node(s). This layer is designed using a wrapper around Linux system calls and enables co-allocation of the resources. The **Resource monitor** is a lightweight daemon that periodically checks the online status of the processors and memory nodes.

The **Resource Allocation Layer** allocates threads, sockets and memory using the underlying resource abstraction layer. The **Thread Library** is a C++ wrapper around the pthread library and enables manipulation of the processor-thread binding and memory policies of a thread using the resource abstraction layer. Additionally, it provides in-depth performance statistics, including context-switches and priorities, on a per-thread basis. The **Memory Allocation Library** enables topology aware memory allocation. It supports the NUMA memory policies available in Linux including interleaving, local allocation and strict allocation. The **socket library** currently supports TCP, UDP and Parallel TCP. The library provides in-depth performance information of the network streams. The library is extensible and is useful in the design of composable protocols such as Reliable Blast UDP [He02] and LambdaStream [Xiong05] [Vishwanath06]. The Rail Allocation Layer synergistically co-allocates resources for improved performance. This layer can aid in isolating resources and reducing contention. The layer also enables pipelining of rails. In graphics streaming applications, pipelining GPU and network rails is important for reducing resource contention, including the contention in IO bus due to the GPU and network subsystem competing for it.

We have exposed the capabilities at various layers as a lightweight API so that researchers interested in applying this approach have multiple levels to integrate their applications with RTK. Details of the API are available at the RTK website. We envision middleware and applications using RTK to fully exploit the topologies of end-systems. RTK could be used in the design of adaptive run-time systems to optimize resource allocation. One such example is the **MultiRail Socket Library**, which enables seamless use of multiple network rails for network intensive applications. It is implemented in C++ and derived from the rail socket library. It exploits:

- Data Parallelism by striping the data onto multiple data streams. The current implementation uses static data striping and can be augmented to use adaptive striping policies .
- Task Parallelism by employing worker threads to stream each of the data streams. This makes efficient use of system resources in a multi-core, many-core environments.
- Network Parallelism by streaming the data streams over the multiple network paths available between a source and destination pair.

Additionally, the library employs efficient memory interleaving heuristics to improve memory bandwidth.

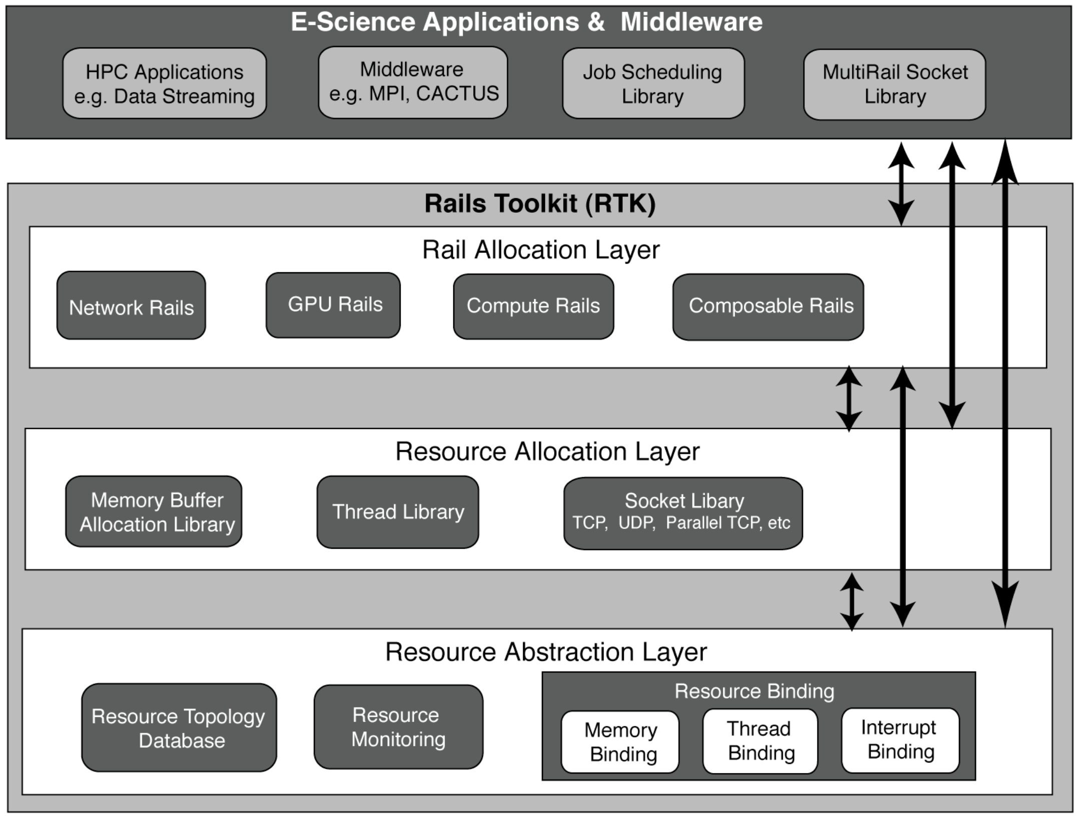
E-Science Applications & Middleware
HPC Applications e.g. Data Streaming
Middleware e.g. MPI, CACTUS
Job Scheduling Library
MultiRail Socket Library
Rails Toolkit (RTK)
Rail Allocation Layer
Network Rails
GPU Rails
Compute Rails
Composable Rails
Resource Allocation Layer
Memory Buffer Allocation Library
Thread Library
Socket Libary TCP, UDP, Parallel TCP, etc
Resource Abstraction Layer
Resource Topology Database
Resource Monitoring
Resource Binding
Memory Binding
Thread Binding
Interrupt Binding

Figure 39: Rails Toolkit Architecture
This figure depicts the various layer in the design of the Rail Toolkit (RTK) architecture. RTK consists of Resource Abstraction Layer, Resource Allocation Layer and Rail Allocation Layer,. E-Science applications and middleware can use any of the 3 layers for optimizing their performance to an end-systems topology.

## System properties critical for topology-aware resource allocation

We discuss properties that help improve an application's performance by enabling efficient topology-aware resource allocation. In this paper, we restrict our focus towards properties critical for network-intensive workloads. However, we would like to note that these properties are also necessary for other e-Science workloads, including compute-intensive workloads.

We define the ***Interrupt Affine* property** as one wherein the interrupt processing is performed on the processor to which the IO device is physically bound. Interrupt affinity reduces the message latency by servicing the interrupts on the nearest processor. As seen in Figure 40, NIC 0 is physically attached to the PCIe bridge physically connected to processor 0. If the interrupt processing of NIC 0 occurs on any core of processor 0, we consider this to be interrupt affine. If the interrupt processing of NIC 0 occurs on processor 1, the interrupt affinity is not set. Thus, we define interrupt affinity relative to the physical topology of the IO device.

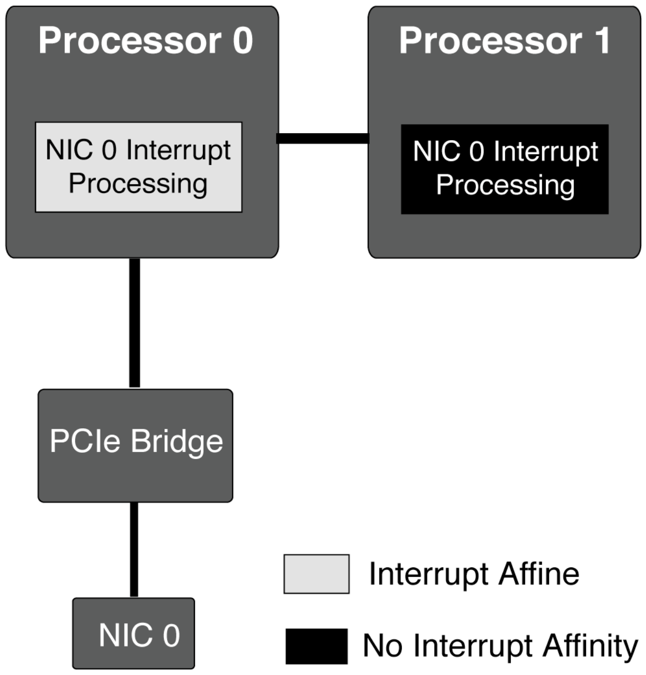

Figure 40: Interrupt Affinity
This figure depicts the Interrupt Affinity (IA) property. IA is set if the interrupt processing occurs on a processor where the device is physically connected

We define the ***Thread Affine property*** as one wherein the network application thread is scheduled on the processor in charge of the interrupt processing. Thread affinity reduces cache pollution and improves latency as the interrupt processing and the application thread are scheduled on the same processor. As seen in Figure 41, if the network application thread is scheduled on processor 0 and the interrupt processing of NIC 0 occurs on processor 0, we consider this to be thread affine. In Figure 41, we have both interrupt affine and thread affine properties. Thus, for network intensive workloads, we define thread affinity relative to the corresponding interrupt processing.

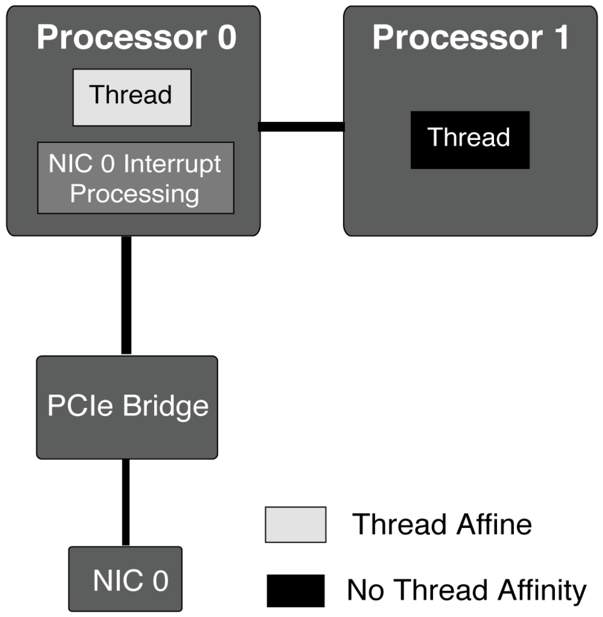

Figure 41: Thread Affinity
This figure depicts the Thread Affinity (TA) property. TA is set if the network application thread is bound to the processor where the interrupt processing occurs

We define the ***Memory Affine Property*** as one wherein the memory buffer used by the network application thread is allocated on the memory bank with the lowest access latency with respect to the application thread. In case of NUMA-based systems, memory allocation on the local memory bank is considered to be memory affine. In Figure 42, the network application thread is scheduled on processor 0. If the memory is allocated on node 0, we consider this to be memory affine. Memory affinity helps in reducing the data access latency. In case of system architectures with deep and multi-level memory hierarchies, memory affinity refers to allocation of memory on memory nodes with the least access latency. Lower memory access latency is critical for data-intensive e-Science.

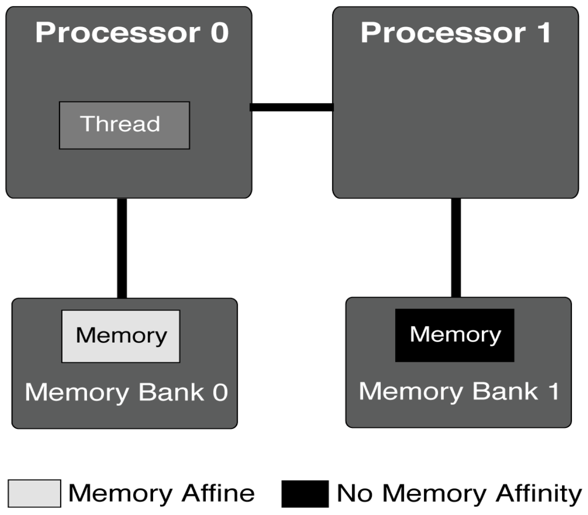

Figure 42: Memory Affinity

This figure depicts the Memory Affinity (MA) property. MA is set if the application buffer is allocated on the memory bank where the network application thread is bound.

Additionally, using RTK, one can enable multiple properties simultaneously for improved performance. Enabling thread and interrupt affinity together would help in an improved performance over the individual affinities due to lower cache pollution among others. As mentioned earlier, RTK can be used to form parallel rails to exploit the inherent parallelism in end-systems.

If **T** is the achievable performance of a single rail, In an **N**-rail system, the expected performance would be: **N x T x** $\partial$, where $\partial$ is the parallel efficiency. In an ideal parallel system, the parallel efficiency is approximately unity ($\partial \rightarrow 1$)**,** and this system exhibits additive performance. The goal would be to identify the affinity combinations that would help parallel efficiency. This could be used towards the design of efficient run-time systems.

## 7.2. <u>Experimental Analysis</u>

In this section, we study the efficacy of the RTK toolkit on a set of micro-benchmarks and application-level benchmarks. We focus our attention on network-intensive benchmarks. The experimental testbed consisted of Two dual-core, dual-processor AMD 2.6 GHz Opteron TYAN 2895 systems with 4GB RAM and two PCIe 16X slots. The two machines were connected back-to-back with two 10 GE Myrinet NIC each. The Linux kernel version used was 2.6.18 with MSI enabled. The MTU used for the experiments was 9000 bytes. The 1.4.1 Myrinet driver was used in the experiments

### 7.2.1 Micro-benchmarks

We evaluate the performance of rails and the RTK toolkit on network intensive workloads. In e-Science cluster-based applications, a node routinely needs to send and receive data from multiple nodes. We designed a simple TCP micro-benchmark program to measure the efficacy of the rails approach on the achievable throughput, CPU usage and message latency. The benchmark program is written in C and creates four network streams between the two test nodes. Two network streams are bound to each of the 10G NIC. The client and server programs use the RTK Rail Abstraction Layer API. We compare the RTK version with a socket-based program. The socket program does not use the RTK API and relies on the default system affinity and scheduling heuristics.

#### 7.2.1.1 Effects of Various Rail Configurations on Message Latency

Figure 43 compares the performance of setting the rails affinities on the message transfer latency for various payloads using four concurrent TCP (two per NIC). We compare this performance with the default case in Linux. As seen from the graph, as the payload size increase, thread and memory affinities help in reducing the transfer latency. For a payload of 8MB, thread and memory affinity together yield reduce the message latency by 33% over the default Linux case. This is mainly due to the improved cache locality and lower data access latency due to memory affinity. This is reduction in message latency is critical for applications using MPI and network-intensive applications.

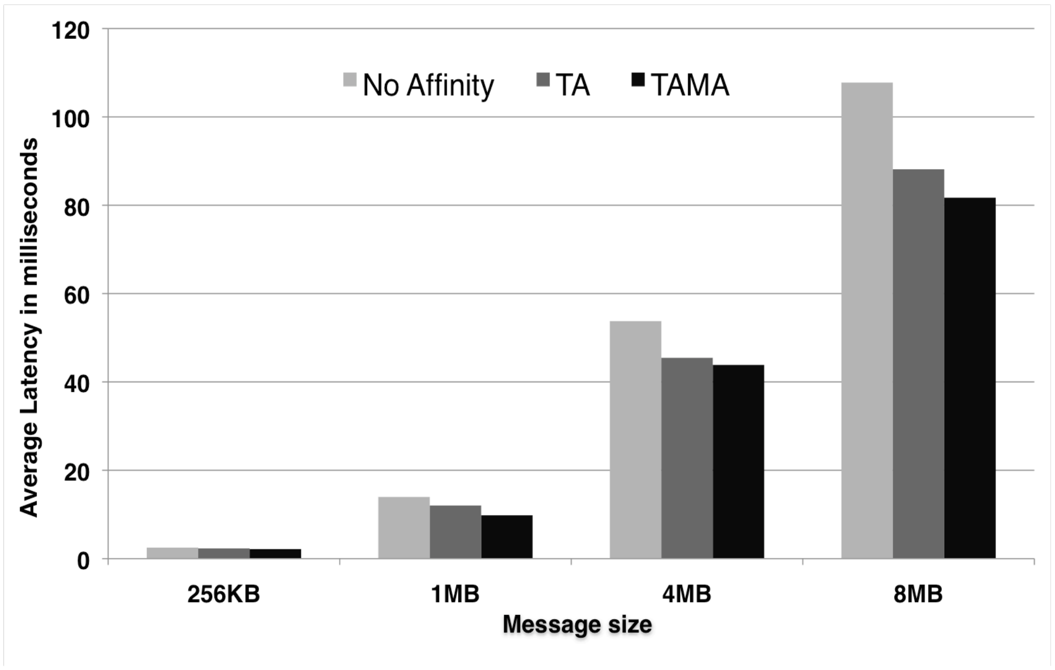

Figure 43: Effect of RTK on message latency of multiple TCP streams
From the figure, we see that as the message size increases, memory affinity is key for improved throughput

A similar trend is seen in the case of UDP in Figure 44 where thread and memory affinity together help reduce the transfer latency. The effect of thread affinity on latency is clearly visible as the payload size exceeds 4K (page size). The effects of memory affinity are not very pronounced in comparison to thread affinity due to the fact that the payload fits into the processor cache.

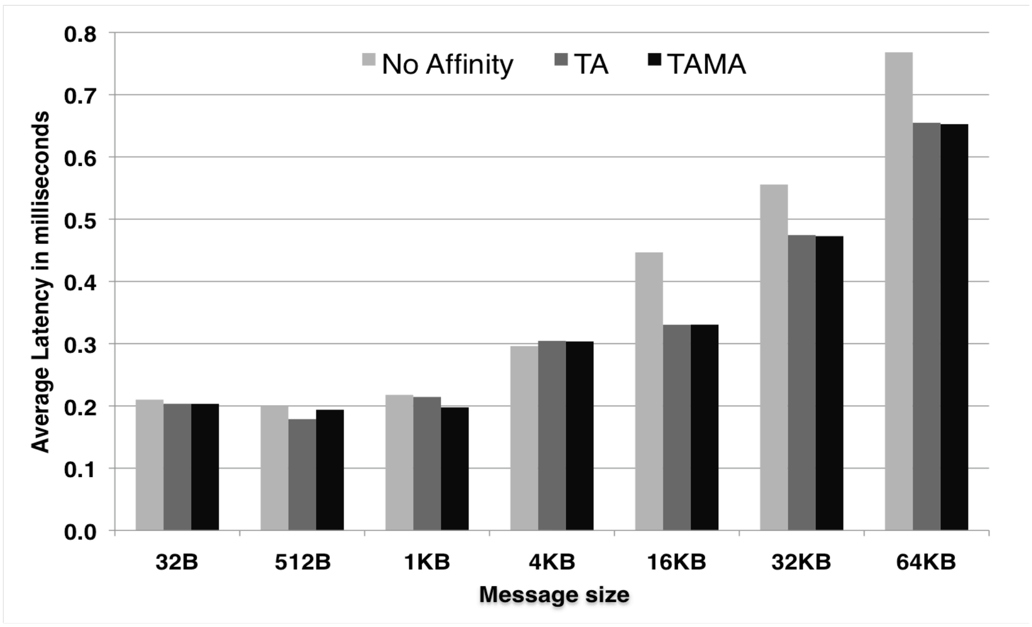

Figure 44: Effect of RTK on the Latency of UDP Streams
From the figure we see that Thread affinity plays a key role in improved throughput

#### 7.2.1.2 Effects of Various Rail Configurations on Throughput and CPU Utilization

The effect of affinities of the throughput of network-intensive TCP and UDP workloads is shown in Figure 45 and Figure 46. The workload consists of four concurrent streams (two per NIC). We compare the achievable throughput for the stream using the RTK Abstraction Layer API to enable affinities (at both the sender and the receiver) with the achievable throughput on the system relying on default system settings. We notice that enabling affinities improves the achievable throughput. This improvement is significant for higher payloads. In case of TCP, for a payload of 256 KB, affinities result in an improvement of 2 Gbps over the default settings. This is primarily due to factors, including lower cache pollution and lower memory access latency. Thus, allocating resources taking the topology into account is critical for network intensive e-Science and throughput intensive applications including wide-area data transfer.

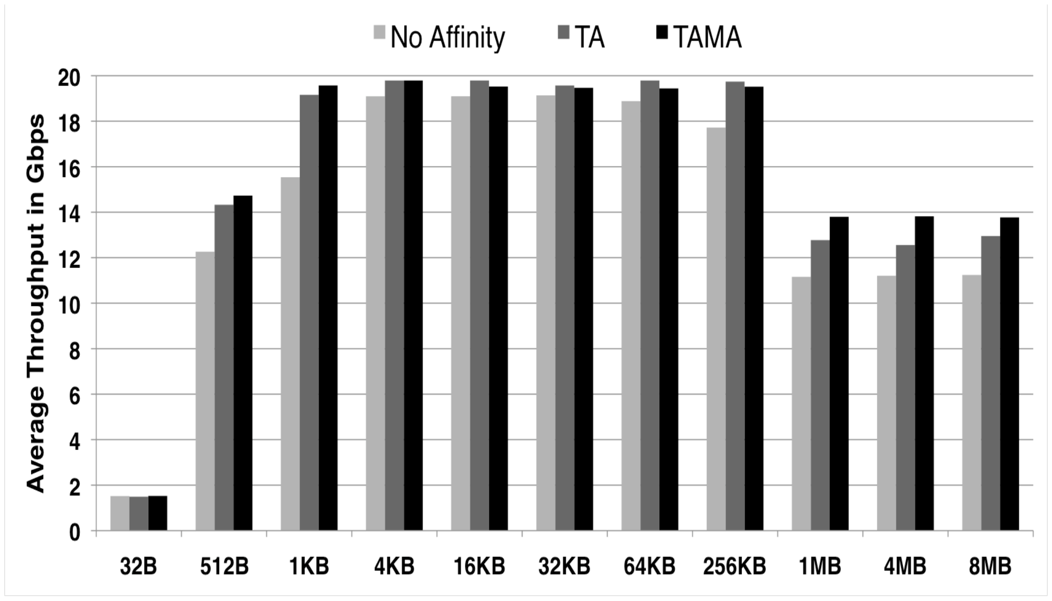

Figure 45: Effect of RTK on the Aggregate Throughput of TCP Streams
The Figure depicts the effect of affinities on the aggregate throughput of four TCP streams. As the payload size increases, enabling affinities leads to higher throughput in comparison to a default Linux system.

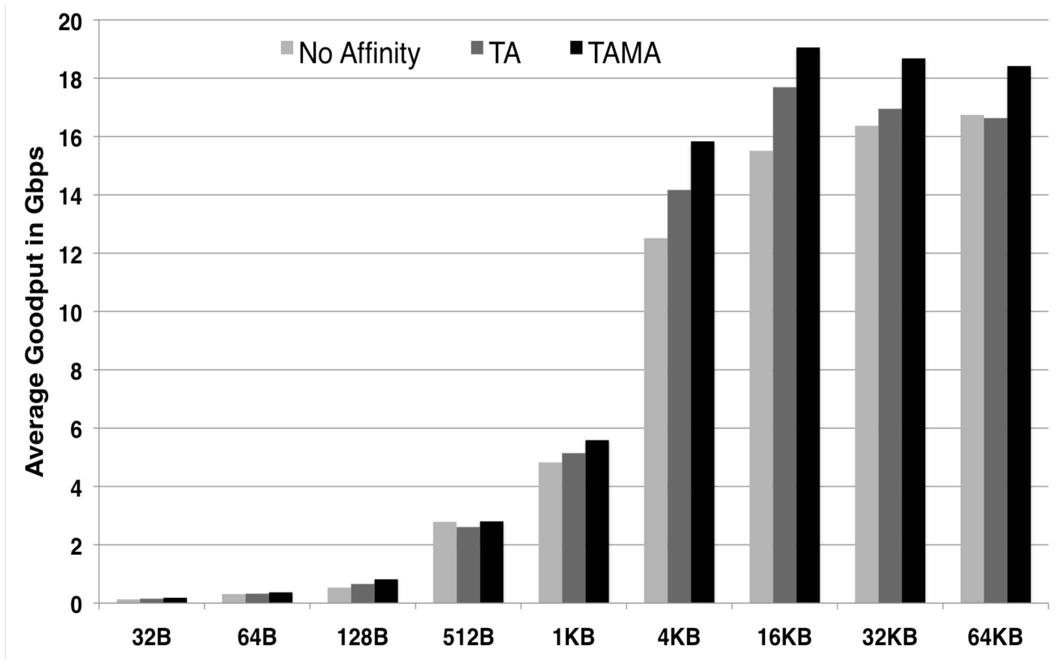

Figure 46: Effect of RTK on the Aggregate Goodput of UDP Streams
The Figure depicts the Effect of affinities on the aggregate goodput (thoughput with 0% packet loss) of four concurrent UDP streams. As payload increases, enabling affinities leads to a higher performance in comparison to a default Linux system.

**Effect of Rails on CPU utilization**

Figure 47 depicts the effect of affinities on the average CPU utilization of the network streaming at the receiver to process 1 Gbps. In this experiment, we have 4 concurrent streams competing to process the network streams. This is very common in cluster-based applications. Lower CPU usage by the network application would yield precious CPU cycles for the compute-intensive components. Enabling affinities results in a reduced CPU usage, which is primarily due to reduced contention of resources and lower cache pollution. In case of 8MB payload, a fully affine system leads to the 50% reduced CPU usage. This is mainly due to the fact that memory affinity leads to low-latency data access. This is critical for cluster-based applications wherein precious additional CPU cycles (leveraged from an affine network component) can be dedicated to compute-intensive components.

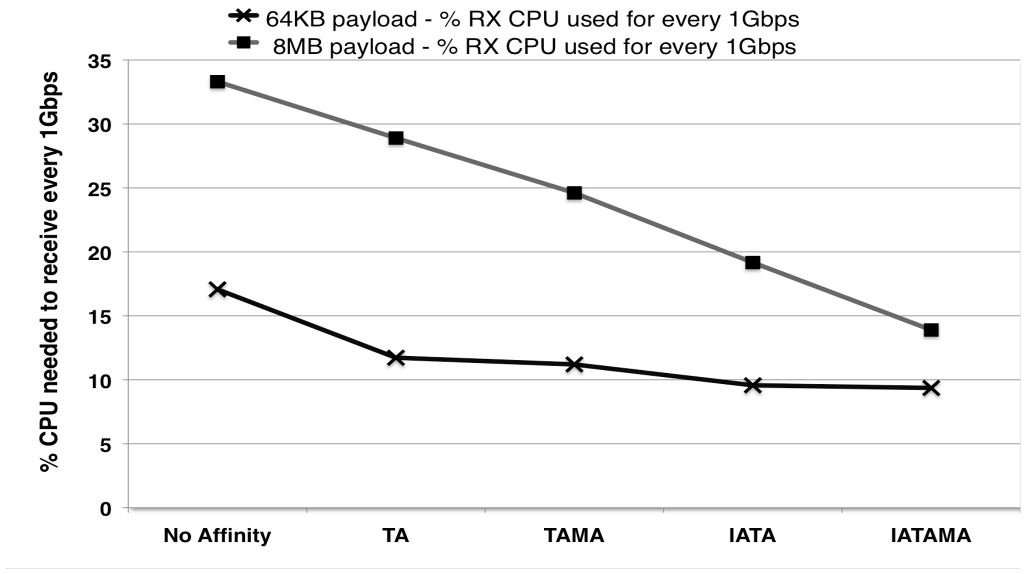

Figure 47: Effect of RTK on CPU Utilization
The Figure depicts the effect of affinities on the average CPU Utilization of four concurrent TCP Streams to process 1Gbps. Using RTK, the application needs less CPU to process 1Gbps TCP traffic.

#### 7.2.1.3 MultiRail TCP benchmarks

With the advent of Multi-lane NICs, efficient methods to exploit parallelism throughout the system and networks are necessary. This is critical for future LambdaGrids based applications using IP over Ethernet over DWDM. The current options include, Ethernet channel bonding and designing a multi-threaded protocol. MultiRail-TCP is a simple socket-like API that allows applications to leverage the performance benefits of rails without having to significantly modify an application's source code. Internally, it takes advantage of data parallelism by splitting the data onto multiple streams; task parallelism by creating worker threads to stream the data; and network parallelism by using the multiple NICs available on the system. Similar to the micro-benchmarks, our MultiRail-TCP experiment created network rails with thread and memory affinity. The memory was interleaved between the two memory banks. As seen in Figure 48, MultiRail-TCP achieves a throughput 2 Gbps higher than a Multi-Threaded TCP over 2 NICs. This is due to the fact that the network rails reduce resource contention that is present in Multi-Threaded TCP wherein the threads are scheduled based on the Linux scheduler's heuristics. MultiRail-TCP achieves 10 Gbps higher throughput than Linux Ethernet channel bonding. This is primarily due to the locking overheads in the Linux channel-bonding driver. Linux channel bonding works at gigabit rates but fails to scale to 10Gbps rates. Thus, one can achieve high performance at a user-space by exploiting parallelism throughout the system.

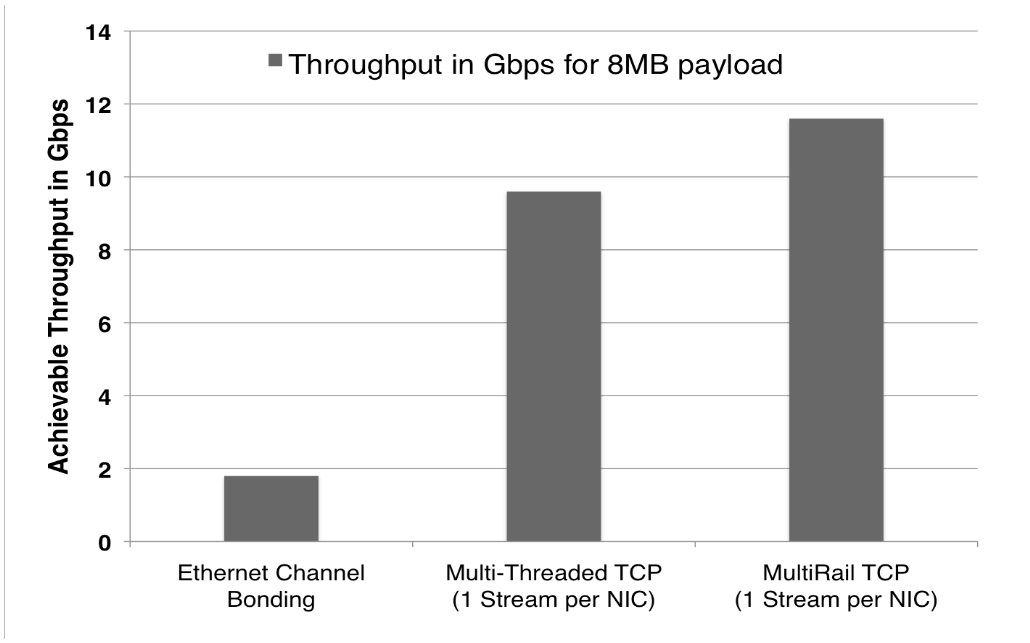

Figure 48: MultiRail TCP Benchmarks

This figure compares the performance of 2 x 10GE Linux Ethernet Channel Bonding, user-space Multi-Threaded TCP over 2 NICs and user-space MultiRail Library based TCP over two network rails for transferring a payload of 8MB.

### 7.2.2 Application Benchmarks

Accelerators including GPUs are increasingly becoming prevalent in e-Science. GPUs are used for computation, information visualization and collaboration between scientists. Efficient data streaming between GPUs on multiple nodes, and, streaming between the GPU and the CPU is critical in a GPU's performance in HPC clusters. We evaluate the efficacy of the RTK API on the performance of the "Netvideo" GPU streaming application. Netvideo is used for streaming 4K frames (4096 by 2048 pixels) of supercomputing e-Science simulations for remote and interactive visualization by scientists. This is very useful for steering simulations, especially in the petascale era.

4K visualization at 24 frames per second (fps) in RGBA format (32bit per pixel, with red, green, blue and alpha channels at 8-bit each) requires 6.4 Gbps of network bandwidth to stream from the rendering site to the display site. 'Netvideo', shown in Figure 12, consists of:

1. A sending application, streaming 4K frames from main memory (usually simulation data).
2. A receiving application that receives the frames, and downloads them to the graphics card for display. To optimize the pixel download, Netvideo uses pixel buffer objects (an OpenGL feature) that offers asynchronous DMA transfer to the GPU through its PCI-e link.
3. A 10G Myrinet interconnect between the sending and receiving machines for data transfer.

In this scenario with multiple devices requiring very high throughput, namely the network interface and graphics card, the rails approach helps in reducing the contention between the various devices resulting in an improved performance.

The results are as following:

1. The default Linux mechanism balances the interrupt load between devices (IRQ balance daemon) without any affinities and yields an end-to-end (from memory to remote display) bandwidth of 5.6 Gbps (approximately 21 fps).
2. Using RTK API, Netvideo achieves an end-to-end throughput of 6.2 Gbps (approximately 23 fps). This is an increase of 10.7% and is close to interactive data visualization.

# 8. CONCLUSION

LambdaRAM enables time-critical, high-performance data collaboration over both local and wide-area for data-intensive applications. In the NASA's climate modeling and analysis, LambdaRAM would result in faster weather prediction and improve the accuracy of the forecast by enabling models of higher complexity. Computational Chemistry, Genomics, Biomedical imaging suffer from similar bottlenecks and would benefit significantly from LambdaRAM. LambdaRAM's design empowers a scientist to focus on the science instead of spending time on data management issues. We strongly believe that LambdaRAM will aid in the design of efficient I/O systems for petascale applications and in the design of efficient LambdaGrids for data-intensive applications.

# CITED LITERATURE

[Abadi91] M. Abadi and L. Lamport. The existence of refinement mappings. Theoretical Computer Science, 82(2):253–284, May 1991.

[Abdulla08] P. A. Abdulla, N. B. Henda, G. Delzanno, and A. Rezine, "Handling parameterized systems with non-atomic global conditions", In Proceedings of the Ninth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI). Springer Verlag, 2008.

[Anderson94] E. A. Anderson and J. M. Neefe, "An exploration of network RAM," EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-98-1000, Dec. 9, 1994.

[Arons01] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck, "Parameterized verification with automatically computed inductive assertions", In CAV'01, pages 221–234. LNCS 2102, 2001.

[Balaban05] I. Balaban, Y. Fang, A. Pnueli, and L. Zuck. An invisible invariant verifier. In Proc. 17th Intl. Conference on Computer Aided Verification (CAV'05), pages 408–412, 2005.

[Bassi03] Alessandro Bassi, Micah Beck, Terry Moore, James S. Plank, Martin Swany, Rich Wolski, Graham Fagg, "The Internet Backplane Protocol: A Study in Resource Sharing", Future Generation Computing Systems, (19) 4, May, pp.551-561. Elsevier, 2003

[DeFanti03] DeFanti, T., Leigh, J., Yu, O., Krishnaprasad, N., Eliason, J., Alimohideen, J., He, E., "Quanta: a toolkit for high performance data delivery", Future Generation Computer Systems (2003), 01/01/2003 - 01/01/2003

[Fang04] Y. Fang, N. Piterman, A. Pnueli, and L. Zuck. Liveness with invisible ranking. In Proc. of the 5th conference on Verification, Model Checking, and Abstract Interpretation, volume 2937 of Lect. Notes in Comp. Sci., pages 223–238, Venice, Italy, January 2004. Springer-Verlag.

[Fang06] Y. Fang, K. L. McMillan, A. Pnueli, and L. Zuck. Liveness by invisible invariants. In FORTE, pages 356–371, 2006.

[Feeley95] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath, “Implementing global memory management in a workstation cluster,” in 15th ACM Symposium on Operating Systems Principles (SOSP 1995), ser. Operating System Review, vol. 29(5), Copper Mountain, Colorado, Dec. 3–6, 1995, pp. 201–212.

[GEOS] http://gmao.gsfc.nasa.gov/systems/geos5/. The goddard earth observing system model, version 5 (geos-5), nasa, goddard space flight center, 2007.

[GPFS] General Parallel File System, IBM Corporation, http://www-03.ibm.com/systems/clusters/software/gpfs/index.html

[Gu06] Yunhong Gu, Robert L. Grossman, Alex Szalay and Ani Thakar, “Distributing the Sloan Digital Sky Survey Using UDT and Sector”, Proceedings of e-Science 2006

[HDF] Hierarchical Data Format, HDF Group, http://hdf.ncsa.uiuc.edu/products/hdf4/index.html

[He02] Eric He, Jason Leigh, Oliver Yu and Thomas A. DeFanti, "Reliable Blast UDP : Predictable High Performance Bulk Data Transfer", Proceedings of IEEE Cluster Computing, Chicago, Illinois, September, 2002

[He03] He, E., et al, “Quanta: a Toolkit for High Performance Data Delivery over Photonic Networks,” Journal of Future Generation Computer Systems, Volume 19, Issue 6, August 2003, pp. 919-933.

[Hines06] M. R. Hines, J. Wang, and K. Gopalan, “Distributed Anemone: Transparent low-latency access to remote memory,” in 13th International Conference on High Performance Computing (HiPC 2006), ser. Lecture Notes in Computer Science, Y. Robert, M. Parashar, R. Badrinath, and V. K. Prasanna, Eds., vol. 4297. Bangalore, India: Springer, Dec. 18–21, 2006, pp. 509–521

[Jeong06] Jeong, B., Renambot, L., Jagodic, R., Singh, R., Aguilera, J., Johnson, A., and Leigh, J., "High-Performance Dynamic Graphics Streaming for Scalable Adaptive Graphics Environment," accepted by ACM/IEEE Supercomputing 2006.

[Koussih99] S. Koussih, A. Acharya, and S. Setia, "Dodo: a user-level system for exploiting idle memory in workstation clusters," in The Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC'99), Redondo Beach, California, Aug. 3–6, 1999, pp. 301–308.

[Krishnaprasad04] N. Krishnaprasad, V. Vishwanath, S. Venkataraman, A. Rao, L. Renambot, and A. Johnson, J. Leigh, "Juxtaview a tool for interactive visualization of large imagery on scalable tiled displays", In Proc. of IEEE Cluster 2004, San Diego, CA, 09/20/2004 - 09/23/2004 (CLUSTER 2004), 2004

[Leigh06] Leigh, J., Renambot, L., Johnson, A., Jeong, B., et al, "The Global Lambda Visualization Facility: An International Ultra-High-Definition Wide-Area Visualization Collaboratory," Journal of Future Generation Computer Systems, Volume 22, Issue 8, October 2006, pp. 964-971.

[Liao05] Wei-keng Liao, Kenin Coloma, Alok Choudhary, Lee Ward, Eric Russel, and Sonja Tideman, "Collective Caching: Application-Aware Client-Side File Caching". In Proceedings of the 14th International Symposium on High Performance Distributed Computing (HPDC), July 2005

[Manna95] Z. Manna and A. Pnueli. Temporal Verification of Reactive Systems: Safety. Springer-Verlag, New York, 1995.

[MAP] Modeling, Analysis and Prediction Project 2006, NASA Goddard Space Flight Center. http://map06.gsfc.nasa.gov

[MERRA] Modern Era Retrospective Reanalysis for Research and Applications (MERRA), http://gmao.gsfc.nasa.gov/research/merra/intro.php

[Moore01] Reagan W. Moore, "Data Management Systems for Scientific Applications" in "The Architecture of Scientific Software," pp. 273-284, Kluwer Academic Publishers, 2001

[Newman03] Harvey B. Newman, Mark H. Ellisman, John A. Orcutt, “Data-intensive e-science frontier research” , Communications of the ACM 46 (11) (2003) 68–77.

[NFS] Network File system, RFC 1094

[Owre99] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS System Guide. Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1999.

[Pakin07] Scott Pakin, Greg Johnson, “Performance Analysis of a User-level Memory Server”, In Proceedings of the 2007 IEEE International Conference on Cluster Computing (Cluster 2007), Austin, Texas, pp. 249–258, September 2007

[Pnueli96] A. Pnueli and E. Shahar, “A platform combining deductive with algorithmic verification”, In Rajeev Alur and Thomas A. Henzinger, editors, Proceedings of the Eighth International Conference on Computer Aided Verification CAV, volume 1102, page 184, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[Pnueli01] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In TACAS’01, pages 82–97. LNCS 2031, 2001.

[Pnueli03] A. Pnueli and T. Arons, “TLPVS: A PVS-based LTL verification system”, In Verification–Theory and Practice: Proceedings of an International Symposium in Honor of Zohar Manna’s 64th Birthday, Lect. Notes in Comp. Sci., pages 84–98. Springer-Verlag, 2003.

[PVFS2] Parallel Virtual File System 2, http://www.pvfs.org/

[Roussev06] V. Roussev, G. G. Richard III, and D. Tingstrom, “dRamDisk: Efficient RAM sharing on a commodity cluster,” in 25th IEEE International Performance, Computing, and Communications Conference (IPCCC 2006), Phoenix, Arizona, Apr. 10–12, 2006, pp. 193–198

[Sandstrom 03] T. A. Sandstrom, C. Henze, and C. Levit, The hyperwall. In Proc. Conference on Coordinated and Multiple Views in Exploratory Visualization, 124–133, July 2003.

[Seablom08] M. Seablom, “High productivity Science Through Distributed Collaborations”, 7th Annual ON*Vector International Photonics Workshop, San Diego, CA, February 2008.

[Shoshani02] Arie Shoshani, Alex Sim and Junmin Gu, Storage Resource Managers: Middleware Components for Grid Storage, In Nineteenth IEEE Symposium on Mass Storage Systems (MSS'02), 2002.

[Singh06] Singh, R., Schwarz, N., Taesombut, N., Lee, D., Jeong, B., Renambot, L., Lin, A., West, R., Otsuka, H., Peltier, S., Martone, M., Nozaki, K., Leigh, J., Ellisman, M., “Real-time Multi-scale Brain Data Acquisition, Assembly, and Analysis using an End to End OptIPuter”, Future Generation Computer Systems, 10/01/2006 - 10/31/2006

[Smarr03] Larry L. Smarr, Andrew A. Chien, Tom DeFanti, Jason Leigh, Philip M. Papadopoulos, "The OptIPuter," Communications of the ACM, Volume 46, Issue 11, November 2003, pp. 58-67

[Starlight] Starlight – The Optical Startap. http://www.startap.net/starlight/

[Vishwanath06] Vishwanath, V., Leigh, J., He, E., Brown, M. D., Long, L., Renambot, L., Verlo, A., Wang, X., DeFanti, T. A., “Wide-Area Experiments with LambdaStream over Dedicated High-bandwidth Networks,” IEEE INFOCOM, April 2006.

[Vishwanath08] V. Vishwanath, L. Zuck and J. Leigh. “Specification and Verification of LambdaRAM – A Wide-Area Distributed Cache for High Performance Computing”. In the proceedings of the 6th IEEE/ACM Conference on Formal Methods and Models for Codesign (MEMOCODE) 2008, June 5-7 2008, Anaheim, CA, USA

[Xiong05] Xiong, C., Leigh, J., He, E., Vishwanath, V., Murata, T., Renambot, L., and DeFanti, T., “LambdaStream – a Data Transport Protocol for Streaming Network-intensive Applications over Photonic Networks,” Proceedings of The Third International Workshop on Protocols for Fast Long-Distance Networks, Lyon, France, Feb. 2005.

[Zhang03] Zhang, C., Leigh, J., DeFanti, T.A., Mazzucco, M., Grossman, R., “TeraScope: Distributed Visual Data Mining of Terascale Data Sets over Photonic Networks”, Journal of Future Generation Computer Systems (FGCS), 08/01/2003 - 08/01/2003

[Zuck04] L. Zuck and A. Pnueli. Model checking and abstraction to the aid of parameterized systems. Computer Languages, Systems, and Structures, 30(3–4): 139–169, 2004.

## VITA

**NAME** Venkatram Vishwanath

**EDUCATION**

| | |
|---|---|
| 2004 – 2009 | Ph.D., Computer Science, University of Illinois at Chicago, Chicago, Illinois |
| 2000 – 2003 | M.S., Computer Science, University of Illinois at Chicago, Chicago, Illinois |
| 1995 – 1999 | B.S., Electronics Engineering, University of Mumbai, India |

**PUBLICATIONS**

Book Chapters

[1] J. Leigh, A. Johnson, L. Renambot, V. Vishwanath, T. Peterka and N. Schwarz, "*Visualization of Large-Scale Distributed Data",* A Book Chapter to appear in "Data Intensive Distributed Computing: Challenges and Solutions for Large-Scale Information Management", Editor: T. Kosar, IGI Global Publishing, USA, 2009

Journal Articles

[2] V. Vishwanath, R. Burns, J. Leigh and M. Seablom. *"Accelerating Tropical Cyclone Analysis using LambdaRAM, A Distributed Data Cache Over Wide-Area Ultra-Fast Networks",* Future Generation Computer Systems/The International Journal of Grid Computing: Theory, Methods and Applications, Elsevier B.V., February 2009.

[3] T. DeFanti, J. Leigh, L. Renambot, B. Jeong, A. Verlo, L. Long, M. Brown, D. Sandin, V. Vishwanath, Q. Liu, M. Katz, P. Papadopoulos, J. Keefe, G. Hidley, G. Dawe, I. Kaufman, B. Glogowski, K. Doerr, R. Singh, J. Girado, J. Schulze, F. Kuester, and L. Smarr, "*The OptIPortal - A scalable visualization, storage, and computing interface device for the OptiPuter",* Future Generation Computer Systems/The International Journal of Grid Computing: Theory, Methods and Applications, Elsevier B.V., February 2009.

[4] J. Leigh, L. Renambot, A. Johnson, B. Jeong, R. Jagodic, N. Schwarz, B. Svistula, R. Singh, J. Aguilera, X. Wang, V. Vishwanath, B. Lopez Silva, D. Sandin, T. Peterka, J. Girado, R. Kooima, J. Ge, L. Long, A. Verlo, T. DeFanti, M. Brown and D. Cox, *"The Global Lambda Visualization Facility: An International Ultra-High-Definition Wide-Area Visualization Collaboratory",* Future Generation Computer Systems/The International Journal of Grid Computing: Theory, Methods and Applications, Elsevier B.V., October 2006.

[5] A. Hirano, L. Renambot, B. Jeong, J. Leigh, A. Verlo, V. Vishwanath, R. Singh, J. Aguilera, A. Johnson, T. DeFanti, L. Long, N. Schwarz, M. Brown, N. Nagatsu, Y. Tsukishima, M. Tomizawa, Y. Miyamoto, M. Jinno, Y. Takigawa, O. Ishida, *"The First Functional Demonstration of Optical Virtual*

*Concatenation as a Technique for Achieving Terabit Networking",* Future Generation Computer Systems/The International Journal of Grid Computing: Theory, Methods and Applications, Elsevier B.V., October 2006.

[6] G. W. Pieper, T. A. DeFanti, Q. Liu, M. Katz, P. Papadopoulos, J. Keefe, G. Hidley, G. Dawe, I. Kaufman, B. Glogowski, K. Doerr, J. P. Schulze, F. Kuester, P. Otto, R. Rao, L. Smarr, J. Leigh, L. Renambot, A. Verlo, L. Long, M. Brown, D. Sandin, V. Vishwanath, R. Kooima, J. Girado, B. Jeong, "*Visualizing Science: The OptIPuter Project,*" SciDAC Review, Issue 12, Spring 2009, published by IOP Publishing in association with Argonne National Laboratory, for the US Department of Energy, Office of Science, pp. 32-41.

## Conference and Workshop Publications

[7] V. Vishwanath, S. Nam, L. Renambot, J. Leigh, H. Takahashi, M. Takizawa, S. Kobayashi, O. Kamatani, O. Ishida, *"Achieving Large Bandwidth by Leveraging Parallelism in End-Hosts and Networks"*, Proceedings of the IEEE Photonics Society Summer Topicals Conference, Newport Beach, California, 2009

[8] V. Vishwanath, J. Leigh, T. Shimizu, S. Nam, L. Renambot, H. Takahashi, M. Takizawa, O. Kamatani, *"The Rails Toolkit (RTK) - Enabling End-System Topology-Aware High End Computing"*, The 4th IEEE International Conference on e-Science, Indianapolis, USA, 12/07/2008 - 12/12/2008

[9] V. Vishwanath and L. Zuck, *"Verification Requirements of Data Intensive High Performance Computing Middleware"*, Proceedings of the (EC)^2: Exploiting Concurrency Efficiently and Correctly workshop held in conjunction with Computer Aided Verification (CAV) 2008, Princeton, NJ

[10] V. Vishwanath, L. Zuck and J. Leigh, *"Specification and Verification of LambdaRAM – A Wide-Area Distributed Cache for High Performance Computing",* Proceedings of the 6th IEEE/ACM Conference on Formal Methods and Models for Codesign (MEMOCODE) 2008, June 5-7 2008, Anaheim, CA, USA

[11] V. Vishwanath, T. Shimizu, M. Takizawa, K. Obana, J. Leigh, *"Towards Terabit/s Systems: Performance Evaluation of Multi-Rail Systems"*, Proceedings of the 20th IEEE/ACM Supercomputing Conference 2007 (SC 2007), Reno, Nevada, November 12-18 2007.

[12] J. Brassil, J. Leigh, J. Mambretti, B. Mark, R. McGeer, L. Renambot, L. Roberts, S. Schwab and **V.** Vishwanath, *"The Case for Bonded Lambdas (BoLas)"*, In DARPA's Workshop for Routing Protocols and Management (RPM) For High Capacity Networks, October 2007.

[13] W. Feng, V. Vishwanath, J. Leigh and M. Gardner, *"High-Fidelity Monitoring in Virtual Computing Environments"*, Proceedings of the International Conference on the Virtual Computing Initiative, Research Triangle Park, NC, May 2007.

[14] V. Vishwanath, T. Shimizu, M. Takizawa, K. Obana, J. Leigh, *"Towards Terabit/s Systems: Performance Evaluation of Multi-Rail Systems"*, Proceeding of The High-Speed Networking Workshop: The Terabits Challenge, co-located with the 26th IEEE INFOCOM Conference, Anchorage, Alaska, 05/08/2007 - 05/11/2007

[15] J. Ge, T. Peterka, R. L. Kooima, V. Vishwanath, D. J. Sandin and A. Johnson, *"A Distributed Volume Rendering Pipeline for Networked Virtual Reality"*, Proceedings of the International Workshop on Network-based Virtual Reality and Tele-existence (INVITE), May 2007.

[16] E. He, X. Wang, V. Vishwanath and J. Leigh, *"AR-PIN/PDC: Flexible Advance Reservation of Intradomain and Interdomain Lightpaths"*, Proceedings of the 49th IEEE Conference of GLOBECOM 2006, San Francisco, California, U.S.A., November 2006.

[17] X. Wang, V. Vishwanath, B. Jeong, R. Jagodic, E. He, L. Renambot, A. Johnson and J. Leigh, *"LambdaBridge: A Scalable Architecture for Future Generation Terabit Applications"*, Proceedings of IEEE Conference of BROADNETS, San Jose, California, U.S.A., October 2006.

[18] V. Vishwanath, J. Leigh, E. He, M. D. Brown, L. Long, L. Renambot, A. Verlo, X. Wang, T. A. DeFanti, *"Wide-Area experiments with LambdaStream over dedicated high-bandwidth networks"*, Proceedings of 24th IEEE Conference of INFOCOM, Barcelona, Spain, May 2006

[19] V. Vishwanath, P. Balaji, W. Feng, J. Leigh and D. K. Panda, *“A Case for UDP Offload Engines in LambdaGrids”*, Proceedings of The Fourth International Workshop on Protocols for Fast Long-Distance Networks (PFLDNet 2006), Nara, Japan, 02/02/2006 - 02/03/2006

[20] V. Vishwanath, L. Renambot, J. Leigh, D. Lee, A. Nayak, N. Schwarz, L. Long, A. Verlo, R. Singh and F. Dijkstra, *“Interactive Remote Visualization of Large High-Resolution Time-Varying Geophysical and Biological Datasets using LambdaRAM”,* Proceedings of the 18th IEEE/ACM Supercomputing Conference 2005 (SC 2005), Seattle, Washington, November 12-18 2005.

[21] C. Xiong, J. Leigh, E. He, V. Vishwanath, T. Murata, L. Renambot and T. DeFanti, *“LambdaStream - a Data Transport Protocol for Streaming Network-intensive Applications over Photonic Networks”*, Proceedings of the 3rd International Workshop on Protocols for Long-Distance Networks (PFLDNet), Lyon, France, February 2005.

[22] N. Schwarz, S. Venkataraman, L. Renambot, N. Krishnaprasad, V. Vishwanath, J. Leigh, A. Johnson, G. Kent and A. Nayak, *“Vol-a-Tile - a Tool for Interactive Exploration of Large Volumetric Data on Scalable Tiled Displays”*, Proceedings of IEEE Conference on Visualization 2004 (Viz), Austin, Texas, October 2004.

[23] L. Renambot, A. Rao, R. Singh, B. Jeong, N. Krishnaprasad, V. Vishwanath, V. Chandrasekhar, N. Schwarz, A. Spale, C. Zhang, G. Goldman, J. Leigh and A. Johnson, *“SAGE: The Scalable Adaptive Graphics Environment”,* Proceedings of the 4th Workshop on Advanced Collaborative Environments (WACE), Nice, France, September 2004.

[24] N. Krishnaprasad, V. Vishwanath, S. Venkataraman, A. Rao, L. Renambot, J. Leigh, A. Johnson and B. Davis, *“JuxtaView – a Tool for Interactive Visualization of Large Imagery on Scalable Tiled Displays”,* Proceedings of the 6th IEEE International Conference on Cluster Computing (CLUSTER), San Diego, CA, September 2004.