Flexible navigation through a multi-dimensional parameter space using Berkeley DB snapshots

Dimokritos Stamatakis
Brandeis University
415 South St
02453, Waltham, MA
United States
dimos@cs.brandeis.edu

Werner Benger
AHM Software GmbH
Technikerstrasse 21a
A-6020 Innsbruck,
Austria
w.benger@ahm.co.at

Liuba Shrira
Brandeis University
415 South St
02453, Waltham, MA
United States
liuba@cs.brandeis.edu

ABSTRACT

The concept of a visualization pipeline is central to many applications providing scientific visualization. In practical usage scenarios, when the pipelines fuse multiple datasets and combine various visualization methods they can easily evolve into complex visualization networks directing data flow. Creating and managing complex visualization networks, especially when data itself is time-dependent and requires time-dependent adjustment of multiple visualization parameters, is a tedious manual task with potential for improvement. Here we discuss the benefits of using Berkeley Database (BDB) snapshots to make it easier to create and manage visualization networks for time-dependent data. The idea is to represent visualization network states as BDB snapshots accessed via the widely used Hierarchical Data Format (HDF5), and exploit the snapshot indexing system to flexibly navigate through the high-dimensional space of visualization parameters. This enables us to support useful visualization system features, such as dynamic interpolation of visualization parameters between time points and flexible adjustments of camera parameters per time point. The former allows fast continuous navigation of the parameter space to increase animation frame rate and the latter supports multi-viewpoint renderings when generating Virtual Reality panorama movies. The paper describes how the snapshot approach and the new features can be conveniently integrated into modern visualization systems, such as the Visualization Shell (Vish), and presents an evaluation study indicating that the performance penalty of this convenience compared to maintaining visualization networks in HDF5 files is negligible.

Keywords

Scientific Visualization, Big Data, HDF5, Database Snapshots, Parameter space exploration

1 INTRODUCTION

Creating high-quality scientific visualizations of large time-dependent datasets can be time-consuming for the scientists. In data-flow based visualization systems this process can involve step by step creation of manually tuned instructions in the form of visualization pipelines (networks) that describe the flow of data from sources to sinks and are the basis elements of more complex visualization networks [1].

For large observational or computational data sets, common in todays systems, the list of visualization instructions can be long and different data points may need different structures, e.g. direct different number of cameras representing different validation perspectives, like in Virtual Reality environments.

Current visualization systems do not provide satisfactory support for managing long varying visualization instruction lists. For example, visualization systems store lists of instructions in spreadsheets, but the rigid structure makes it hard to accommodate instructions with varying number of parts [2]. Instruction lists are represented by specifying transformations that map each instruction into its successor instructions [3], which provides instruction derivation provenance but makes it hard to flexibly navigate through the visualization starting from arbitrary data point of interest.

To better support the large time-dependent datasets, visualization systems need to provide features that make instruction management more flexible and easier to manage. For example, it would be beneficial if the visualization systems could support visualization parameter interpolation between different time steps (similar to dead reckoning in video games or any non-linear video editing software such as Adobe

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Premiere [4]) to reduce creation effort by avoiding storing redundant instructions. Furthermore, dynamic parameter interpolation would also allow to achieve higher animation rates or provide coarser views as needed during the visualization. Many current systems however do not support interpolation.

This paper discusses the benefits of using database snapshots to improve visualization system support for time-dependent data. Specifically, we describe how we extended the Visualization Shell (Vish) [5] using Retro [6], a snapshot system for the Berkeley Data Base (BDB) [7] to provide dynamic interpolation and flexible navigation for lists of visualization networks for time-dependent data in HDF5 format [8]. Our system represents lists of network states as a list of network versions stored as successive BDB snapshots. This representation can easily accommodate instructions with varying structure. We can take advantage of the snapshot indexing system to navigate the network versions and can support flexible traversals of the high-dimensional visualization parameter space stored in snapshots from arbitrary data points. The snapshot representation also makes it easy to support dynamic interpolation of parameters between the time points.

Our visualization system uses the portable HDF5 format to represent both the time-dependent data and the visualization networks. The time-dependent data is stored via HDF5 library in a file system well-suited for the large files. A natural question arises, why not store visualization networks in HDF5 files as well. To this effect, we have also experimented with a system that stores visualization networks directly in HDF5 files, instead of snapshots, implementing the flexible navigation manually, instead of relying on snapshot indexing. Using the snapshot system however was considerably simpler. Moreover, an important benefit of using database snapshot system is providing a reliable transactional storage for visualization networks that represent a substantial scientist time investment. By using the transactional snapshots we benefit from an automatic recovery of the visualization network structures after an early or involuntary program termination, something that would need to be achieved with complex ad-hoc recovery methods in a system that stores the networks in HDF5 files.

Our system therefore uses two different backend storage systems, HDF5 files for time-dependent data, and BDB snapshot system for the visualization instructions. Such approach, combining multiple storage systems each optimal for intended data use is becoming increasingly common in the new generation of big data systems due to the realization that no single storage system is optimal for the rich set of data comprising todays big data systems [9]. Our implementation takes advantage of the recently introduced HDF5 VOL software layer [8] that allows applications to access HDF5 data objects in different storage backends. For our system, we have implemented a new VOL that provides access to BDB snapshots, allowing the Vish system to seamlessly manipulate data and visualization metadata through the same HDF5 API.

A visualization system must perform well. We have conducted a study that evaluates the performance of our snapshot based Vish system and its new interpolation and navigation features, using micro benchmarks. We have also compared our system performance to the design that stores both data and visualization instructions in the HDF5 files. The measurement results indicate that our snapshot based system performs well, and the performance penalty we pay for the simplicity of implementation and reliability compared to storing instructions in HDF5 files is acceptable.

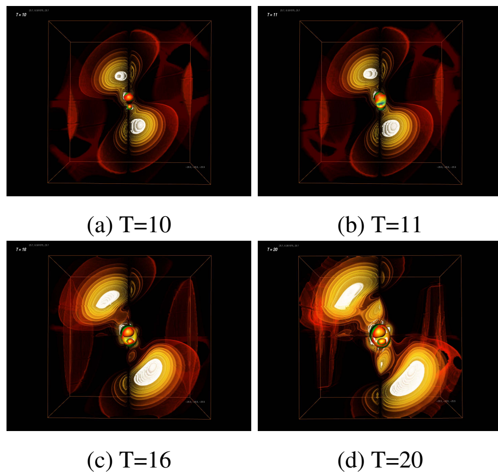
Our main contributions in this paper are the following:

- A simple database snapshot-based approach for managing visualization metadata for time-dependent HDF5 datasets
- Design and implementation of the BDB VOL plugin for HDF5
- Design and implementation of the Interpolation component in Vish visualization system
- Performance evaluation study of our snapshot-based approach and the interpolation feature

The rest of the paper is organized as follows. Section 2 describes a concrete scenario that motivated our work. Section 3 discusses related work, Section 4.1 explains the software structure of Vish system, the context of our work, Section 4.2 describes the interpolation feature, Section 4.3 highlights the salient points of our implementation, Section 5 presents the performance study and Section 6 our conclusions.

2 MOTIVATION

Solving Einstein's equations on supercomputers [10] is a grand challenge involving many institutions and generations of scientists. These efforts have culminated by the recently announced detection of gravitational waves. Numerical simulations of black hole collisions as the strongest sources are essential for the proper analysis of the detected signals. A milestone in numerical relativity was the first fully three-dimensional simulation of a grazing collision of two black holes [11]. Fig. 1 is showing four time steps from this simulation. Shown is the real part of the complex Newmann-Penrose pseudoscalar Ψ_4, an indicator of the outgoing gravitational radiation field, as elaborated in more detail in [12]. Visualizations like those are not only esthetically pleasing [13], but also important for scientific development to assess the quality and features of



(a) T=10 (b) T=11

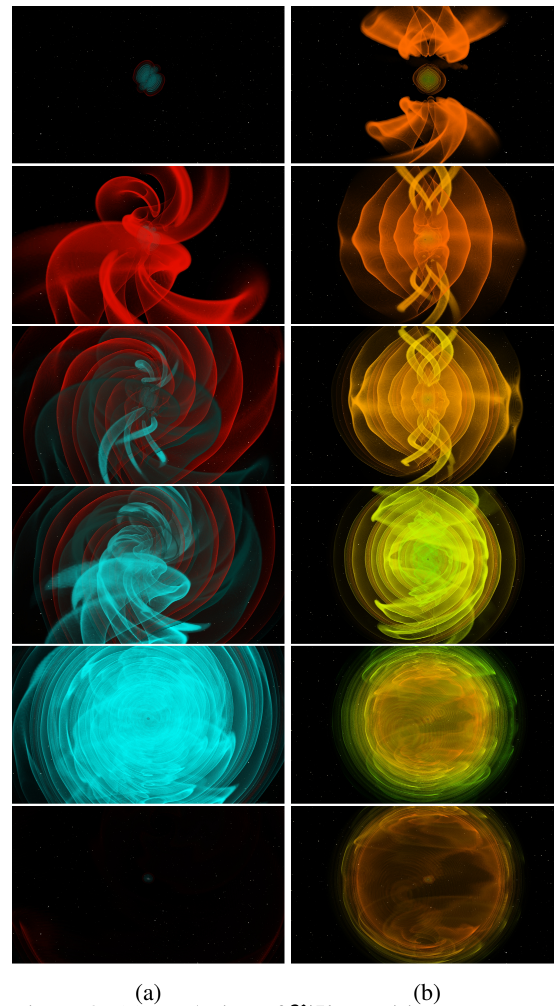
(c) T=16 (d) T=20

Figure 1: Burst of gravitational waves emerging from colliding black holes

big data. The first confirmed detection of gravitational waves [14], known as GW1509014, was simulated at the Max-Planck Institute for Gravitational Physics according to the observed collision parameters. This simulation produced a dataset of 400GB of binary data of much higher detail and precision than the 1999 dataset, particularly also covering a significantly longer time range lasting several orbits. Due to the nature of these astrophysically violent events the dynamic range of the data is huge, in space and time. While the 1999 dataset could be covered with parameters of constant range, trying the same approach on the 2016 dataset does not yield pleasing results, as shown in Fig. 2a: Initially the radiation is so weak that it is hardly visible at all, but its strength increases rapidly to emerge like a "flash" during a very small amount of time, leaving only residual radiation of a "wobbling" rotating black hole. This residual radiation fades away quickly at low intensity to ultimately form a non-radiating Kerr black hole.

The situation is similar to high dynamic range rendering: The output medium (images with 8 bits of intensity for each color) just cannot cover the entire range of the input data (orders of magnitude). In this case of evolving data a more suitable range can be found at each time step. Determining this range computationally is difficult though because of the wave-like nature of this astrophysical process leading to visually disturbing oscillations. This leaves manual adjustment of the data mapping range for color-coding as the only option, similar to controlling animations according to a movie director's intention via any non-linear video editing software. With the ability to fine-tune any parameter of a visualization network over time, we can thus extract the maximum structural information out of time-dependent data at the cost of less quantitative assessment abilities.

Another approach to cover high dynamic range is a global transformation such as computing the logarithm, as demonstrated in Fig. 2b. This approach worked in



(a) (b)

Figure 2: (a) Evolution of $\Re\Psi_4$, positive components in red, negative in cyan. (b) Evolution of $\log|\Re\Psi_4|$, covering a wider range at the cost of signature and detail.

this specific situation to provide an overview of the dataset without manual interaction at the loss of signature information – we can only see the absolute value of $\Re\Psi_4$, but not if its positive or negative, which roughly corresponds to stretching or compression of spacetime at the very location.

So while specific workarounds were able to yield visually pleasing results for the particular application scenario, a more systematic approach for arbitrary fine-tuning similar to professional video editing systems is desirable.

3 RELATED WORK

Visualization data management systems support versions of visualization graphs for different reasons and correspondingly use different approaches. The use of a visualization pipeline is pretty much a core standard among software frameworks for scientific visualization. Paraview[15] and Visit[16] are based on the Visualization Toolkit VTK[17]; OpenDX[18] and

Amira/Avizo[19] are other widely used frameworks. Vish system [5] is an independent visualization system with a much smaller user community, but comes with the most systematic approach to deal with visualization concepts while being academically open-source and a comparable small codebase; it is therefore the framework of our choice for our work, as described in 4.1.

VisTrails [20] is more of a management system to visualization systems than a visualizion framework itself. It uses executable XML specifications to generate sequences of visualization network graphs (Vistrail specifications) that can vary in structure and provides efficient runtime that allows efficient incremental visualizations along the time line. This method supports regular structure and parameter variations between networks at different time points but does not support flexible time and data dependent variations supported by our approach. VisTrails stores graphs using SpreadSheets so when the schema of the graph evolves, the history needs to be reformatted. Our snapshot approach supports arbitrary evolving network schemes without reformatting. The VisTrails data management system [3] represents specifications corresponding to successive data points by storing operations that transform one network into another to support derivation provenance. The representation requires to apply the full operation history to visualize a given data point making it inconvenient to traverse the version graph from arbitrary points. In contrast, our snapshot approach allows to navigate visualizations from arbitrary time points and can support provenance programmatically by storing transformations as additional attributes in snapshots.

ModelDB [21] manages a branching version history of machine learning models and visualizations aimed at exploration of alternative parameter configurations and like VisTrails specifies transitions using operations to support provenance. Our snapshot based approach, specialized for time-dependent data, currently only supports linear version histories.

Polystore [9] is a new generation scientific data management system that uses multiple storage backends to optimally accommodate data of different types, and implements an integrating layer to provide a unifying access to the different data parts. We adopt a similar approach by storing time-dependent data and visualization metadata in different storage backends, and take advantage of the HDF5 VOL infrastructure to add the snapshot system backend by implementing a new VOL.

4 SYSTEM DESIGN

This section describes our approach to managing visualization instructions for time-dependent data, and highlights the salient points of our implementation. We start by summarizing our requirements motivated by the use case of gravitational wave data visualization. To support visualization of time-dependent data a system needs to support:

- incremental creation of time-dependent visualization instructions, allowing scientists to conveniently adjust parameters for different time points, and exploit parameter interpolation to save effort when parameter changes can be computed.
- varying visualization instruction structure (e.g. extra cameras, or light sources) to offer customized views for different time points.
- adjustable frame rate (higher to lower) during visualizations using dynamic parameter interpolation.
- flexible navigation through the visualization parameter space from arbitrary time points.
- reliable persistent storage for instruction lists protecting scientist time investment in the presence of system crashes.

We describe below how our system design satisfies these requirements. We start by briefly explaining the features of Vish visualization environment, the context of our work. We then explain how we use a snapshot system to create and navigate instruction lists and present the design of the parameter interpolation feature. We then describe how we integrated the snapshot system backend into Vish.

A general view of the software layers in our system can be seen in Figure 3, where we show the Vish Visualization system (described in 4.1) and how its several components use the HDF5 library. First, we have the existing visualization module which is responsible for visualizing datasets aimed by networks. Then, we developed the parameter interpolation module, described in Sec. 4.2, and finally the dataset and network accessing modules allowing access to HDF5 data in native HDF5 format and BDB VOL format, respectively.

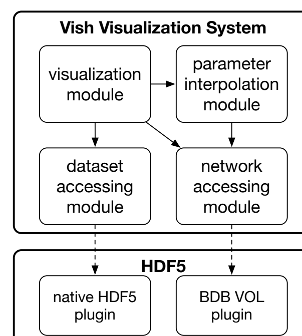
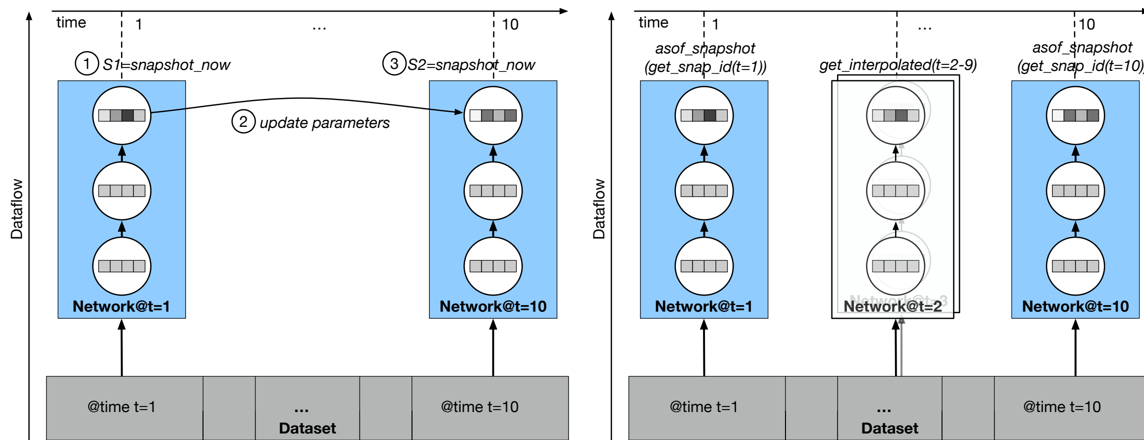


Figure 3: The software layering of our system

4.1 Vish

The Vish Visualization Shell [5] is a software environment specifically designed to "make everything a plugin". It manages all its plugins functionally through



(a) Create a sequence of visualization pipelines for time-dependent data only for the desired time steps

(b) Display time-dependent data using the sequence

Figure 4: The process of dynamic parameter interpolation

a small kernel of abstract object interfaces which are used to implement graphics, user interface, I/O, or object relationships itself. This approach allows for modular well-encapsulated components that can be implemented independently of each other. The concept of a visualization pipeline and the resulting graphs for practical applications are built into this minimalistic object management framework. It can then be managed via a runtime-loaded graphical user interface, for instance via visual programming, or to interface any language that supports scripting.

As a reference implementation, Vish comes with its own, minimalistic scripting language that allows to store and load a visualization's network state, while remaining human-readable and human-editable. It is however not optimized for performance since parsing text inherently comes with a performance overhead. Using the Vish plugin architecture, it is straightforward to implement an alternative way of loading and storing a visualization network state using another format such as HDF5, which then avoids a parsing overhead due to its self-descriptive binary nature. This is the format we use in our work for storing visualization networks.

Even more, a visualization network's state can be managed while loaded by some Vish object itself. A graphical user interface is nothing else than a plugin providing a Vish object with such management functionality. Via the Vish kernel API, objects, their parameters and connections are exposed in an abstract way to allow generic interaction. We can use this very functionality to also produce new network states from existing ones, for instance via interpolation as explained below.

Time is supported in various Vish modules as a continuous floating-point quantity, leaving the notion of "time steps" to be implemented locally to each data set. This way also non-equidistant time steps, such as produced by the CFL condition in numerical simulations [22], are taken care of. Consequently, indexing data given at discrete time steps by a continuous time value requires data interpolation. When producing an animation with a given frame rate, the continuous time is subsampled by discrete steps again, however, independent of the time steps of the original data. We can hereby smoothly fuse data from different sources with different time discretizations. The Vish user interface contains plugins to sample the continuous time parameter space and produce discrete time steps for animation. We use this feature to navigate time points for network states and data.

4.2 Parameter Interpolation

Visualization of time-dependent datasets might require adjusting parameters other than time as visualization proceeds along successive time points in the data set. In this case, we want to keep track of different states of the network graph corresponding to different time points. In order to easily manage these states, we decided to use a Database snapshot system that provides a simple interface for creating and accessing data versions as database snapshots. We store the visualization networks in BDB [7], a key-value database, and manage network states using Retro [6], a snapshot system for BDB. Retro supports an easy implementation of the simple network state management workflow where a scientist visualizes the current data point using the current network state, adjusts the network state parameters as needed, stores the new network state by taking a snapshot of the adjusted state, and proceeds to the next time point. However, creating a separate network state manually for each key frame is time consuming and maybe unnecessary if the parameter changes can be interpolated automatically. Thus, we decided to add an option for dynamic parameter interpolation, which will calculate the missing intermediate parameter values and remove the burden from the scientists. This

can be easily implemented with our snapshot approach, since Retro allows programs to access snapshots and compute with data stored in snapshots the same way as with data stored in the database [6]. Thus our system can create only needed states and during visualization load the two corresponding consecutive snapshots and then calculate the desired network parameters dynamically for the individual intermediate time points.

Figure 4 illustrates the interpolation workflow. In Figure 4a we show the process of creating snapshots for a time-dependent dataset, representing an animation. We see the states of the visualization network graph at each time point, representing directions for how to visualize at that point. Each node in the graph represents a visualization object, which disseminates data from the dataset (source) to the sink. Within each object we see the object's parameters, with different color representing different parameter value. Arrows point to the direction of the data flow. Initially, we have the visualization network at time point $t = 1$ which points to the dataset at time point $t = 1$, and then we take a snapshot so that to store the network state at that point using the snapshot system API operation `snapshot_now`. This operation takes a snapshot of the current state and returns a snapshot identifier (`S1` in this case). We enhanced the implementation to store a persistent mapping from time point to snapshot identifier (`get_snap_id(t=x)`), so that to correlate each snapshot with a time step. Then, moving at time point $t = 10$, we update some parameters of the very first node in the graph (different color boxes) and when visualization looks good, we call the `snapshot_now` again for time $t = 10$, which returns snapshot identifier `S2`.

Simply displaying the network as of $t = 1$ for time points $t = 2$ to $t = 9$ might cause the visualization to change rapidly when reaching time point $t = 10$ which is undesirable for animations. Thus, we will use interpolation to smooth it. Figure 4b shows the process of loading the states of the visualization network graph from snapshots and displaying, resulting in playing the animation. For each time point, the visualization system has to check the time-snapshot mapping whether there is a corresponding snapshot and if not, interpolate parameters after loading two consecutive snapshots. At time $t = 1$, the system calls the Retro operation `asof_snapshot(get_snap_id(t=1))`, where `get_snap_id(t=1)` returns `S1` and `asof_snapshot(S1)` loads the snapshot with identifier `S1`. Thus, we load the snapshot corresponding to time point $t = 1$ providing the visualization network as it was at the time we took the snapshot.

Since time points $t = 2$ to $t = 9$ do not have corresponding snapshots, they are remapped to time $t = 10$ allowing us to load snapshot `S2`. Once we have parameter values for $t = 1$ and $t = 10$, we can calculate values for $t = 2 - 9$ using linear interpolation such that for each parameter A given at discrete time steps t_0 and t_1 we can compute its value as a smooth function

$$A(t) = A(t_0)(1-\tau) + A(t_1)\tau \quad , \qquad (1)$$

where τ is the relative time, i.e. $\tau = (t - t_0)/(t_1 - t_0)$ with $t_0 \leq t \leq t_1$. Higher order interpolations such as used on cubic splines [23] are possible as well, but require more snapshots to be accessed and evaluated, and a more complex time-to-snapshot mapping. We do the same for all intermediate time steps that we don't have a corresponding snapshot.

Of course, interpolation can also be used at creation time to create one snapshot per time point. This approach simplifies time to snapshot indexing but potentially stores a much larger number of snapshots and also does not allow flexible coarser or finer grained interpolations. The only case it could be beneficial is if the interpolation is effortsome, so storing snapshots for later playback would be faster. Thus, we believe the dynamic interpolation approach is preferable. Our performance evaluation considers both approaches in Section 5.

4.3 Berkeley DB VOL plugin

The current version of the Vish visualization system supports HDF5 format for both computational or observational data objects and visualization network objects, storing both data and networks in the HDF5 file system, and accessing them via the HDF5 library. This section describes how we extended the HDF5 library in the visualization system to support managing versioned visualization network objects in a snapshot system.

Since the native HDF5 file system does not support versioning, we had to provide versioning ourselves. We had two goals for our design. We wanted to provide versioning in a light way manner without modifications in the HDF5 codebase, and we wanted to allow the visualization system to continue use the HDF5 API for data and networks. Our design builds on a recent feature in HDF5 called Virtual Object Layer (VOL) plugins that allows the implementation of custom storage back-ends. A VOL plugin is a seamlessly connected component of the HDF5 library that is responsible for storing and accessing HDF5 data containers in a particular storage back-end.

We support versioning of visualization networks using a VOL plugin that stores network states represented as HDF5 objects in a backend that supports versioning. Our versioning back-end storage is Retro system that provides snapshots for Berkeley DB [6]. Retro allows easy versioned network state creation, simple version indexing and supports easy computation with versioned states e.g. to support dynamic interpolation feature explained earlier. Importantly, Retro provides reliable crash consistent storage for versions thus satisfying

```
Current-state queries are unchanged by Retro
    results ← { gid = H5Gopen2(fid, node_name, ...);
                attr = H5Aopen(gid, attr_name, ...);
                H5Aread(attr, mid, buf);
                ... }

Applications may declare snapshots at any time and
get back a snapshot identifier
    S ← snapshot_now

As of queries are delimited with the snapshot identifier
    results ← asof_snapshot S { gid = H5Gopen2(fid, node_name, ...);
                                attr = H5Aopen(gid, attr_name, ...);
                                H5Aread(attr, mid, buf);
                                ... }
```

Figure 5: Example of using the Retro API to retrieve attributes from a visualization network

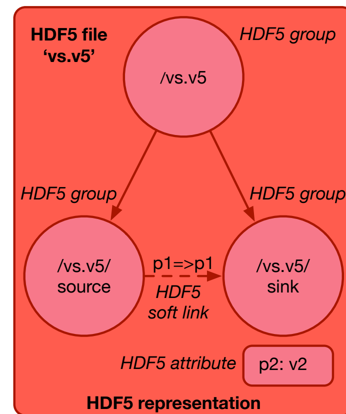
our requirement for consistent recovery of complex network graphs stored in memory during a crash. We can obtain versioning by simply implementing a new VOL plugin. Moreover, since networks are stored in Retro in HDF5 format, the visualization system can continue to manipulate them using the HDF5 API. This approach achieves both our design goals.

Figure 5 shows an example of using the Retro API to retrieve an attribute of a node from a visualization network stored as HDF5 objects. The query opens a group with name `node_name` from the HDF5 file with identifier `fid`. Then, it opens an attribute with name `attr_name` and reads the attribute value in the specified buffer `buf`. We create snapshots by using the `snapshot_now` operation which returns a snapshots identifier. In order to perform a past state query in state S, we simply wrap the same query with `asof_snapshot S`.

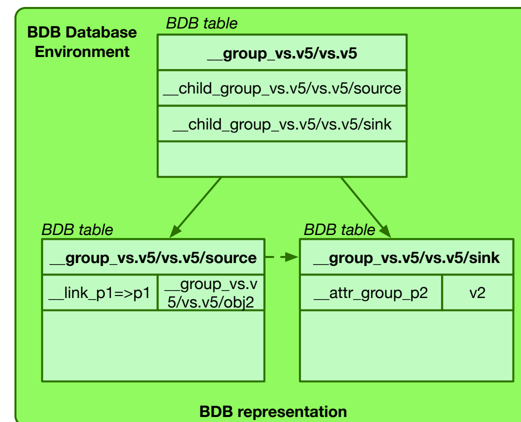
We developed a new HDF5 BDB VOL plugin to intervene between regular HDF5 API calls and Retro. The VOL plugin developer is responsible for mapping the HDF5 data model to the schema of the backing store. In our case, BDB schema is simple and consists of tables and key/value pairs within the tables. Thus, we can organize HDF5 objects as separate BDB tables and store their attributes as key/value pairs. Connections between HDF5 objects are represented as pointers between BDB tables.

We represent visualization networks as HDF5 objects, as shown in Figure 6a. In this example network, nodes *source* and *sink* are stored as subgroups under */vs.v5* group. HDF5 soft links are used for node connections and HDF5 attributes to store node parameters. The mapping from HDF5 objects to BDB tables in this example is illustrated in Figure 6b.

We are using Berkeley DB transactions for all accesses to networks, so that to support recoverability and crash consistency. We extended therefore HDF5 user API with operations to begin and end transactions, and with Retro API operations `snapshot_now` and `asof_snapshot`. Since Retro assigns a number as



(a) Network representation in HDF5



(b) Network representation in BDB

Figure 6: Visualization network representations

an identifier for snapshots, we preserve the mapping from time steps to snapshot identifiers using a separate mapping table stored in BDB. BDB and by extension Retro is optimized for relatively small data accesses and would not be efficient backend for large scientific time-dependent data. Our design therefore exploits the flexibility of VOL and continues to store data objects in the native HDF5 file system.

5 PERFORMANCE EVALUATION

We have implemented the versioned metadata module in Vish and conducted an evaluation study. The goal of the study is to evaluate the cost of managing versioned visualization networks using BDB snapshots, and compare it to a simple alternative design where network versions are stored in HDF5 files. In all cases the time-dependent datasets are stored in HDF5 files. Our experiments use Vish to visualize the time-dependent data set, measuring the performance of creating and loading network versions to visualize successive data points.

Our experimental setup consists of a machine with a 6-core i7 at 4.6GHz, equipped with 32GB memory, 6GB

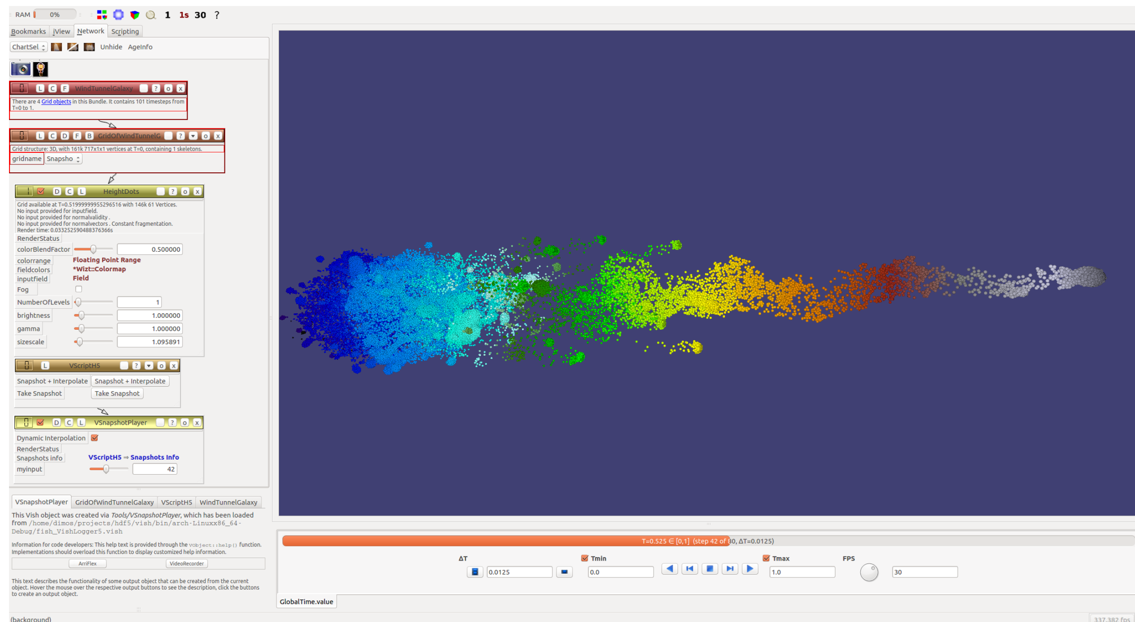


Figure 7: Displaying a time-dependent dataset on Vish, by loading snapshots and performing dynamic interpolation

GPU Nvidia GTX 1060 and an SSD drive. It is running Ubuntu Linux 16.10 with the Retro Berkeley DB C API 5.3, HDF5 1.9.3 with the VOL support, and the developer's edition of the Vish Visualization Shell. We used several datasets for our experimental setup ranging to several GB in size and used up to 81 time steps.

Our versioned metadata module using Retro/BDB implements the design we described in earlier sections. When the user performs modifications to the network through the GUI, the modifications are forwarded to the Retro/BDB VOL that performs them with a transaction that writes them into the BDB log. Later, when user creates a version (a snapshot) or terminates visualization, the transaction commits, flushing the log containing these modifications at commit time, and eventually writing the modifications lazily at low cost to the Retro store. Thus, all committed modifications are preserved after a system crash.

For the versioned metadata module using native HDF5 files we used a simple-minded design (called Native HDF5) that stores consecutive versions of the network as different HDF5 groups. HDF5 groups are analogous to directories in a typical file system. They may contain other groups (subgroups), attributes, datasets, etc. When creating a version, the Native system creates a new HDF5 group, writes it in its entirety to the new group, and associates that group with a unique identifier that allows to access the version later using a simple index to support interpolation. Since both the new group and the index are written to the file system at version creation time this approach does not provide crash consistency. Obviously, the simple-minded approach stores redundant data if only few parameters change between versions. A diff-based encoding and additional indexing would provide a more compact solution but would require more substantial development effort, including a more complex recovery procedure after crash to avoid version and index corruption.

Our experiment emulates a user workflow similar to Figure 4, where a user creates a versioned visualization for a time-dependent data set, starting with an initial network specification for the first data point. When visualizing at a certain time step the user adjusts the visualization parameters, stores them by creating a version (a snapshot, or a new group) and repeats these steps for all data points. At any point, the user can visualize the data for any time step, or play all time steps as an animation. We used a 2.8GB time-dependent data set from an astrophysical simulation [24] using 81 time steps. Figure 7 shows a single visualization time step of that simulation obtained in our experiment by loading snapshots and performing dynamic interpolation.

Our experiment measures the basic cost of writing the consecutively created versions (not including think time) and of subsequently loading the versions and visualizing consecutive data points, using either the simple versioning module with HDF5 (native HDF5), or the Retro/BDB VOL snapshot system. Our experiments also evaluate the dynamic interpolation feature that allows to only create network versions for selected data points omitting intermittent points and interpolating them dynamically at display time as seen in Section 4.2. Additionally, we also evaluate an alternative approach with one-to-one snapshot to time step mapping, which interpolates missing values between manually adjusted versions at creation time, rather than at display. This creates a network version for each data time point avoiding the need to interpolate

at run time. As we explained, the dynamic interpolation is preferred, but the one-to-one mapping approach can use simpler indexing.

In Figure 8 we see the cost of creating a snapshot by using the native HDF5 approach and the Retro/BDB VOL, when doing one-to-one mapping and dynamic. Initially we observe that the native HDF5 approach is costlier in snapshot creation, as expected. This is because it writes the entire network in HDF5, regardless of which parameters are updated. It takes 35 ms to create a snapshot when doing one-to-one mapping, and 44 ms when doing dynamic interpolation. The one-to-one mapping is slightly faster than the dynamic, since the creations are performed in a loop, without involving a user interface. We see better performance for Retro because this design only stores the updated parameters, rather than the entire network and keeps track of different versions natively. Note Retro costs reflect the writing of the transactional log, which includes writing to disk all updates to the committed version at commit. As we can see from Figure 8, Retro only takes about 0.5ms to create a snapshot in one-to-one mapping and less than 4 ms in dynamic, including both the creation of a new snapshot and the transaction commit. It is slower in dynamic, per version, since we perform more modifications between two snapshots. In general, even the 44 ms to create a version with native HDF5 are not perceptible for a user in an interactive setting. However, the one-to-one mapping that creates many versions, can cause noticeable latencies of up to a second with our data set.

Figure 9 shows the overhead of loading from a version when using the native HDF5 or Retro with one-to-one mapping and dynamic. Loading from a snapshot involves opening the snapshot for a specified time step and reading only the interpolated parameters. This avoids accessing the entire network but also incurs the overhead of snapshot indexing in order to find the specified snapshot. In this case, the native HDF5 is faster because while it writes the entire version it only has to open the specified version (from the corresponding HDF5 group within the same HDF5 file) and get the required parameters.

When loading a snapshot created with one-to-one mapping, we saw a response time of 565 µs on average, compared to Retro which took 663 µs. Next, when loading snapshots in dynamic, both methods are faster because there are less snapshots to manage. Native HDF5 takes 271 µs and Retro 364 µs. Obviously, version loading time is typically more important than creation since in the common case we expect versions to be created manually but displayed automatically. Nevertheless, we do not expect automatic visualization to be limited by the metadata accessing time, since the typical expected bottleneck is reading the large time-dependent

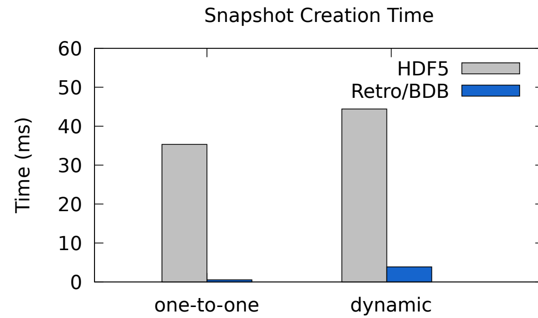


Figure 8: Time to create a snapshot of the current visualization network under different configurations.

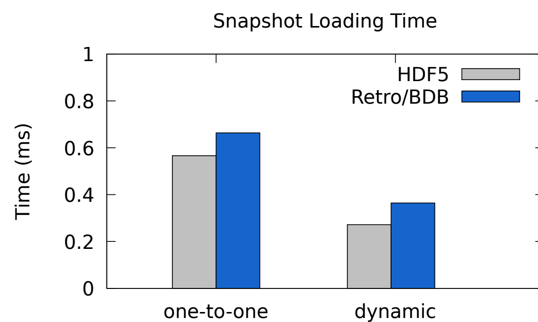


Figure 9: Time to load from snapshot and update the visualization network under different configurations.

datasets. In both cases of native HDF5 and Retro, we have around 0.5 ms response time to bring metadata for displaying a frame. This is negligible compared to the time to load the big data, and does not limit the frame rate.

6 CONCLUSION

We have presented a new, database snapshot system based method for managing visualization instructions for large time-dependent scientific datasets that are not well supported in current visualization systems. We explained how a simple programming model provided by the snapshot system makes it easy to manage versioned visualization metadata and to provide new labor-saving visualization system features such as version interpolation that reduce the manual effort needed to develop visualizations. We have described how we implemented our approach in the Vish visualization system and presented experimental results using a small data set from an astrophysical simulation indicating satisfactory performance while providing better reliability guarantees. Future work is using our approach for developing visualization for the large gravitational waves dataset.

Acknowledgment

This work is partially supported by National Science Foundation awards CNS-1318798, IIS-1251037, and IIS-1251095. We thankfully acknowledge Nikos Tsikoudis for his support with the Retro BDB.

7 REFERENCES

[1] R. B. Haber et al. Visualization Idioms: A Conceptual Model for Scientific Visualization Systems. In *Visualization in Scientific Computing*. 1990.

[2] Jankun-Kelly et al. A Spreadsheet Interface for Visualization Exploration. In *Proceedings of the Conference on Visualization '00*, VIS '00, pages 69–76, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press.

[3] Steven P. Callahan et al. VisTrails: Visualization Meets Data Management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 745–747, New York, NY, USA, 2006. ACM. doi:10.1145/1142473.1142574.

[4] Adobe Premiere. URL: http://www.adobe.com/products/premiere.html.

[5] Werner Benger et al. The Concepts of VISH. In 4th *High-End Visualization Workshop, Obergurgl, Tyrol, Austria, June 18-21, 2007*, pages 26–39. Berlin, Lehmanns Media-LOB.de, 2007.

[6] Ross Shaull et al. A Modular and Efficient Past State System for Berkeley DB. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pages 157–168. USENIX Association, 2014.

[7] Michael A. Olson et al. Berkeley DB. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association.

[8] HDF5. Hierarchical data format version 5. http://www.hdfgroup.org/, 2009. The HDF Group.

[9] Jennie Duggan et al. The BigDAWG Polystore System. *SIGMOD Rec.*, 44(2):11–16, August 2015. doi:10.1145/2814710.2814713.

[10] Gabrielle Allen et al. Solving Einstein's Equation on Supercomputers. *IEEE Computer*, 32(12):52–59, December 1999. doi:10.1109/2.809251.

[11] Miguel Alcubierre et al. The 3D Grazing Collision of Two Black Holes. *Phys.Rev.Lett.*, 87, 2001.

[12] Werner Benger et al. Using Geometric Algebra for Navigation in Riemannian and Hard Disc Space. In Vaclav Skala and Dietmar Hildebrand, editors, *GraVisMa 2009 - Computer Graphics, Vision and Mathematics for Scientific Computing*. UNION Agency, Na Mazinach 9, CZ 322 00 Plzen, Czech Republic, 2010.

[13] Werner Benger et al. Visions of numerical relativity. In A.Gyr et. al., editor, *Proceedings of the 3d International Conference on the Interaction of Art and Fluid Mechanics (SCART2000)*, pages 239–246, ETH Zürich Switzerland, Feb 28 – Mar 3 2000. Kluwer Academic Publishers.

[14] Abbott B. P. et al. Observation of gravitational waves from a binary black hole merger. *Phys. Rev. Lett.*, 116:061102, Feb 2016. doi:10.1103/PhysRevLett.116.061102.

[15] Kitware, Inc. Paraview - open-source scientific visualization, 2010. URL: http://www.paraview.org/.

[16] DOE/ASCI. Visit, 2002-2010. URL: https://wci.llnl.gov/codes/visit/.

[17] Kitware. Visualization toolkit, 2005. URL: http://www.kitware.org/.

[18] Visualization and Inc. Imagery Solutions. Ibm open visualization data explorer, 2000-2006. URL: http://www.opendx.org/.

[19] D. Stalling, M. Westerhoff, and H.-C. Hege. Amira - an object oriented system for visual data analysis. In Christopher R. Johnson and Charles D. Hansen, editors, *Visualization Handbook*. Academic Press, 2005. URL: http://www.amiravis.com/.

[20] L. Bavoil et al. VisTrails: enabling interactive multiple-view visualizations. In *VIS 05. IEEE Visualization, 2005.*, pages 135–142, Oct 2005. doi:10.1109/VISUAL.2005.1532788.

[21] Manasi Vartak et al. ModelDB: A System for Machine Learning Model Management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, HILDA '16, pages 14:1–14:3, New York, NY, USA, 2016. ACM. doi:10.1145/2939502.2939516.

[22] Stefano Nativi et al. Differences among the data models used by the geographic information systems and atmospheric science communities. In *In: Proceedings American Meteorological Society - 20th Interactive Image Processing Systems Conference. (2004*.

[23] Edwin Catmull et al. A Class of Local Interpolating Splines. In ROBERT E. BARNHILL and RICHARD F. RIESENFELD, editors, *Computer Aided Geometric Design*, pages 317 – 326. Academic Press, 1974. doi:http://dx.doi.org/10.1016/B978-0-12-079050-0.50020-5.

[24] Dominik Steinhauser et al. Simulations of ram-pressure stripping in galaxy-cluster interactions. 591:A51, 2016. doi:10.1051/0004-6361/201527705.