Epoch Persistence:
Safe, efficient, on-demand rendering for streaming data

Joseph A. Cottam
Indiana University/CREST
jcottam@indiana.edu

Andrew Lumsdaine
Indiana University/CREST
lums@indiana.edu

Abstract
Streaming data is always changing. Incremental results are incomplete, but often useful in their own right. Data analysis and rendering compete with each other for computational resources and access to core data structures because they are executing concurrently. This paper presents a set of data structures that ensures a visualization is internally consistent, and therefore interpretable. The data structures are based on persistent data structures, as commonly found in functional programming, but made more efficient by incorporating computational epochs. This paper also provides a definition for “consistency” that can be applied to visualizations. These data structures are used in the Stencil visualization system, which is used to benchmark the impact of epoch consistency. The net result of employing these data structures is that internally consistent incremental results are displayed as often as hardware allows without significantly impeding data loading.

1. Introduction
Modern computer architecture requires concurrency for efficient execution. Concurrent software requires attention to the correctness of the results. This is especially true for visualization when it is used to support decision making. Figure 1 presents two visualizations of the same data, but only Figure 1a presents the internal data relationships correctly. In both cases, data was streamed in and the input range was discovered incrementally. In the consistent case (Figure 1a), all of the rendered points are drawn from the same internal state. By contrast, Figure 1b includes points projected from a mixture of memory states. Otherwise said, Figure 1b is inconsistent and therefore not useful in analysis. In the case of Figure 1, the different memory states are formed as a new minimum or maximum value is observed. Each state is valid, but mixing results from different states is not. Unfortunately, without prior knowledge of the dataset, it is impossible to know which of the two images is cor-

Figure 1: Consistent and inconsistent Anderson’s Flowers.
rect. Both present clusters of three different types of flowers. Both mix the Versicolor and Virginica clusters.

The fundamental issue is that a visualization must be internally consistent in order to support analysis. The problem is that stream data analysis requires concurrent data processing and traditional data structures prevent different visualization tasks from occurring concurrently (namely rendering and data loading). Preventing these tasks, incurs a significant performance bottleneck on modern multi-core, shared-memory hardware. This paper presents a data structure that removes this bottleneck, argues for its correctness and presents evidence of its practically.

2. Related Work

This paper builds directly on our earlier work [8]. It presents the same technical tool (an epoch-persistent data structure). However, this paper provides three additional advances. First, this paper introduces the conceptual framework of consistency. The arguments for the correctness of the persistent data structure are strengthened in this paper over the prior paper because of their grounding in consistency. Second, this paper introduces a Global State Tracker (GST) in Section 6 for coordinating multiple stateful operators efficiently. Third, this paper provides more details on how Stencil translates from a stateful program to a safe concurrent program using operator relations originally introduced for created axes and legends [7].

Parallel execution in information visualization frameworks has been approached before in the context of both VTK [19] and the Java implementation of Protovis [14]. The VTK/Titan framework [16] employs parallelization in two places: the individual analysis operators and in the graphics pipeline. In VTK/Titan, preprocessed data is loaded into a central data store. Analysis operators are applied to that data store to generate new elements (whole new tables of data or columns appended to existing ones). Periodically, the data table is accessed by the rendering pipeline and images are produced. The parallel graphics pipeline is shared with the scientific visualization work. The analysis operations on the data store, such as the integrated Parallel Boost Graph Library [11], are free to employ parallelism as well. The VTK runtime coordinates when elements run, but only dictates the interface. This provides a great deal of flexibility to the analysis operators to exploit parallelism in their own algorithm specific way. In VTK’s arrangement, coordination between the analysis operators (and their parallel execution) and the rendering pipeline is mediated entirely by the central data store.

The Java implementation of Protovis took a different approach than VTK. In Protovis, there is no standard data store. Instead, there is a visual store and an input iterator from an external store. Combined, these two elements fulfill the role of a classic visualization data store. All stateful operations are assumed to be conducted in an external context and their results presented through the external store’s iterator. The principle difference between the Protovis and VTK strategy is that Protovis has the analysis expose the interface for the data store (enabling operator-driven flexibility) while VTK has the data store expose the interface to the operator. Operators controlled by the Protovis system itself are assumed to be true functions and are associated with elements in the scene graph. Though these functions may have interdependencies (e.g., relying on either parent or child bounds information), these dependencies are statically resolvable and only founding the arguments. This enables safe parallel execution driven by the branching structure of the scene graph [3].

Duke’s additional work in parallelizing information visualization processes focused on phased execution for processing large data sets [9]. Duke presented three domain-specific languages embedded in Haskell and demonstrated parallelization properties for each one. Further work by Piringer, et al. [18] shows that the layer metaphor in many information visualization platforms exposed exploitable parallelism in rendering.

The concepts and principles presented in this paper have been implemented in the Stencil visualization system [5]. Though Stencil itself is not the focus, understanding Stencil programs will help understand the relevant execution model. Stencil is a coordination language for constructing visualizations. All Stencil operations are in response to incoming data. Data are represented as streams of tuples, so dynamic data is at the core of the Stencil runtime system. An important goal of Stencil is that every frame of the visualization be internally consistent. Timing between events should not be able to introduce mis-representations. The relationship between timing and correctness has been explored in the context of in parallel and distributed systems [21]. In a distributed system, consistency rules dictate the when and where memory changes will be visible. In visualization, the consistency
rules need to address the display as well. In gen-
eral, the consistency rules presented in Section 4
provide a form of strong consistency in the central
data store. In this schema, render events serve to
amortize some of the costs of providing strong con-
sistency.

3. Concurrency

There are three general tasks types in a visualiza-
tion framework: initial analysis, consistency build-
ning and rendering. The relationship between these
three tasks is illustrated in Figure 2. The total
analysis length is equal across styles but execution
time is significantly reduced with concurrency. Se-
rial architectures require analysis operations to stop
while the data is brought into a consistent state and
then rendered. This is sufficient for static visualiza-
tions, but penalizes performance for streaming data.
Task-based concurrency can be introduced between
analysis, consistency building and rendering.

In a “half concurrent” architecture, analysis is
concurrent with consistency building or rendering,
but consistency building and rendering are still mu-
tually exclusive. Half concurrent architectures are
efficient when data changes are the principle drivers
of render events or rendering and consistency build-
ning share compute resources. When concurrency is
present between all three tasks, the “full concur-
rent” architecture, higher frame rates are achiev-
able. However, a full concurrent system usually re-
quires more memory and may actually run slower
than a half concurrent system if there are insuffi-
cient computational resources. Both half and full
concurrent systems have advantages over the serial
arrangement in a multi-core environment.

A consequence of concurrency is the need to sup-
port logically simultaneous accesses to some data
structures. For example, if the max or min values
matter to analysis and consistency building, then
each time a new max/min value is found, results
based on old values are no longer valid. The chal-
lenge for efficient execution is to only update old
results when needed and to do as little work as pos-
sible to ensure further analysis does not interfere
with building new consistent states. Persistent data
structures help achieve these goals.

4. Consistency

The main problem the techniques of this paper
address is building a consistent visualizations while
concurrently analyzing and rendering data. For a
visualization to be interpreted as a whole, it must be
internally consistent. Informally, a consistent visu-
alization is one where (1) all data loaded have com-
pleted all of their effects, and (2) that all effects of
data loaded are included in the rendering. An incon-
sistent visualization includes partial effects of some
data and cannot be cohesively interpreted. Even
more dangerous, inconsistent elements may not be
obvious and could lead to incorrect interpretations.
Figure 1 demonstrates a consistent and an inconsis-
tent rendering of a data set. Neither is unreasonable
on the surface, but incorrect conclusions can result
from Figure 1b. Visualization frameworks typically
provide for consistency by making analysis and ren-
dering mutually exclusive phases [3,10,14,19].

Consistency issues can arise in both static and
dynamic data situations. However, they are more
common in dynamic data since additional data may
be presented at any time. This discussion focuses
on dynamic data situations. In broad strokes, in-
consistency is like a race condition that appears in
the visualization. The advantage of a visualization-
specific understanding of race-conditions is it can
distinguish between malignant race conditions and
benign ones (where interpretability is not affected).
In benign race conditions, a different result is
achieved, but it does not affect the interpretation.
An example of a benign race condition is different rendering timings resulting in different intermediate visuals. The interpretation any given visual is still valid, but the exact visual presented may be different in different runs.

A formal definition of consistency enables reasoning about concurrency in visualization programs. This definition will be established in the context of data-flow programming, but the known equivalence to data state makes it generally applicable [4]. The significant components of data-flow programming are transformation operators and dependence. Figure 3 provides a fragment of a data-flow network that creates a scatter plot. Nodes represent operators and links represent data dependencies with. Consistency is defined in terms of the dependency networks and an auxiliary global state tracker (GST) function.

The GST creates an identifier for each memory state in the system. At its core, the GST is a timestamp of memory changes associated with a specific result. By associating analysis results with memory state identifiers, results from older memory states can be identified and updated. To accurately distinguish “old” memory states from new ones, the GST must have two properties. Specifically, the GST must be a sequence that is (1) it progresses with each state change and (2) it is repetition free. Providing these two properties in the GST means that comparing GST values from arbitrary selected times will always reveal potential consistency violations. If the GST does not meet condition 1, then state changes may occur that comparing the GST does not catch. This immediately defeats the purpose of using the GST. Repetition freedom, condition 2, is required because GST values may be compared across large time spans. If repetitions are allowed, there is the potential for a sampling effect that lead to incorrectly concluding that a state has not changed when it actually has.

A simple timestamp satisfies the GST requirements. However, using a counter that increments whenever the state updates provides more detailed information that can be used for more efficient scheduling. The rest of this section discusses where count updates need to be placed and provides a formal definition for consistency based on a counter GST. For the purposes of description, it is assumed that inputs are processed one at a time.

Logically, the GST can be thought of as an object that returns a state identifier. Whenever a result is ready to be stored, the GST is invoked and the state identifier is stored with the result. Whenever a state change occurs, the GST is invoked in a way that causes the state identifier to change. (This could be achieved by passing different arguments to a procedure or invoking different methods on an object, depending on the implementation of the GST.) Figure 3 includes GST calls in gray following each calls to the stateful “Scale” operator. Assuming that GST calls are appropriately placed, consistency is defined in terms of a collection of relations among the visual variables (i.e., position, color, shape, etc.) [2] in groups of rendered elements (e.g., points in a scatter plot). A visual variable in a group of elements is consistent if all of the GST values of each entity in the group belongs to one of two relations:

**Sequence** Each of values of the variable corresponds to a unique GST value.

**Snapshot** All the values of the variable are associated with the same GST.

The sequence relation is applies when the data’s order is significant. This occurs when the context of a value depends only on values that have come before it (e.g., running maximal value). The snapshot relation is used whenever working with dynamically discovered global properties. The consistency of a group can be directly verified by examining the GST values.

A group of rendered elements is consistent if all visual variables in the group are consistent and one of the following conditions holds for all pairs of variables.

**Sequence × Sequence** Two sequence-consistent groups are automatically consistent with each other.

**Sequence × Snapshot** The greatest GST in the sequence-consistent variable is less than or equal to the GST for the snapshot-consistent variable.

**Snapshot × Snapshot** The GSTs are the same.

Groups are consistent with each other if all pairs of visual variables between the groups are consistent according to the same rules as intragroup consistency. Finally, a visualization is consistent if all groups are consistent with each other.

GST tracking only applies to stateful operations. For simplicity, the results of pure functions and constant values are considered under the snapshot relation, with the that they automatically take the current value of the GST.
4.1 Validity of Consistency

In a consistent visualization based on valid analysis, the parts of the visualization can be safely compared to each other because all represented values have had the opportunity to be influenced by the same data. This conforms to an intuitive notion of visualization interpretability while providing a basis to evaluate actual program performance. This section presents arguments for the validity of the consistency relationship with respect to visualization interpretability.

At the attribute level, snapshot consistency applies when a visual element is based on a global, dynamically computed value. For example, snapshot consistency applies if x-position when projecting into a fixed range from a dynamically discovered domain (e.g., converting from \textit{a priori} unknown input values to between 0 and 1). In this case, earlier computed positions may need to change in response to a value loaded later. Assuming snapshot consistency is lost because a subset of x-positions are based on a global maximum, while a disjoint subset is based on a different global maximum. In such a case, the two subsets cannot be visually compared safely. However, the distinction between the sets may not be clear. Figure 1b exhibits exactly this problem. Each of the three species groups in Figure 1b (indicated by fill color), is plotted with a different memory state but all points within a group is drawn from the same memory state. Therefore, the visualization cannot be safely interpreted when snapshot consistency is expected but not provided. The same argument extends to the intergroup, intra-group and full visualization consistency.

Sequential consistency is applicable when a visual element depends only on computations performed thus far. For example, sequential consistency applies if size depends on a running total. In the simplest case, if size were so defined, an element’s size would never change after it is initially set. (Updates to existing visual elements based on newly presented data are considered deletions, followed by insertions. Therefore the immutability of created elements is preserved in the model, though the implementation may perform direct memory changes.) To conform to concurrent computing notions of consistency, a series of state changes in an operation must be ordered [21]. The results of calculations based on those updates can be put into a corresponding order. The GST tracks the order of those updates. If two size definitions are ambiguously ordered (e.g., have the same GST), it is unclear which reflects the state updates of the other. This constitutes a violation of the rules of consistency from concurrent computing and invalidates comparisons between the two elements.

Consistency between snapshot- and sequence-consistent visual attributes requires that the snapshot be computed at the same time or after the last change in the sequence-consistent group. The reasoning behind this follows the same lines as the reasoning for snapshot to snapshot consistent groups. The snapshot consistent element, to be considered in conjunction with the sequence consistent element, must reflect the same underlying state. If the GST for the snapshot consistent group is behind the largest GST in the sequence consistent group, it may not reflect at least one new data point (the one that created the greater GST in the sequence consistent group) and comparisons are potentially invalid.

The consistency relationship as presented provides a lower bound on interpretability of a visualization. If snapshot consistency is violated, large groups of items may be incomparable. If sequence consistency is violated, then individual elements within the visualization may be incomparable. Intelligently implementing the GST and related storage can result in significant space and time savings.

5. Persistent Data Structures

The general idea behind persistent data structures is that old versions of the data structure are retained as the values in the data structure change [17]. For example, adding a new node to the start of a singly linked list leaves the old list unchanged. Replacing all nodes from the start of a singly linked list to a changed node in the middle has the same effect. The key point is that old views of the data still exist, even as new ones are created. Persistence supports consistency because the persistent copies can be worked on independently.

Simply providing persistence is not enough. For high-performance streaming visualization, when and how persistent views are made matters. A copy-on-write first write policy is too costly for large data structures. Linked data structures also become more costly as they grow unless changes can be isolated to small regions. The solution to this problem of efficiency is found by looking at \textit{when} different versions need to co-exist. Simply put, rendering is the only time values are directly observed. Making copies around rendering times leads to an efficient solution. This section describes when and how an intermittently persistent data store decides to make
The principle data structure for a visualization system is the data store \[13\]. The data store contains a description of visual elements to render (positions, colors, etc.) and selected input data for labeling, producing tabular reports and consistency calculations. To fulfill its roles in a visualization framework, the data store is generally conceptualized as a set of 2D matrices (called tables) with following abilities: (1) Insert a set of values into a table; (2) Modify values in a table; (3) Key-based addressing for retrieving items based on a property value; and (4) Index-based lookup for sequentially reviewing a subset of items.

Typically, insertion and modification occur frequently (often multiple times per millisecond) while key-based lookup is less frequent. These operations involve a few data points each, but the access patterns are determined by input data and are thus unpredictable. In contrast, index-based lookup is used to access large portions of the data in a regular pattern. It is typically used in consistency construction and rendering. The goal of the intermittently persistent data store implementation is to support incremental data change operations concurrently with global consistency construction and rendering. Furthermore, to reduce overall system work, the merge consistency construction with later results was given a high priority (referred to a “confluence” for persistent data structures). The infrequency and actual significance of rendering is similar to a transactional epoch and leads to a change-list tracking strategy related to those used in Software Transactional Memory (STM) \[20\].

The intermittently persistent data store can be conceptualized as having two main parts and two support structures. The overall data structure and the flow of the data through the parts is shown in Figure 4. The main parts are the “Young List” and “Tenured List”. These are roughly equivalent to a back and front buffer in double buffered graphics. The young list is a list of changes that have been made to the data store since the initiation of the most recent consistency building phase. The young list is used to support frequent updates to small amounts of data, not rollback as in STM. Analysis that refer to the data store preferentially use values from the young list. The tenured list is a set of values that have been put into a consistent state. Like the front buffer in double buffered graphics, it is not modified after it is built. When a persistent view is required by the system (such as during rendering), the tenured collection can be returned at any time. The tenured list supports efficient iteration but only supports modification through expensive copying operations, so updates are best done in batches.

Implementation decision significantly effect performance of the intermittent persistent data structure. For Stencil, the implementation was driven by the observed access patterns when loading streaming data. To keep overhead for updates to the young list low, it is implemented as a row-based store. A column-based store \[13\] backs the tenured list because consistency construction typically updates entire columns at a time.

Because the Tenured List is guaranteed to be consistent, making the persistent view requires additional work called ‘consistency construction.’ Two support structures are also used in Stencil’s implementation to reduce the amount of time it takes to make these persistent copies of the Tenured List. The support structures in the data store are the “Transfer List” and “Shadow Copy.” These two structures are directly tied to the flow of data through the system. Initially the Transfer list is empty and analysis results are placed in to the Young list. When a new persistent copy is requested, a semaphore blocks both read and write access to the data structure while the following operations are performed.

- The Young and Transfer Lists are swapped.
- A shallow copy of the Tenured list is also made (pointers for each column, not their data) during this; this is called the “Shadow Copy.”
- A snapshot of analysis operators involved used for consistency construction is taken

Once these operations are complete, access is re-
stored and data loading resumes concurrently with consistency construction. For consistency construction, the Transfer list is converted from a row-store into a column store and merged into the shadow copy. When a Shadow copy column is modified for the first time, a copy of the specific column is made before the modification. When all new updates have been merged, final consistency construction is performed using the operator snapshots created earlier. After constancy construction, the tenured list is replaced by the shadow copy and the transfer list is cleared.

Capturing operator snapshots enables consistency construction to proceed without interference from further data loading. They are not needed if purely functional operators are used. The actual process Stencil uses for snapshot creation is discussed more in Section 6.

6. Stencil Implementation

The Stencil visualization system implements the persistent data structure discussed in Section 5 to achieve a half-concurrent model (as defined in Section 3). This section describes that implementation in more detail, and sets up the performance discussion in Section 7. The implementation decisions described here represent one of many ways that persistent data structures and global state tracking can be implemented. Many of the components described here are also used for automatic guide creation [6] and automatically translating programs from data flow to data state models [7]. The Stencil system is implements a half-concurrent model from Section 3 because render events can trigger stateful updates.

Stencil represents a visualization as a set of dependencies between input data streams and transformation operators. An example dependency graph can be seen in Figure 3, which is used to compute the scatterplot of Figure 1a.

The Stencil compiler transforms a program into a concurrent form with the help of three operator metadata relations (discussed in detail in Section 6.1). The relevant metadata are Counterpart (relating Writers to Readers), StateID (tracking the actual transitions made by Writers and used to construe the GST from Section 4) Viewpoint (capturing a snapshot of analysis state). The viewpoint relation enables task-based concurrency, but is potentially costly. Section 5 describes how persistent data structures are used to reduce the cost of viewpoint support.

Stencil’s implementation of the GST is based on vector timestamps [21]. Every analysis operators contributes to a collective GST. The individual parts of the distributed GST are accessed through the stateID operator relation (see Section 6.1). Each operator increments an internal state tracker called its stateID. These operator-level stateIDs must conform to the progress and repetition requires for the overall GST. The global GST is the list of all operator stateIDs in a stable order for the current execution. This distributed GST can be thought of as a number in an arbitrarily large base system. Each operator contributes a single “digit” to the collective number.

Stencil does not store a GST value with every result. Instead, visual attribute groups are associated with the GST of the last time the entire visualization was brought into a consistent state. This reduces the storage overhead from linear in the number of data points stored to a constant for any given visualization. Furthermore, because Stencil never retains more than two persistent states, the “no-repetitions” requirement for the GST and the contributing stateIDs is relaxed to “stateIDs must not repeat between consecutive consistency constructions.”

The distributed GST schema and associating GST with attribute groups (instead of individual results) enables quick detection of which parts of a visualization need further work to made consistent. Only column headers need to be examined (not each cell in the data store). Each visual attribute (e.g., color, position, etc.) in the visualization that requires snapshot consistency is associated only with the digits of the GST that are related to the operators that affect it. Visual attributes only require further processing when the associated digits change, otherwise the attribute can simply take the current GST.

This fine-grained GST also gives insight into the significance of different race conditions in a visualization. In general, if two elements depend on the same stateful analysis operator and become inconsistent then a malignant race condition has occurred. However, if two elements do not share a stateful analysis operator, than they effectively have disjoint GSTs in Stencil’s implementation. Under the global GST, the two may appear inconsistent. However, unpacking the digits of GST reveals the corresponding race condition was benign and can be ignored.
6.1 Operator Relations

Stencil uses operator relations support many of its unique features. This includes providing for global consistency and concurrently loading data while building consistent states. This section section describes three core operator relations (these relations are not unique to consistency and also used for other Stencil transformations [7, 7]).

The basic problem is that different phases of execution have different restrictions on valid operations but the different phases must also remain related to each other. Operator relations provide a way to facilitate that coordination, without encumbering the visualization itself with details. Instead, each transformation operator must provide for a number of high-level relations.

Operators in Stencil are procedures that map inputs to outputs according to one or more rules. Operators may contain multiple, related mapping rules. Each mapping rule in an operator is referred to as a “facet.” Operators roughly correspond to objects in Java and facets to methods (though they are not necessarily implemented in this way). Facets are decorated with metadata that relates them to general classes and to each other.

The most broadly used relation is *counterpart*. Counterpart provides a means for immediate, read-only access to a mapping operator and its current state. By definition, a counterpart cannot be a writer facet. The behavior of a counterpart falls into one of three categories. First, if the original operator is a continuous, then the counterpart computes the same value as the original would, given its current memory state, but does not modify the memory. Second, if the original operator is categorical, then the counterpart queries for the prior reaction to the input and does not modify memory. Third, the counterpart may signal an error if the input passed is not valid in the current memory state.

Figure 5 provides example behavior for continuous and categorical counterpart operators. (Count implements a categorical projection, and Scale implements a continuous projection. The .query methods are counterparts to the .map operators.) Regardless of the category, the counterpart operator is inherently repeatable, since it does not modify memory. (A more complete discussion of the counterpart relation is given in [5].)

The *viewpoint* relation goes further than counterpart. While counterpart returns a facet that will not change values, viewpoint returns a new operator that will never be changed. Viewpoint enables concurrent execution of potentially state changing actions (i.e., loading new data) with the creation of a consistent state (i.e., calculation of global properties with their dependents). Viewpoint essentially takes a snapshot of the operator’s current state and provides it through a new operator. Stencil guarantees that the new operator will not have any writer facets invoked (e.g., all calls will be to counterpart facets), so it may be simpler than the full operator. The snapshot supporting viewpoint may be taken eagerly as a deep copy or it may be based on persistent data structures. The core requirement is that the viewpoint’s backing state does not change, even when the original does.

The *stateID* relation is part of the implementation of the GST. StateID enables operators to communicate information about state changes, without revealing implementation details. The stateID relation takes an operator and returns an operator facet such that whenever the internal operator state changes, the return value of the stateID facet also changes. In practice, the stateID facet is typically implemented as a method that returns an internally held counter. The counter is incremented whenever state changes actually occur. This affinity to actual state changes reduces unnecessary calculations by creating a close correspondence between GST changes and actual consistency issues. StateIDs only need to be provided by stateful operators, since the effect system allows Stencil to identify which operators need to contribute to the distributed GST.

7. Performance

We tested the Stencil system for both task and data-level parallelism. The main analysis was conducted using a space-filling curve visualization (see Figure 6). Each new element was placed at the head of the layout (point D) and other points shifted ac-
Our investigation has two major findings. First, that serial implementations were very sensitive to frame rates while the concurrent one was not. Each framework configuration was executed using different maximum frame rates (between 1ms and 8000ms enforced delay between render requests). Actual frame rates depended on the amount of data loaded, but were comparable between all frameworks for the same amount of data. Each test configuration was run ten times, with results presented in Figure 7 (variance was less than 2% for all data sizes above 256). The concurrent Stencil implementation exhibited no significant change in performance for any render delay. Prefuse (and the serial Stencil implementation), were very sensitive to the render delay. Execution times were inversely related to frame rate. This constancy of the concurrent performance vs. the variability in the serial performance indicates that the concurrent architecture, at a minimum, provides a less delicate system that is capable of performing tasks in spite of configurations that are adverse to more traditional implementations.

The second finding is that, independent of frame rate, the concurrent architecture eventually holds a significant advantage. For most render delays the timing curve eventually experienced a phase transition in the serial versions (appearing as a ‘hip’ in Figure 7). This phase transition occurs when total render time exceeding the render delay. Eventually, the serial implementations only loads a few data points between renderings (which then occupy ≥99% of the runtime). There is no hip in the 8000ms serial case because it finishes loading the data before 8000ms have passed. A similar phase transition was not observed in the concurrent Stencil version. Furthermore, no transition is expected since the amount of work required to initiate the consistency construction and rendering tasks is constant for a given visualization (i.e., the number of column pointer copies is fixed). When the Stencil system is applied to the 23 million data point dataset, the runtime is approximately 20s regardless of the render delay. From these pieces of evidence, we conclude that the serial architecture will eventually fall behind the concurrent one by a significant amount.

8. Future Work

The enter/exit system employed by D3 provides many of the benefits of epoch consistency. The young list corresponds to the new entry set. Consistency building is mixed between the ‘enter’ events and the regular mapping. Though threading model in Javascript did not allow true parallelism until recently (with the introduction of the web worker API), asynchronous callbacks certainly lead to concurrency. Epoch consistency may be adapted to D3
for better responsiveness in large scale visualizations using streaming data.

Parallelization in Stencil has focused on working with dynamic, potentially infinite data sources. However, many visualizations work principally with static, finite data. Dynamic data represents a more general case, but additional optimization and parallelization opportunities exist for static, finite data. Investigations into these additional opportunities is currently being pursued with the addition of a ‘table’ data source to complement the ‘stream’ data source. Difficulties in interoperating between algorithms designed for static and dynamic data are anticipated, but work on incrementalization provides directions to pursue [1,12].

Including more operator meta-data, such as operator commutativity and associativity, would enable parallelism inside of the analysis tasks. This would open up opportunities for Map/Reduce style execution. Similarly, since many consistency calculations are embarrassingly parallel, GPGPU techniques might be applied.

9. Conclusions

Persistent data structures provide one way to support meaningful concurrency support in visualization frameworks. This paper has described a useful persistent data structure variant that supports data analysis and rendering concurrently. This data structure provides consistency in the visualization while supporting true parallel execution of several related tasks. The net result is faster visualizations on dynamic data. The parallelization techniques and definition of consistency can be applied in other frameworks to provide similar support for streaming data and consistency preserving parallelization.

References


