DRepl: Optimizing Access to Application Data for Analysis and Visualization

Latchesar Ionkov
Los Alamos National Laboratory
lionkov@lanl.gov

Michael Lang
Los Alamos National Laboratory
mlang@lanl.gov

Carlos Maltzahn
University of California, Santa Cruz
carlosm@cs.ucsc.edu

***Abstract*—Until recently most scientific applications produced data that is saved, analyzed and visualized at later time. In recent years, with the large increase in the amount of data and computational power available there is demand for applications to support data access in-situ, or close-to simulation to provide application steering, analytics and visualization. Data access patterns required for these activities are usually different than the data layout produced by the application. In most of the large HPC clusters scientific data is stored in parallel file systems instead of locally on the cluster nodes. To increase reliability, the data is replicated, usually using some of the standard RAID schemes. Parallel file server nodes usually have more processing power than they need, so it is feasible to offload some of the data intensive processing to them. DRepl project replaces the standard methods of data replication with replicas having different layouts, optimized for the most commonly used access patterns. Replicas can be complete (i.e. any other replica can be reconstructed from it), or incomplete. DRepl consists of a language to describe the dataset and the necessary data layouts and tools to create a user-space file server that provides and keeps the data consistent and up to date in all optimized layouts.**

***Keywords*-data storage; data replication; fault tolerance; exascale; DISC**

I. Introduction

The amount of data produced by scientific applications increases with the size of the high-performance supercomputers they run on. The future exascale systems will require hundreds of petabytes storage only for scratch space [1]. It may be prohibitive to transfer all data produced by exascale simulations outside of the compute cluster. These issues made in-situ and close-to analytics and visualization solutions an important research topic. Analyzing and steering the simulation while it is running can reduce the resources (both computational and storage) used. In most cases the visualization and analytics applications need a small part of the data produced by the scientific application, but because the data layout is optimized to increase the performance of the simulation, finding and reading the required data is slow and may interfere with the data producer.

To improve data availability and reliability, storage systems use some form of data replication. Most commonly a variant of RAID [2] is used. The advantage of using RAID is that it is well supported, including in hardware. Each replica uses additional storage and requires more electrical power, but because all replicas are identical, these systems usually don't use the multiple replicas to improve storage performance. The storage nodes are getting smarter, with faster CPUs and more cores, even though they are not fully utilized. With "cheap" and available computational power, it makes sense to have replicas with different data layouts. The data producers and data consumers can use the replica that is best optimized for their access pattern.

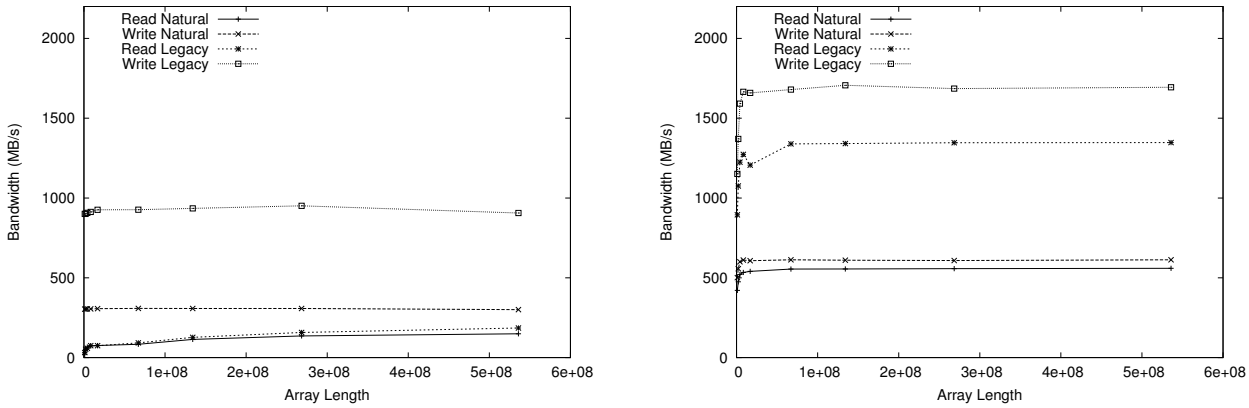
DRepl tries to improve the performance of the visualization and analysis tools while keeping the amount of storage and reliability guarantees the same as when using other data replication mechanisms. DRepl is transparent to the applications and doesn't require any modifications. It runs as a file server that provides different files for each layout of the data desired. The data layouts (views) can be stored in a file (replica) on the underlying parallel file system (materialized), or can be virtual (non-materialized). If a view is materialized, reading from its file reads the data from the real file on the parallel file system. Reading from non-materialized view uses data from one of the materialized views and converts the data on the fly. When writing data to a view, DRepl updates all replicas. Depending on the concurrency model, the updates can be synchronous or asynchronous.

DRepl defines a language that is used to describe the dataset, views and replicas. The description is used to generate source code to convert between data layouts. The code produces a user-level 9P2000 [3] file server. This approach allows support for legacy scientific applications, without any modifications in their code. As long as one of the views matches the legacy data layout, the application can continue to work as before, even if the replicas store the data in a more portable and convenient layouts, like HDF5 [4] or NetCDF [5].

DRepl is a work in progress. This paper describes the experiments run in order to prove the feasibility of our approach as well as the design and ongoing implementation of DRepl.

II. Motivation

In order to evaluate feasibility and performance of our approach, we constructed and ran experiments with prototype implementations of our ideas.



(a) RAID-0 HDD

(b) Fusion-IO Disk

Figure 1: Legacy vs. Natural access to legacy data

A. Dataset

We used a simple dataset in which the scientific application stores three values for each "point" of the simulation. The number of "points" is configurable. The range for our experiments was from 1 million to 170 million.

```
struct Point {
    a    float
    b    float
    c    float
}

Point    data[N]
```

B. Views

We define three views (data layouts) of the dataset:

1) Natural/default (abc):

```
Point    data[N]
```

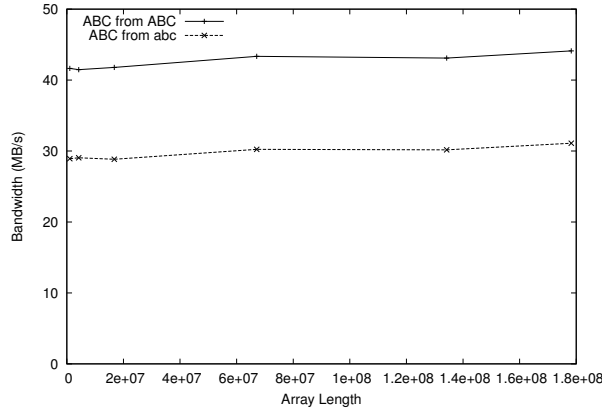


Figure 2: Reading legacy view from legacy and natural layouts

2) Legacy (ABC): Most of the legacy applications, especially the ones written in Fortran, store separate arrays for each value.

```
float    a[N]
float    b[N]
float    c[N]
```

3) Visualization (b): In most cases the visualization requires only some of the values.

```
float    b[N]
```

C. Accessing legacy data

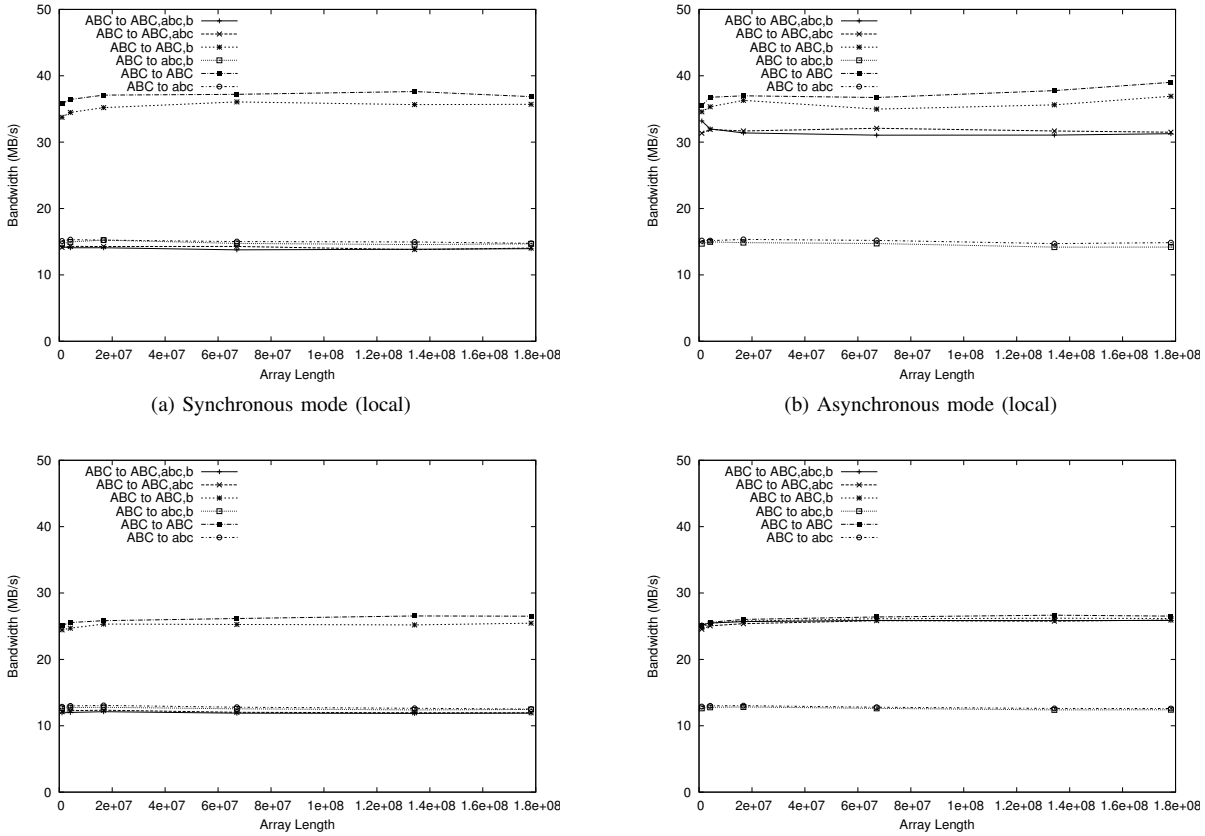
In this experiment the data is stored in the legacy (ABC) format. We implemented programs that use the legacy and natural layouts internally. They read and write the data from/to a file that uses the legacy view. If the internal representation is also legacy, the program uses a single operation for each array. When the program uses the natural representation, the legacy view is accessed in chunks of 1024 elements.

Figure 1 shows the results of the test on RAID-0 array of two SAS disks vs. Fusion-IO card. Write operations are faster than the reads in both cases, because the write operation returns once the data is cached to the OS data buffers and is not necessarily on written on the disk yet.

On rotational RAID-0 array, the difference between converting the data to the natural view when reading is less than 10 percent, while converting it for writing is three times slower.

On the Fusion-IO card, both reading and writing data to legacy format is more than twice as fast as converting it from legacy to natural format.

The small difference between the two modes of reading from the RAID-0 array could be due to the cache present on the RAID controller. Bypassing the RAID-0 controller



(a) Synchronous mode (local)

(b) Asynchronous mode (local)

(c) Synchronous mode (remote)

(d) Asynchronous mode (remote)

Figure 3: Write performance for legacy format

on the test machines will verify this, but was impossible due time constraints but will be a near-term extension to this work. From the initial investigation the approach shows promise with improvements in both read and write times on two different physical disk media, with a very simple data access pattern.

D. File server that provides multiple data views

We implemented a simple user-level file system that serves each view as a separate file. When the file server is mounted in a directory, it contains three files:

- ABC Data in legacy format. First $4 * N$ bytes contain the data for the array a, followed by $4 * N$ bytes for array b, and then $4 * N$ bytes for array c.
- abc Data in natural format. Contains N elements, each 12 bytes long with the values of a, b and c for that element.
- b Data in the visualization format. Contains N elements, each 4 bytes long with values only of b.

The file server is written in the Go [6] language, using the go9p [7] library. It allows materialized and non-materialized views for each of the three views as well as synchronous and asynchronous updates to the other views when data is written.

Converting data from one view to another can be very inefficient. For example, if a program writes to the first 400 bytes of file `ABC`, updating the first 100 values of a, the operation needs to be converted to 100 writes to file `abc`, each writing 4 bytes with stride 12 bytes. Even using functions like `writev` can be slow, because of the time and resources required to prepare the data for the call. Using processors close to the data can improve the performance somewhat. In order to improve it even further, we use `mmap` to map the content of the materialized views to memory. The conversion between different views then is equivalent to conversion of data in memory.

We ran sets of experiments varying the following parameters:

- Array Length
  We run tests with array lengths from 1 million to 170 million elements.
- Materialized views
  We tried six different combinations of materialized views: `ABC/abc/b`, `ABC/abc`, `ABC/b`, `abc/b`,

ABC, and abc.

Synchrony of updates
: We tried synchronous vs. asynchronous updates. At least one of the replicas is updated synchronously before the write operation completes. Reading performance is not affected by the synchrony parameter.

Remove vs. Local
: We tried running the file server on the same computer as the program that accesses the data, as well as on another computer.

Read and write operations access all data in the view.

The experiments were done on 4 socket, 12 core servers (total 48 cores) with 128GB of RAM. The interconnect was Infiniband. The OS buffer cache was cleaned between every run.

1) Local File Server: We chose to run our tests on traditional rotational disks, configured in a RAID-0 array and Fusion-IO card that achieves one of the highest I/O bandwidths at present, and likely has the properties of the hardware present in the future exascale systems. Figure 2 shows reading speed for the legacy format when ABC and abc views are materialized. There is 31 percent penalty when data needs to be converted from the natural to the legacy layout. The results for converting the opposite way is comparable.

Figure 3 shows the write performance for different materialized views as well as synchronous vs. asynchronous mode. In synchronous mode (Fig. 3a), the presence of replica that is not in the format of the view through which the data is updated decreases the write speed by more than half. The presence of a replica with similar layout (ABC and ABC,b) has negligible effect on the performance. In asynchronous mode (Fig. 3b), the write performance is degraded only if there is no replica with layout similar to the view through which the data is updated. Even though the b replica has similar layout to ABC, it is not a complete replica and therefore the replica abc is chosen to be updated synchronously.

2) Remote File Server: Running the file server on a remote node (Fig. 3c and 3d) doesn't change the performance results drastically. As expected, the difference between ABC->ABC and ABC->abc is decreased due to the usage of additional hardware. We assume that for applications that utilize all cores on the compute node, the difference between running the file server locally and remotely, will be more pronounced.

III. Design: DRepl Language

The DRepl language allows definition of datasets and views. It allows definition of custom types based on a set of primitive types, as well as arrays. It is loosely based on the type and variable definition in the Go language. The language is designed to be able to represent native application datasets with rich views to allow visualization and analytics optimized access to data of interest. NetCDF [5], HDF5 [4] and local large-scale HPC applications data formats were investigated to ensure good representation of real datasets.

A. Dataset Section

The dataset section defines the types that are used in the dataset as well as the variables that make up the dataset.

1) Primitive Types: DRepl defines 7 primitive types:

int8	1-byte signed integer
int16	2-byte signed integer
int32	4-byte signed integer
int64	8-byte signed integer
float32	single-precision floating point number
float64	double-precision floating point number
string*[0-9]+*	variable size string of characters

The suffix of the string type defines the maximum size of the string that can be stored.

```
PrimitiveType = "int8" | "int16" |
   "int32" | "int64" | "float32" |
   "float64" | "string"
```

2) Structs: Multiple elements can be arranged in a struct. A name needs to be assigned to each of the elements.

```
StructType = "struct" "{" { FieldDecl ";" } "}"
FieldDecl = IdentifierList Type
IdentifierList = identifier { "," identifier }
```

Example:

```
struct {
   a        float64
   b, c     float32
}
```

3) Arrays: Array is a numbered sequence of elements of the same type. Arrays can be single- or multi-dimensional, of fixed or variable size. Only one of the dimensions of a multi-dimensional array can be of variable size. Instances of the variable-sized arrays need to specify a variable that stores the size of the array.

Variable-sized types can't be used as element types.

```
ArrayType = "[" ArrayLengths "]" Type
ArrayLengths = Expression { "," Expression }
```

The Expression in ArrayLengths can be arithmetic expression containing integer constants (named or unnamed), or it can be the name of an already defined integer variable. In the latter case, the variable contains the size of the array in that dimension.

In the example below, b is a two-dimensional 5×5 array, and c is variable-sized array with size stored in the sz variable.

```
var sz   int32
var b    [5,5]float64
var c    [sz]float32
```

4) Named Types: The user can assign names to the defined custom types. The names allow the user to use "shortcuts" when using a type more than once.

```
TypeDecl = "type" identifier Type
```

5) Types:

```
Type = identifier | CustomType
CustomType = ArrayType | StructType
```

Unlike most languages, DRepl allows a type to be referenced before it is defined, so there is no need for forward declaration mechanisms.

6) Variables and Constants: A variable is a named instance of a type. The names of the variables need to be unique.

Constants are named values that can't change. They don't use storage space.

```
VarDecl = "var" IdentifierList Type
IdentifierList = identifier { "," identifier }
ConstDecl = "const" identifier "=" Expression
```

7) Dataset: A dataset is a collection of types, variables and constants.

```
Dataset = "dataset" "{"
   { TypeDecl | VarDecl | ConstDecl } "}"
```

This example shows how the dataset from section II is defined in the DRepl language:

```
dataset {
    const N = 1000000

    type Point struct {
        a, b, c    float32
    }

    var data [N]Point
}
```

B. View Section

Views define subsets of the dataset. No new types can be defined in the view section, unless they are subtypes of the dataset types. The user can define "substructs", i.e. structs that have only some of the fields of a dataset struct, or "slices" – parts of an array type defined in the dataset section.

The variables defined in the view should be based on the variables in the dataset. Each view variable has corresponding dataset variable and is of the same type as the dataset variable, or compatible subtype.

1) View Struct:

```
ViewStruct = "{" { ViewFieldDecl ";" } "}"
ViewFieldDecl = IdentifierList ViewType
```

Example of usage of a view struct:

```
var b [] {
    b
} = data
```

The view variable `b` is defined as an array with the same size as the dataset variable `data`, but each element of the array contains only the field `b` from the original `data` elements.

2) View Slice: The view slice contains only part of the elements of the original dataset array.

```
ViewSlice = "[" SliceLengths "]"
SliceLengths = Expression { "," Expression }
```

The `Expression` in the slice length definition can be arithmetic expression containing temporary variable names that are used to express the slice. For example, the snippet below defines a view variable that contains every 5^{th} element of the `data` array.

```
var d[i] = data[i*5]
```

3) Named View Types: As in the dataset section, the user can assign a name to any defined view type, in order to simplify the view definition. The example below shows the definition of type `Subpoint` that is based on dataset type `Point` but contains only fields `a` and `c`.

```
type Subpoint {
    a, c
} Point
```

4) More Complex Examples: A view variable that contains every 3^{rd} `b` field of the elements of `data`:

```
var d [i] {
    b
} = data[i*3]
```

The same result with defining view type:

```
type PointB { b } Point
var d [i]PointB = data[i*3]
```

Reversing the column and row order in a two-dimensional array:

```
var d [i,j] = a[j,i]
```

5) View Declaration:

```
ViewDecl = "view" identifier ReadOnlyFlag "{"
   { ViewTypeDecl | VarTypeDecl } "}"
ViewDecl = "view" FileName
ReadOnlyFlag = "read-only" | <nothing>
```

The view can be defined in the same file as the dataset, or it can be defined in a separate file and the file name can be specified to include to content of the file while parsing the dataset definition.

The data in the dataset can't be updated via read-only views.

C. Replica Section

Replica defines how the views are going to be laid out in the file system. Replica is a sequence of one or more views. The replicas can be "complete" or "incomplete". A complete replica contains all the data from the dataset. If the view only contains a subset of the dataset, the remaining data is appended at the end to ensure a complete replica. The views are stored in the file in the order they appear in the replica declaration.

Each materialized view has to belong to a replica. The views that are not part of any replica are non-materialized and when read from them, the data is transformed on-the-fly.

Currently the language definition doesn't allow specification whether the primitive types are written on the disk as big- or little-endian. This can change in the future.

```
ReplicaDecl = "replica" identifier
   CompleteFlag ArrayOrderFlag "{"
   { ReplicaView} "}"
ReplicaDecl = "replica" FileName
CompleteFlag = "complete" | <nothing>
ArrayOrderFlag = "row-major" | "column-major" |
                 <nothing>
ReplicaView = "view" identifier
```

IV. Design: Replica Layout

Each replica is a separate file, or directory, with name, the name specified when declaring the replica. If any of the views of the replica contains variable-sized data, the replica is a directory with multiple files in it. Otherwise, data for all views is stored in the same file.

The data from the views laid out in the order they are declared in the replica definition. Each view starts at offset divisible by 8, and padding is added between the views if the previous view's size is not divisible by 8.

A. Primitive Types

Each of the primitive types starts at offset that naturally aligns to its type.

type	alignment	size
int8	1	1
int16	2	2
int32	4	4
int64	8	8
float32	4	4
float64	8	8
string$N$	2	2 + N

The string content is preceded by an int16 value that specifies the actual length of the string.

B. Structs

The fields in a struct are laid out sequentially without any explicit padding between them, or at the end of the struct. The alignment rules for the type of the first field define the alignment requirements for the struct.

C. Arrays

Elements of an array are laid out sequentially without any padding between them or at the end of the array. The alignment rules of the element type define the alignment requirements for the array type.

D. Variable-size data

Because POSIX file systems don't allow insertion of data in the middle of file, DRepl needs to split replicas that contain variable-size data into more than one files. A variable-sized type is laid out using the same rules as any other type. The data defined after this type is stored in a new file.

DRepl uses numerical names for the files comprising a replica, starting from 0 and increasing by 1 after each variable-sized data is laid out.

V. Design: Transformation Rules

When data is written to one of the views, or data is read from non-materialized view, DRepl needs to transform the data from one data layout to another. First, the code needs to identify which variable the data belongs to, and what is its type. Then it needs to use the transformation rules associated with that type to transform the data to the related variables in the other views (or even the same view) in case of writing, or use the other related variables in order to construct the requested data, in case of reading.

While processing the dataset description, for each variable in the dataset, the DRepl parser locates all related variables in the view. For each pair of related variables it produces a map on how to transform data from one variable to the other. If a view is marked as read-only, the maps describing how to transform from its variables to other are not created.

A. Conversion Map

The conversion map consists of block descriptions. Each block has defined size and list of blocks its data transforms to.

```
    type Block struct {
offset int64
        size    int64
        list of Destination
    }

    type Destination struct {
        fname   string
        offset  int64
    }
```

For example, for a dataset description:

```
dataset {
   var p struct {
      a, b, c float32
   }
}

view "default" {
   var p
}

view "viz" {
   var pa { a } = p
   var pba { b, a } = p
}
```

The conversion map for "default" is:

```
Block { // p.a
   offset=0
   size=4
   list of {
       Destination { fname="viz" offset=0 }
       Destination { fname="viz" offset=8 }
   }
}

Block { // p.b
   offset=4
   size=4
   list of {
       Destination { fname="viz" offset=4 }
   }
}
```

A special block is used to describe arrays. In addition to the offset, it contains the element size as well as the number of elements. If the number of elements is specified as 0, the array is variable-sized and uses all remaining space in the file.

```
type ArrayBlock struct {
    offset  int64
    elsize  int64
    elnum   int64
    list of struct {
        destoffset func(offset int64) int64
        list of Block or ArrayBlock
    }
}
```

The function `destoffset` is used to convert the indices between differently sliced arrays. It also allows adjustments between slices which elements are not of the same size (for example when the original array is an array of struct and one of the slices contains only some of the struct fields). If the function returns negative number of specified offset, the element of that offset is to be skipped and doesn't exist in the other slice.

The offsets in the Blocks and Destinations in the ArrayBlock description are relative to the offset of the currently processed element.

For example, for a dataset description:

```
dataset {
   type Point struct {
       a, b, c float32
   }

   var data  [50]Point
}

view "default" {
   var data
}

view "viz" {
   var bb  [i] { b } = data[i*5]
}
```

View "viz" contains only the `b` field of every 5^{th} element of array `data`. The conversion map for default is:

```
ArrayBlock {
   offset=0
   elsize=12
   elnum=50
   list of {
       destoffset=func(o int64) int64 {
          idx:=o/12
          if idx%5!=0 {
              return -1
          }
          return idx*4
       }
       list of Block {
          offset=4
          size=4
          list of {
             Destination { fname="viz" offset=0 }
  }
       }
    }
}
```

The conversion map for a replica is constructed by the conversion maps for all the views that belong to it.

B. Updating Data

The dataset can be updated via a `write` operation to one of the replica files. The `write` operation receives as arguments the file offset and the number of bytes written. The file server finds all Blocks and ArrayBlocks that belong to the specified range and applies the conversion rules specified in the blocks.

C. Reading Data through Non-materialized View

Non-materialized, read-write views still have conversion maps created. Using the (offset, count) pair, the appropriate blocks are found in the conversion map and the data is copied from one or more of the materialized views.

VI. Conclusion

DRepl provides a novel method for optimized access to application datasets that are read and written with multiple contrasting patterns. Initial investigations showed increased performance in both read and writes on various physical media. A prototype file system was implemented and the language was designed and specified to allow construction of multiple dataset views. These views are displayed as separate but consistently updated files from the DRepl file server.

DRepl's approach allows flexibility on where the conversion between replicas is being performed. The file server can be run locally on the node that runs the scientific application, on the parallel file system nodes, or on nodes that perform I/O aggregation and forwarding. It works well with legacy scientific applications without imposing changes in their code.

Using multiple complete replicas of the data increases the reliability of the storage system.

VII. Related Work

There has been quite a body of work using middle-ware to optimize reading or writing of application data. PLFS [8]

is a transparent layer optimized for writing of parallel application checkpoints. It allows each process of a parallel application to believe it is writing to the a single file while the PLFS middle-ware separates these writes to disjoint files, extensions to this work are focused on increasing read performance. ADIOS [9] is an API that allows efficient data reordering that is transparent to the application but requires modification of the application to use the API. Further work with ADIOS [10] has reordered data using space filling curves for faster access. MPI IO [11] also has the concept of defined views and requires modification of the parallel application, but provides collective I/O operations and strided access of data. MPI I/O has two-phase I/O option which allows reordering of data as it is being accessed. Semantic file systems try to represent the data by the information contained in the file rather then its position in the file system. The SFS semantic file system [12] was a layer on top of NFS that would create files based on user defined transducers. The transducers would allow retrieval of pieces of files, although this work was not focused on large application data. The ATTIC [13] system allowed transparent access to compressed files. UNIX systems have presented changing information through virtual files, examples such as /dev, /proc [14] and also in Plan 9 [15], this is analogous to in-situ access to the state of various data structures in the kernel. Long distance visualization [16] uses multi-resolution data views to allow reasonable response times. In this scenario, when looking for an area of interest the resolution is sub-sampled to allow fast scanning, and when an area of interest is selected the high resolution data is then streamed in. None of the related work combines multiple semantic views with replicas to provide the configurability and resilience of our proposed solution.

VIII. Future Work

We are still working on implementing the full version of the DRepl file server and testing it with real applications. Once it implemented and working correctly, we are planning to work on optimizing it for highly parallel loads.

DRepl can be used for easier conversion of data produced by legacy applications to standard scientific formats like HDF-5. Once the legacy layout is defined in the DRepl dataset language, it is easy to produce a HDF-5 dataset from it. We are planning to write a HDF-5 back-end to our DRepl file server, that allows unmodified legacy application to store and read the dataset from HDF-5 file.

Another possible extension is adding support to HDF-5 so it can work better with the DRepl file server. It is not possible to describe HDF-5 file layout with our current language.

We need to do more investigations on whether DRepl has enough features to cover all legacy formats that we need to support. One important feature that is missing is the ability to specify the endianness of the data in the file. We may also need to extend the support for data alignment.

When running in asynchronous mode, DRepl doesn't provide any guarantees when the data in the other replicas will become up-to-date. We are planning to experiment with update-on-close (i.e. once the file that is used to update the dataset is closed, all replicas are synced).

In order to improve the performance when using non-materialized views, we need to define which of the replicas is closest to the view. One approach would be to base the closeness function on the number of blocks that need to be used for the conversion. The "closest view" can be coarse-grained (for the whole non-materialized view), or finer-grained (for each variable in the view).

The data layout knowledge can be used to improve the performance of prefetching schemes and make decisions on what striping approaches to use for the replica files on the parallel file systems. We are planning to do more investigations in that field in the future.

Integrating the DRepl language parser with the file server would allow dynamic addition, modification and removal of views. That feature will make it easier for scientists to visualize and check the simulation progress while it is running as well as look for abnormal behaviors.

IX. Acknowledgments

We would like to thank Jim Ahrens, and Scott Brandt for there helpful insights for this paper. This work was performed at the Ultrascale Systems Research Center (USRC) which is a collaboration between Los Alamos National Laboratory and the New Mexico Consortium(NMC). NMC provides the environment to foster collaborative research between LANL, universities and industry allowing long-term interactions in Los Alamos for professors, students and industry visitors.

This work was supported in part by the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC52-06NA25396 with Los Alamos National Security, LLC.

This paper has assigned LANL publication number: LA-UR-11-11589.

References

[1] G. Grider, "Exa-scale FSIO Can we get there? Can we afford to?" presented at the 7th IEEE International Workshop on Storage Network Architecture and Parallel I/Os, 2011.

[2] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '88. New York, NY, USA: ACM, 1988, pp. 109–116. [Online]. Available: http://doi.acm.org/10.1145/50202.50214

[3] "Introduction to the 9p protocol," *Plan 9 Programmer's Manual*, vol. 3, 2000.

[4] T. H. Group, "Hierarchical data format version 5." [Online]. Available: http://www.hdfgroup.org/HDF5

[5] U. P. Center, "Network common data form." [Online]. Available: http://unidata.ucar.edu/software/netcdf/

[6] R. Pike, "Another go at language design," Presented at the Stanford University Computer Systems Laboratory Colloquium OSCON, 2010. [Online]. Available: http://assets.en.oreilly.com/1/event/45/AnotherGoatLanguageDesignPresentation.pdf

[7] L. Ionkov, "Package for implementing 9p servers and clients in Go." [Online]. Available: http://code.google.com/p/go9p

[8] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: a checkpoint filesystem for parallel applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 21:1–21:12. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654081

[9] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS)," in *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, ser. CLADE '08. New York, NY, USA: ACM, 2008, pp. 15–24. [Online]. Available: http://doi.acm.org/10.1145/1383529.1383533

[10] Y. Tian, S. Klasky, H. Abbasi, J. Lofstead, and R. Grout, "Edo: Improving read performance for scientific applications through elastic data organization," in *Proceedings of the IEEE International Conference on Cluster Computing*, ser. Cluster '11. IEEE, 2011.

[11] R. Thakur, W. Gropp, and E. Lusk, "A case for using MPI's derived datatypes to improve I/O performance," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, ser. Supercomputing '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 1–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=509058.509059

[12] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole, Jr., "Semantic file systems," in *Proceedings of the thirteenth ACM symposium on Operating systems principles*, ser. SOSP '91. New York, NY, USA: ACM, 1991, pp. 16–25. [Online]. Available: http://doi.acm.org/10.1145/121132.121138

[13] V. Cate and T. Gross, "Combining the concepts of compression and caching for a two-level filesystem," in *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-IV. New York, NY, USA: ACM, 1991, pp. 200–211. [Online]. Available: http://doi.acm.org/10.1145/106972.106993

[14] R. Faulkner and R. Gomes, "The process file system and process model in unix system v," in *Proceedings of the USENIX Conference*, January 1991.

[15] R. Pike, D. Presotto, K. Thompson, and H. Trickey, "Plan 9 from Bell Labs," vol. 10, no. 3, pp. 2–11, Autumn 1990.

[16] J. P. Ahrens, J. Woodring, D. E. DeMarle, J. Patchett, and M. Maltrud, "Interactive remote large-scale data visualization via prioritized multi-resolution streaming," in *Proceedings of the 2009 Workshop on Ultrascale Visualization*, ser. UltraVis '09. New York, NY, USA: ACM, 2009, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/1838544.1838545