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, using 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 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. DRepl decouples the data producers and consumers and the data layouts they use from the way the data is stored on the storage system. DRepl has shown up to 2x for cumulative performance when data is accessed using optimized replicas.**

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

I. Introduction

The amount of data produced by scientific applications increases with the aggregate memory size of the high-performance supercomputers they run on. Future exascale systems will require hundreds of petabytes storage just to satisfy the need 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 allows flexibility that might be used in future storage architectures that implement burst-buffer [6] schemes. A burst-buffer is a type of hierarchical file system designed to reduce the number of checkpoints sent to the parallel file systems. Sitting between the node and the parallel file system, the burst-buffer is high-speed temporary storage for

978-1-4799-0218-7/13/$31

quick depositing of multiple checkpoints with only a fraction of them being moved to the parallel file system. Burst-buffer nodes can run the DRepl file server, providing transparent in-transit access to the data for the visualization and analytical applications in the appropriate format.

We implemented a prototype of DRepl and compared it with a baseline hand-optimized proof-of-concept system.

The results show improvements in the cumulative performance when data is stored in multiple replicas and the data is accessed from the replica that has its layout optimized for the particular access pattern. Our prototype still needs more work to get performance closer to the hand-optimized system.

The chief contribution of DRepl is to provide optimized access from various applications to the same dataset. DRepl decouples the data producers and consumers and the data layouts they use from the way the data is physically stored on the storage system. The rest of the paper is laid out as follows, immediately following this introduction is the Design section details the DRepl language and replication system. Next is section III which exposes how DRepl transforms data from the language definition to the various views and replicas. In section IV performance data is shown for two implementations of DRepl, one user-space and one kernel-space, against two other hand coded file servers. Finally the Related work, Conclusions and Future work sections finish the presented research.

II. Design

The goal of DRepl is to provide ways for replicating scientific datasets in semantically consistent way, so tools with different access patterns can use a custom replica that allows the fastest access. In order to do that, DRepl needs semantic knowledge of the data read or written by the application.

Some of the existing libraries allow developers to provide partial, or full description of the scientific dataset. For example, developers using MPI-IO [7] can specify the MPI types of the data accessed. HDF5 goes even further, allowing description of the whole data model, where the data elements have names and can be organized in groups. The drawback of using these libraries is that they require significant changes in the code base. Furthermore, describing dataset with their API is often quite verbose and not easily readable.

DRepl is a research project that will allow experimentation with different levels of replica consistency and synchronization. In order to achieve this, we need to decouple its behavior from the way the application accesses its data. DRepl also needs to be unobtrusive and yet easy to use by the application developers and researchers.

Instead of providing an API to define the dataset (or adopting one of the existing ones), we decided to create a declarative language that allows definition of dataset, views of the dataset, as well as, replicas that store the dataset on persistent storage. Using specific language allows expressive and easy to understand definition of datasets. In order to ensure familiarity, we chose syntax similar to the syntax of type and data declarations shared by many of the popular languages like C, C++, Java, etc.

We distinguish three major entities related to data and the way it is used and stored. *Dataset* is an abstract definition that describes the data types and data objects that are of interest of any application, regardless of whether it is a producer or consumer of the data. *View* is a subset of the dataset that defines the parts of the data that are of interest of particular application, or set of applications. If the data is accessed as file(s) from the file system, the view may also define the order the data objects are laid out in the files. *Replica* is a subset of the dataset that is persistently stored on a storage system. A replica is *full* if it contains all the data from the dataset, or *partial* if it contains only part of it. If stored on a file system, the replica also defines the layout of the data.

Using separate language allows us to decouple the dataset definition by the way various tools and applications see and access the data. The dataset defines the abstract data model which can span across data generated by multiple simulations and sources. Each application can have its own private view of the data, or subset of the data. The actual layout of the data as stored on the file system can be decoupled from some (or all) of the applications. Legacy applications can still read and write data in the format they use even though the data is stored in HDF5 or other standard scientific format.

The DRepl language allows definition of datasets, views and replicas. In order to allow legacy data access, DRepl acts as virtual file system that contains separate files for each of the defined views. The data from the dataset is backed to the specified replicas. The application reads from one of the view files on the virtual file system and DRepl reads one or more of the underlying replicas to satisfy the request. Writing to a view file is translated into one or multiple writes to the replica files. Figure 1 shows an example of a dataset produced by two simulations, Sim1 and Sim2 and accessed by two other applications, Viz1 and A1. The data is stored in three replicas, with the A, B, C, and D parts having two copies each. DRepl is managing the three replicas to provide the custom views the four applications have defined. As you can see DRepl has a lot of freedom in producing the views and can optimize for load balancing replicas, to prioritize the view of an application or to maximize data resilience, for example.

The rest of this section defines the syntax and the semantics of the DRepl language.

A. DRepl Language

The DRepl language includes definition of datasets, views and replicas. It allows definition of custom types based on a set of primitive types, as well as composite types as arrays and structs. It's syntax is loosely based on the type and variable definition in the Go language, which is similar to the type and variable definitions in C, C++ and Java. The language is designed to allow representation of native application datasets with complex views to allow visualization and analytics optimized access to data of interest. NetCDF [5], HDF5 [4] and the data formats from local large-scale HPC applications were investigated to ensure good representation of real datasets.

The content of a DRepl definition file can be divided into three main sections: dataset definition, view definitions, and replica definitions.

1) Dataset Section: The dataset section defines the types that are used in the dataset as well as the variables that make up the dataset. The dataset section is an abstract description of the dataset but it doesn't define the way the elements in multidimensional arrays are laid out (row-major, row-minor, z-order, etc.). These specifics are defined when views are described based on the dataset.

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

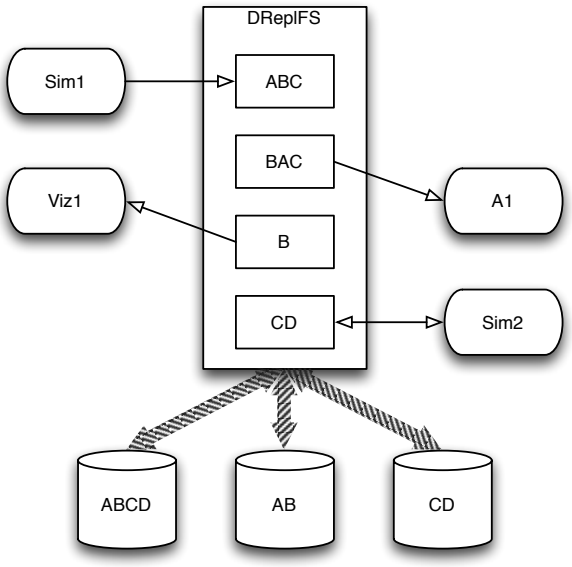

Figure 1: Example of DRepl being used by four applications and storing the data in three replicas

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

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

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
}
```

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
```

Named Types: The user can assign names to the defined custom types. The names allow usage of "shortcuts" when a type is often referenced.

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

2) Types:

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

Unlike most languages, DRepl allows types to be referenced before they are defined, so there is no need for forward declaration mechanisms.

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
```

Dataset: A dataset is the top-level construct in a dataset section. Dataset is a collection of types, variables and constants.

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

The example below defines a one-dimensional array whose elements have three 32-bit float values:

```
dataset {
    const N = 1000000

    type Point struct {
        a, b, c    float32
    }

    var data [N]Point
}
```

3) View Section: Views define subsets of the dataset. Data in a view can be accessed by read and write operations on the virtual file provided by the DRepl file server. If a view is stored as part of a replica, the view is *materialized*. Reading data from materialized views is fast, but there is performance penalty when the data is modified due to the need to update replicas.

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

The variables define in the view are based on dataset variables, providing full, or partial content of the dataset variable. Each view variable is of the same type as the variable it is based on, or compatible sub-type.

View Substruct:

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

Example of usage of a view substruct:

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

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

View Slice: The view slice contains only some of the elements of an 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 which elements from the array are included in 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]
```

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 `d` that contains only the `b` fields of every 3^{rd} element 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]
```

The temporary variables used for defining slices don't have to be used for the same dimension across the slice definition. For example, in order to reverse the column and row order in a two dimensional array, one can use:

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

View Declaration: The view declaration defines a view. Each view has a name that corresponds to the name of the virtual file that allows access to the data in the layout defined in the view definition body. Additionally, a number of flags can be specified that control the element order in arrays as well as if the view can be used to update the dataset. The data in the dataset can't be updated via read-only views.

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

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.

In future, the view declaration will also allow flags that allow control over the endianness of the data in the view.

5) Replica Section: Replica sections define how the data from the views is stored on the underlying storage system. Replica is a sequence of one or more views. Each replica has a name that corresponds to a filename on the file system. The replicas can be *complete* or *incomplete*. A complete replica contains all the data from the dataset. If the views included in the replica contain only part of the dataset, the remaining data is appended at the end of the replica file to ensure a complete replica. The layout of the additional data is not defined and may change with the implementation.

Each materialized view has to belong to at least one replica. The views that are not part of any replica are non-

materialized and when they are read, the data is transformed on-the-fly from one or more of the materialized views. On writes to non-materialized views DRepl updates all relevant replicas.

```
ReplicaDecl = "replica" identifier
   CompleteFlag "{"
   { ReplicaView} "}"
ReplicaDecl = "replica" FileName
CompleteFlag = "complete" | <nothing>
ReplicaView = "view" identifier
```

III. Implementation

The DRepl implementation consists of three main modules: DRepl language parser, replication engine and a file server (dreplfs). The parser receives description of the dataset, views and replicas, and produces internal representation of the view and replica layouts. The representation is passed to the replication engine, which uses it to perform translation to and from the view representations to the replicas' format.

We have two implementations – a user-space and a user/kernel space combined solution. The first implementation (Fig. 2) is written in the Go language and uses the 9P file protocol to create a file server in user-space. The second implementation (Fig. 3) runs the parser in user-space, but the replication engine and the file system are written as Linux kernel file systems and run in the kernel. The user-space implementation can run on more operating systems, but has performance disadvantages.

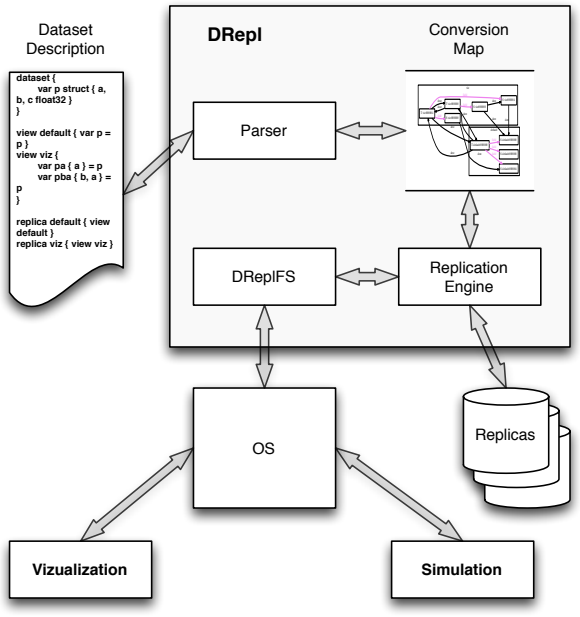

Figure 2: DRepl user-space implementation

Currently the prototype doesn't support dynamic addition and removal of views and replicas. This feature is easy to implement and will be added in near future.

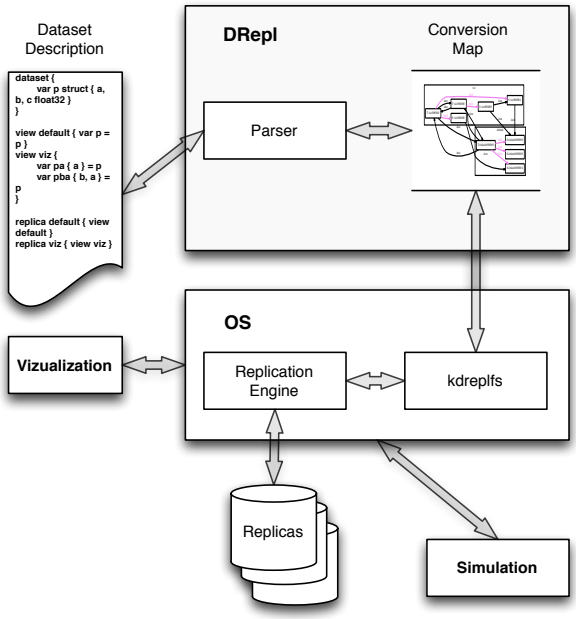

Figure 3: DRepl hybrid implementation

A. Transformation Rules

The replication engine uses the internal representation of the collection of views and replicas to implement the core of the DRepl functionality. The internal representation for each view consists of a list of byte regions (blocks) as well as transformation rules on how to transform the data to related data regions in the other views. In case of unmaterialized view, the transformation rules describe which regions from other materialized views are used to retrieve the data for the region.

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 describing how to transform data from one variable to the other. If a view is marked as read-only, the maps describing how to transform its variables to others are not created.

1) Block Types: The layout of the views is defined as a list of *Blocks*. All blocks have an offset from the beginning of the view, size, source block (if it belongs to unmaterialized view), or a list of destination blocks.

There are three types of blocks required to describe a DRepl view. The simplest block, *SBlock*, defines region that is always read or replicated as a whole entity. All primitive types are described as SBlocks, but in some cases the optimizer can coalesce multiple adjacent SBlocks into a single, bigger SBlock. A *TBlock* is a collection of other blocks. It corresponds to a `struct` composite type in the DRepl language. Finally, *ABlock* defines an (multidimensional) array of identical blocks.

2) 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 Block
    }
```

For example, Figure 4 shows the conversion map for a dataset description:

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

view default {
   var p = p
}

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

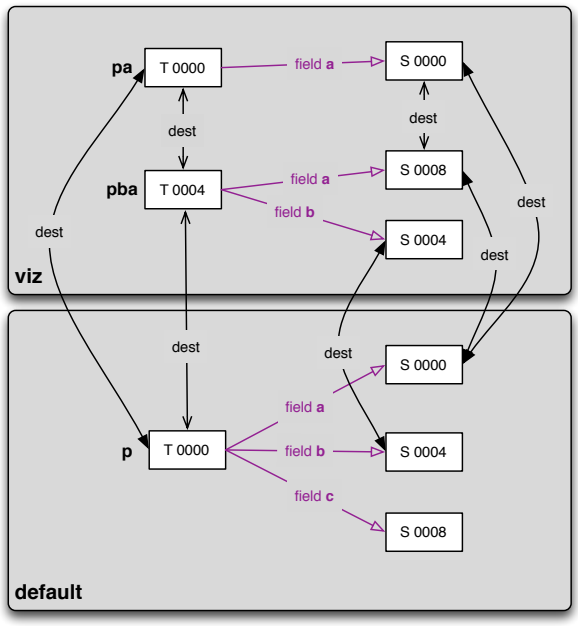

Figure 4: Example of a conversion table of two views

In this figure you can see that the *default* view is in the lower box, with a *TBlock* that holds the variables a,b,c. The top box is the *viz* view. Variable *a* in the *default* view gets two references from the viz view, while element *b* gets one reference and element *c* gets no references from the *viz* view.

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

3) Updating Data: The dataset can be updated via a `write` operation to one of the view files. The `write` operations receive as arguments the file offset, the number of bytes written and the data. The replication engine finds all Blocks that belong to the specified range and applies the conversion rules specified in the blocks.

4) 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 from the conversion map and the data is copied from one or more of the materialized views.

B. Replica Layout

Each replica is a separate file, or directory, with name specified when the replica is declared. 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 are 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.

1) 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.

2) 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.

3) 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.

4) 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 multiple files. A variable-sized type is laid out using the same rules as any other type. The data defined after the variable-sized 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.

Currently the DRepl prototype doesn't implement variable-size data types.

IV. Results

While evaluating the user-space prototype, we wanted to make sure we isolate implementation choices (like programming language, file protocol, etc.) from the performance inherent to our design choices.

We created a simple dataset and tested the bandwidth when reading and writing from it. We wrote a description of the dataset and three views in the DRepl language and measured the performance using DReplfs to access the data. Additionally, we created a hand-optimized file system written in the same programming language and file protocol that implements the same dataset. Comparing DReplfs with the optimized implementation allows the evaluation of the performance penalty due to the complexity of the DRepl language and any inefficiencies in the replication engine implementation. Lastly, we compared the performance to using a proxy file system, implemented again using the same programming language and file protocol, but directly translating the access to the files it serves to the underlying file system. This comparison allows us to isolate the performance of the chosen programming language and file protocol from the overhead added from maintaining views and replicas in DRepl.

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. We ran our experiments with 168 million points.

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

Point    data[N]
```

B. Views

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

1) Array of Structures (aos):

```
Point    data[N]
```

2) Structure of Arrays (SOA): 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 (Partial): In most cases the visualization requires only some of the values. We chose a visualization view that contains only an array of the b values.

```
float    b[N]
```

C. File servers that provide multiple data views

When the file servers are mounted, they provide access to three views of the data:

- SOA — Data in legacy format (structure of arrays). 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.
- aos — Data in natural format (array of structures). Contains N elements, each 12 bytes long with the values of a, b and c for that element.
- b — Data in the visualization (partial) format. Contains N elements, each 4 bytes long with values only of b.

We ran sets of experiments varying the following parameters:

- Replica number — We tried three different combinations of replicas: one replica for each view, two replicas containing SOA and b views (aos view is unmaterialized), and one replica containing the SOA view (aos and b views unmaterialized).
- 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.

Read and write operations access all data in the view sequentially.

The experiments were performed on 4 socket, 4 core servers (total 16 cores) with 32GB of RAM. We used separate SSD disks for each of the replica files. The OS buffer cache was cleaned between every experiments. We also performed the tests on rotational disks and the results were very similar.

D. User-space file servers

We compared performance of three user-space file servers that provide access to the data in the three different views:

- dreplfs — Our prototype that receives a dataset, view and replica definitions in the DRepl language, converts it to the internally used conversion tables, provides access to the views as virtual files, storing the data in the specified replica files. DReplfs is implemented in Go and uses the 9P2000 file protocol (go9p [8] library).
- dsfs — A hand-optimized file server that provides access to the three views mentioned as virtual files and allows storage of the data to one, two or three replicas, each containing one of the views. Dsfs

is implemented in Go and uses the 9P2000 file protocol (go9p library).

ufs
: A "proxy" file server that provides access to the files on the file system over the 9P2000 protocol. Ufs is implemented in go and uses the 9P2000 file protocol (go9p library).

DReplfs and dsfs have an option that allows the transformations to the replicas to be performed asynchronously. Ufs doesn't allow replication.

Converting data from one view to another can be very inefficient. For example, if a program writes to the first 400 bytes of file `SOA`, updating the first 100 values of a, the operation needs to be converted to 100 writes to file `aos`, 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. All three implementations use `mmap` to read and write to the files, instead of the standard POSIX I/O calls.

1) Raw Performance: Figure 5 shows reading performance. The read performance when reading from materialized replica is roughly equal for all tested file servers. Reading from unmaterialized replica is about 15 times slower for dreplfs and three times slower for dsfs. The conversion from materialized to unmaterialized view needs to be done before the read operation can return, so the asynchronous mode doesn't provide any performance increases.

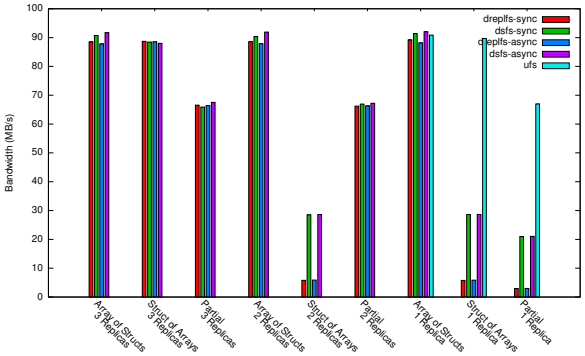

Figure 5: Read performance

Figure 6 shows writing performance. The hand-optimized file server is about three times faster than dreplfs. There is big advantage in running the replication asynchronously. The penalty of maintaining more than one replicas is pronounced more in dreplfs than the hand-optimized file server.

2) Combined Performance: In order to evaluate the performance advantage of creating multiple replicas and reading

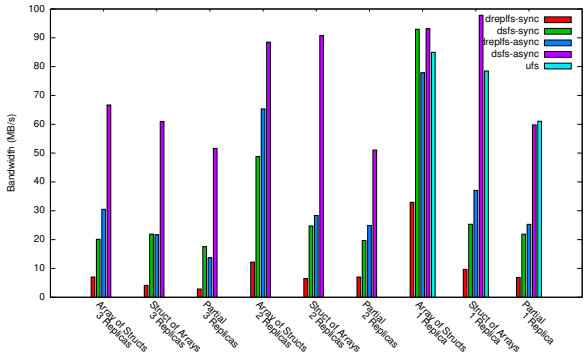

Figure 6: Write performance

from the most optimal, we emulate two separate cases:

single replica
: The simulation application writing the data to a single replica (containing the AOS view) and the visualization application reading the partial view from it;

two replicas
: Replicas containing the AOS and the b views are created, the simulation application reads the data from the AOS replica, the visualization application is reading from the partial replica.

Figure 7 shows cumulative bandwidth for accessing the dataset from one or two replicas. Even though the write performance is decreased when maintaining two replicas, the cumulative performance is improved by the fact that the read operations use the optimized replica.

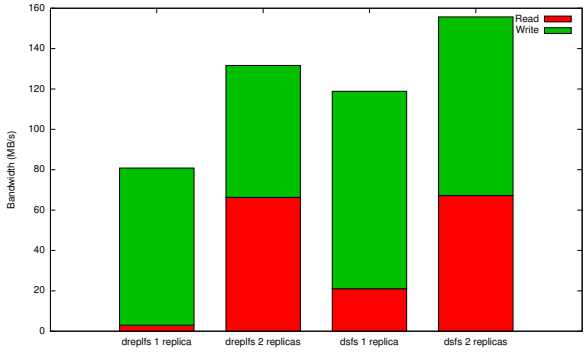

Figure 7: Cumulative bandwidth for accessing one and two replicas

E. Kernel-space file server

We compared performance of the our kernel implementation of the DRepl file system (kdreplfs) with conventional POSIX file system.

1) Raw Performance: Figure 8 shows reading performance. Our tests have shown that there is no difference between asynchronous and synchronous mode (in both cases

the conversion needs to finish before we return the data), so the figure doesn't have the asynchronous results. Reading from materialized view is about nine times faster than reading from unmaterialized view. Further optimizations of the replication engine should might decrease the difference between them.

The POSIX read performance is shown only in comparison to the 3 replicas configurations, because in these configurations and the POSIX case the data is read directly from the filesystem and no unmaterialized view processing is required.

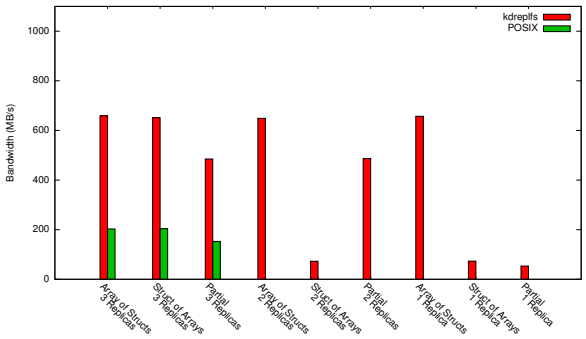

Figure 8: Kdreplfs read performance

Figure 9 shows writing performance. The asynchronous mode is usually faster than the synchronous one, although there are cases when that's not the case. In asynchronous mode, the kdreplfs needs to allocate additional memory in the kernel and copy the data buffer so the replication engine can continue to work even after the write call returns to the user process. The memory allocation adds some overhead. Our implementation makes sure that at least one replica has the data committed before the execution is returned to the user space. That may slow down the asynchronous mode even further.

The POSIX write performance is shown only in comparison to the one replica configuration, because in these configurations and the POSIX case the data is written directly to the file system and no replication of the data is required.

2) Combined Performance: Figure 10 shows cumulative bandwidth for accessing the dataset from one or two replicas. Even though the write performance is decreased when maintaining two replicas, the cumulative performance is improved by the fact that the read operations use the optimized replica. The graph also shows the POSIX performance when the data is written in the SOA format and then read in chunks and converted to the AOS format in memory (the chunk size is 524288 points).

V. Related Work

There has been quite a body of work using middle-ware to optimize reading or writing of application data. PLFS [9]

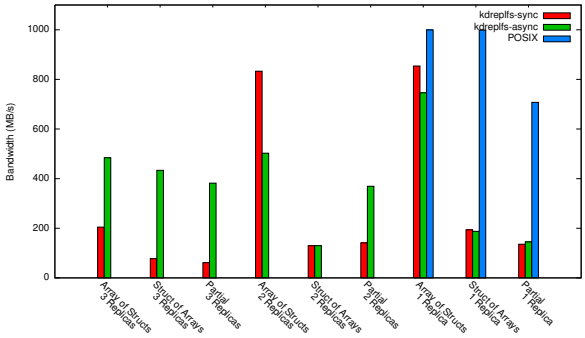

Figure 9: Kdreplfs write performance

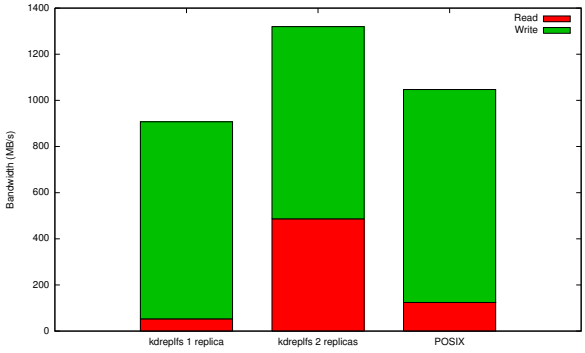

Figure 10: Cumulative bandwidth for accessing one and two replicas

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 [10] 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 [11] has reordered data using space filling curves for faster access. MPI IO 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.

ArrayQL [17] provides language designed for similar purposes as DRepl, although it is based on SQL and is probably more unfamiliar to the developers in the HPC community.

There is work done on creating adaptive layout strategies based on the data access patterns [18]. These techniques improve the I/O performance but don't help if the data needs to be read in-transit by visualization or analytical tools.

VI. Conclusion

DRepl provides a novel method for optimized access to application datasets that are read and written with multiple contrasting patterns. DRepl decouples the data producers from the data consumers and additionally from the physical data layouts used on the storage system. Initial investigations showed increased performance in both read and writes on various physical media. A prototype file system was implemented in both user and kernel-space 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. Also in a realistic work flow where multiple readers and writers are access the data, DRepl will improve the cumulative performance due to the performance gained from the read operations accessing the optimized replica.

VII. Future Work

Development is continuing with DRepl, our next step is to test with real applications and workflows. When full application support is verified we are planning to work on optimizing it for highly parallel loads. We need to do more work on the replication engine performance so we can get it closer to the one we achieve with the hand-optimized file server.

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.

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.

We will also modify dreplfs so the users can create and destroy views and replicas on the fly without restarting it. That would allow faster and more flexible operation when the analysis and visualization are not used very often.

VIII. 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).

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] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*. IEEE, 2012, pp. 1–11.

[7] 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

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

[9] 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

[10] 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

[11] 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.

[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

[17] L. K.-T., M. D., B. J. K. M., Z. Y., and S. M., "Arrayql syntax draft 4," Tech. Rep., Sep. 2012. [Online]. Available: http://www.xldb.org/wp-content/uploads/2012/09/ArrayQL-Draft-4.pdf

[18] H. Song, H. Jin, J. He, X.-H. Sun, and R. Thakur, "A server-level adaptive data layout strategy for parallel file systems," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, may 2012, pp. 2095 –2103.