Dead Timestamp Identification in Stampede

Nissim Harel *
nissim@cc.gatech.edu

Hasnain A. Mandviwala *
mandvi@cc.gatech.edu

Kathleen Knobe †
knobe@crl.dec.com

Umakishore Ramachandran *
rama@cc.gatech.edu

Abstract

Stampede is a parallel programming system to support computationally demanding applications including interactive vision, speech and multimedia collaboration. The system alleviates concerns such as communication, synchronization, and buffer management in programming such real-time stream-oriented applications. A coarse-grain dataflow graph is often a convenient representation of such computations. *Threads* are loosely connected by *channels* which hold streams of items, each identified by a *timestamp*. A thread may read a timestamped item from one channel, perform some analysis on that item and write an item with that same timestamp onto a different channel. Useful work is signified by a timestamp making its way through the entire pipeline of threads and channels. Because threads operate at different speeds, some timestamps will not make their way through the entire pipeline.

There are two performance concerns when programming with Stampede. The first concern is *space*, namely, ensuring that memory is not wasted on a timestamp (*i.e.,* items bearing this timestamp) that is not fully processed. The second concern is *time*, namely, ensuring that processing resource is not wasted on a timestamp that is not fully processed. Prior work on Stampede [7, 8, 9, 10, 11] addressed these concerns separately. Our earlier work on static scheduling ensures that the pipeline only works on timestamps that are fully processed. However, this technique applies only for a restricted set of task graphs. Similarly, our earlier work on garbage collection identifies and removes garbage items from channels. Due to the global nature of this algorithm there can be a significant wastage of memory resource with this approach. In this paper we introduce a single unifying framework, *dead timestamp identification*, that addresses both the space and time concerns simultaneously, and does not have some of the limitations of our prior work. Dead timestamps on a channel represent garbage. Dead timestamps at a thread represent computations that need not be performed. This framework has been implemented in the Stampede system. Experimental results showing the space advantage of this framework are presented. Using a color-based people tracker application, we show that the space advantage can be significant (up to 40%) compared to the previous techniques for garbage collection in Stampede.

1 Introduction

1.1 Stampede

There is a class of emerging applications spanning interactive vision, speech, and multimedia collaboration that are computationally demanding and dynamic in their communication characteristics. Such applications are good candidates for the scalable parallelism exhibited by clusters of SMPs. Applications in this class have requirements that are fairly unique that set them apart from highly studied scientific applications. Firstly, due to their interactive nature, they require special handling of time for management of temporally evolving data. For example, a stereo module in an interactive vision application may require images

*College of Computing, Georgia Institute of Technology
†Compaq Cambridge Research Lab

with corresponding timestamps from multiple cameras to compute its output, or a gesture recognition module may need to analyze a sliding window over a video stream. Secondly, both the data structures as well as the producer-consumer relationships in such applications are dynamic and unpredictable at compile time. Existing programming systems for parallel computing do not provide the application programmer with significant support for such temporal requirements.

A coarse-grain dataflow graph is often a convenient representation of stream-oriented computations. For example, Figure 1 shows a simple vision pipeline. A characteristic of such dataflow graphs is that upstream modules (*e.g.,* digitizer) tend to take less time for processing compared to downstream modules (*e.g.,* Hi-fi tracker). Correspondingly the upstream modules also tend to generate larger amount of data per unit computation than the downstream ones. For instance, the digitizer produces pixel representations of camera images and sends it down the pipeline, while the Hi-fi tracker produces tracking output of objects of interest in a scene.

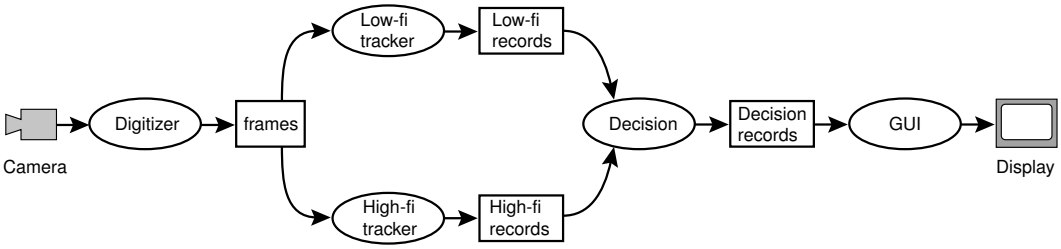

Figure 1: **A simple vision pipeline.**

A major problem in implementing these kinds of application in parallel is "buffer management". The *Low-fi tracker* and the *Hi-fi tracker* analyze the frames produced by the digitizer for objects of interest and produce their respective tracking records. The *decision module* combines the analysis of such lower level processing to produce a decision output which drives the *GUI* that converses with the user. From this example, it should be evident that even though the lowest levels of the analysis hierarchy produce regular streams of data items, four attributes of this class of applications contribute to complexity in buffer management as we move up to higher levels:

- Threads may not access their input in a strict stream-like manner. In order to conduct a convincing real-time conversation with a human, a thread (*e.g.,* the Hi-fi tracker) may prefer to receive the "latest" input item available, skipping over earlier items.

- Newly created threads may have to re-analyze earlier data. For example, when a thread (e.g. a Low-fi tracker) hypothesizes human presence, this may create a new thread (*e.g.,* a Hi-fi tracker) that runs a more sophisticated articulated-body or face-recognition algorithm on the region of interest, beginning again with the original camera images that led to this hypothesis. This dynamism complicates the recycling of data buffers.

- Datasets from different sources need to be correlated temporally. For example, stereo vision combines data from two or more cameras, and stereo audio combines data from

several microphones. Some analyzers may work multi-modally, *e.g.,* by combining vision, audio, gestures and touch-screen inputs.

- Since computations performed become more sophisticated as we move through the pipeline, they also take more time to be performed. Consequently, not all the data that is produced at lower levels of the processing pipeline will necessarily be used at the higher levels. As a result, the datasets become temporally sparser and sparser as the higher levels of processing correspond to higher-level hypotheses of interesting events. For example, the lowest-level event may be: "a new camera frame is captured", but a higher-level event may be: "John has just pointed at the bottom-left of the screen". Nevertheless, we need to keep track of the "time of the hypothesis" because of the interactive nature of the application.

These features imply two requirements. First, data items must be meaningfully associated with time, and second, there must be a discipline of time that allows systematic reclamation of storage for data items (garbage collection).

Stampede is a parallel programming system designed and developed to simplify programming of such applications. The programming model of Stampede is simple and intuitive. A Stampede program consists of a dynamic collection of threads communicating timestamped data items through *channels*. Threads can be created to run anywhere in the cluster. Channels can be created anywhere in the cluster and have cluster-wide unique names (similar to Unix sockets). Threads can *connect* to these channels for doing input/output via *get/put* operations. A timestamp value is used as a *name* for a data item that a thread puts into or gets from a channel. The runtime system of Stampede takes care of the synchronization and communication inherent in these operations. In addition, by imposing a discipline of time at the application level, the runtime system takes care of transparently managing the storage for items put into or gotten from the channels.

1.2 Live and dead timestamps

Every item on a channel is uniquely indexed by a *timestamp*. Typically a thread will *get* an item with a particular timestamp from an input connection, perform some processing[1] on the data in the item, and then *put* an item with that same timestamp onto one of its output connections. Items with the same timestamp in different channels represent various stages of processing of the same input.

The time to process an item varies from thread to thread. In particular, earlier threads (typically faster threads that perform low level processing) may be producing items *dropped* by later threads doing higher level processing at a slower rate. Only timestamps that are completely processed affect the output of the application, while a timestamp that is dropped by any thread during the application execution is *irrelevant*. The metric for efficiency in these systems is the rate of processing *relevant* timestamps (*i.e.,* timestamps that make it all the way through the entire pipeline). The work done processing irrelevant timestamps represents an inefficient use of processing resources.

[1]We use "processing a timestamp", "processing an item", and "processing a timestamped item" interchangeably to mean the same thing.

At a coarse grain time marches forward in this class of applications. That is, the timestamps being processed, in general, tend to monotonically increase with time. Old items (no longer needed by any thread) should be eliminated to free storage. However, since at a fine grain, a thread may be examining individual timestamps out of order, it is not trivial to determine when an item can be eliminated.

The algorithm developed in this paper determines a *timestamp guarantee* for each node (thread or channel). For a given timestamp T, the guarantee will indicate whether T is *live* or whether it is guaranteed to be *dead*. A timestamp T is live at a node N if (a) T is a relevant timestamp, *and* (b) there is some further processing at N on T (*i.e.,* T is still in use at N). Otherwise T is a dead timestamp at node N. If the node is a thread, "in use" signifies that the node is still processing the timestamp; if the node is a channel, "in use" signifies that the timestamp has not been processed by all the threads connected to that channel.

A timestamp may be live at a node at some execution time but dead at a later time. A timestamp may be live at one node but dead at another. Dead timestamps are interpreted differently depending on the node type. If the node is a channel, items in that channel with dead timestamps are garbage and can be removed. If the node is a thread, dead timestamps that have not yet been produced by the thread represent dead computations and can be eliminated. Note that dead computation elimination is distinct from dead code elimination. It is not the static code that we eliminate but rather an instance of its dynamic execution.

A unified view of garbage collection and dead computation elimination results from a single algorithm that determines dead timestamps at all nodes (thread and channels). This identification of dead timestamps is used on channels to indicate dead data (garbage) and at threads to indicate dead computations.

1.3 Background

There are two apparently unrelated technologies in Stampede, scheduling [7] and garbage collection [8].

Our earlier garbage collection work calculates lowerbounds for timestamp values of interest to any of the application threads. Using these lower bounds, the runtime system can *garbage collect* the storage space for useless data items on channels. This algorithm which we refer to as *transparent GC*, is general and does not use any application-specific property.

Our earlier scheduling work computes an ideal schedule at compile-time. It generates a schedule that will pick a timestamp and complete the processing of that timestamp through the entire application pipeline. Thus only relevant timestamps are processed in the entire pipeline. Garbage collection is trivial in this environment. The schedule is totally static and the last use of each item on each channel is clear at compile-time. However, it only works on a restricted class of programs. The task graph must be static and further, the time for a thread to process a timestamp is fixed and predictable at compile-time.

The dead timestamp identification and elimination work presented in this paper develops a *single unified* technique that is used for both garbage collection and scheduling. The focus, type of task graph for which this technique is applicable, and the aggressiveness of the technique are in between those of static scheduling and transparent GC (see figure 2 for a summary of the three techniques):

Analysis	Scheduling	**Dead timestamp identification**	Garbage collection
State	Prior work	**This paper**	Prior work
Task graph restrictions	Static task graph and predictable time per item	**Static task graph**	None
Aggressiveness	High	**Medium**	Low
Focus	Elimination of irrelevant work	Elimination of **irrelevant work and garbage**	Elimination of garbage

Figure 2: **Summary of three optimizations**

- Focus

  The main focus of the scheduler is to eliminate work on irrelevant timestamps. The focus of transparent GC is to remove items that have already been used by all possible consumers. The new technique does both.

- Type of task graphs

  The transparent GC works for arbitrary dynamic graphs. Tasks can be spawned and killed unpredictably and can connect arbitrarily to channels. Even the number of tasks is unbounded. On the other hand, the scheduler assumes that not only is the task graph known at compile time, but the time required for a given task is constant over all timestamps. There is a class of interesting applications for which this assumption can be quite restrictive. Background subtraction is a simple common example, in which the current frame is subtracted from a fixed background image to locate differences. Then a neural net computation analogous to a correlation is performed. The time for this computation depends on the number of pixels in the difference which could vary with each frame.

  The new technique is applicable for task graphs wherein neither the time a thread takes to compute any given timestamp need be constant, nor the length of identically-timed sequences need be long. The only restriction is that all *potential* threads and connections that may ever appear during the application execution be known at compile time to aid the analysis.

- Aggressiveness

  The new technique, being less restrictive compared to the scheduling work, is less aggressive and less efficient at minimizing the computation on dead timestamps. However, since the new technique has access to the application level task graph, it is more aggressive at eliminating garbage than transparent GC.

The rest of the paper is organized as follows. We present a new unified algorithm for dead timestamp identification in Section 2. Implementation details of this algorithm in Stampede are given in Section 3. Performance results showing the reduction in memory footprint of the new algorithm compared to the previous garbage collection techniques in Stampede are

shown in Section 4. Comparison of the dead timestamp identification algorithm to other related research is presented in Section 5. Concluding remarks and future directions are discussed in Section 6.

2 Dead Timestamp Identification

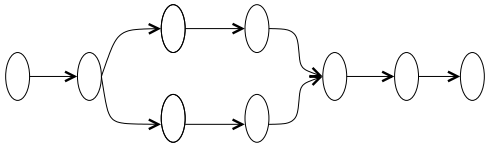

Figure 3: **An abstract task graph**

We can describe an application in Stampede in terms of a task graph. This task graph is a bipartite directed graph of *nodes*, each of which is either a *thread*, which performs a certain computation, or a *channel*, which serves as a medium for buffer management between two or more threads. Directed edges between nodes are called *connections*. A connection describes the direction of the data flow between two nodes. Both types of nodes, threads and channels, have input and output edges called input and output connections. The graph in Figure 3 represents this abstract view of the task graph in Figure 1.

Dead timestamp identification is a process by which the runtime system identifies for each node what timestamps are provably of no use. This forms the basis of both garbage collection and dead computation elimination.

The information that is propagated among the nodes is a guarantee that can be used to *locally* separate live timestamps from dead ones. The dead timestamp algorithm generates two types of guarantees: forward and backward. The *forward guarantee* for a connection identifies timestamps that might cross that connection in the future. The *backward guarantee* for a connection identifies timestamps that are dead on that connection.

Both forward and backward processing are local in that, based on guarantees available locally, they compute a new guarantee to propagate forward or backward along a connection to neighboring nodes.

Next we will describe how possible dependences between connections, in general, and the monotonic property of a connection, in particular, help in determining guarantees on each connection. Then, we will discuss how forward and backward guarantees on a specific node are determined. Finally, transfer functions that help optimize the basic algorithm are described at the end of this section.

2.1 Monotonic and dependent connections

Monotonicity is an attribute of a connection that indicates the forward direction of time. The progression of time is, of course, controlled by the code in the threads. Monotonicity occurs, for example, in the common case of a thread's input connection, where the thread issues a command to get the latest timestamp on an input channel. Assume the timestamp it gets is T. Then as part of managing its own virtual time, it may issue a command that

guarantees it is completely done with any timestamp below T on that channel. Such a guarantee from a thread on an input connection from a channel indicates that timestamps less than T are irrelevant (and can be gotten rid of from the channel) so far as this thread is concerned. Both thread to channel and channel to thread connections can be monotonic.

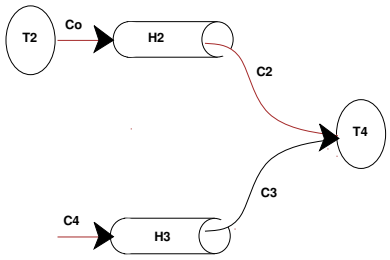

Figure 4: **A sample dependent task graph**

Consider the task graph in Figure 4. Assume that thread T4 gets a timestamp from C2 only if it gets the same timestamp from C3. This happens, for example, in stereo vision, where a thread gets the latest timestamp from one channel and then looks in the other for the matching timestamp. Connection C2 is said to be a *locally dependent* on connection C3. This relationship is not commutative, that is, the relationship (C2 depends on C3) does not imply that (C3 depends on C2). Notice that in this example, for a timestamp TS, $Next\_TS(C2) = Last\_TS(C3)$, because the thread T4 gets a timestamp from C2 only if it gets the same timestamp from C3.

We may also view monotonicity as a type of dependency where a connection, say C, is loosely dependent on itself. In the case of a strictly monotonic connection, the next timestamp to be processed must be greater than the last one processed, or, $Next\_TS(C) > Last\_TS(C)$. But this view is not limited to strictly monotonic connections. In fact, we can describe any level of monotonic guarantee in terms of a dependency. Every connection is, therefore, locally dependent, either on itself or on some other connection. A local dependence results in a *local guarantee*. Dependences in general and monotonicity in particular form the basis of the algorithm, which takes local guarantees and combines and propagates them to produce *transitive guarantees*.

2.2 Forward and backward processing

Dead timestamp identification algorithm has two components: forward and backward processing. The input to this algorithm is the application specified task graph (such as the one in Figure 3) that gives the connectivity among threads and channels, along with the associated monotonicity and dependence properties of the connections. Forward processing at a node N computes the forward guarantee as the set of timestamps that are likely to leave N. Similarly, backward processing at a node N computes the backward guarantee as the set

of timestamps that are dead so far as N is concerned. Dependences in general, and monotonicity in particular, are the basis for these guarantees. These properties allow associating a *timestamp marker* on each connection that separates good (higher) timestamps from bad (equal or lower) ones. Forward processing and backward processing algorithms use these markers available locally at each node on the connections that are incident at that node to generate the guarantees. These algorithms execute at runtime at the time of item transfers. Thus, the process of updating of the guarantees is associated with the flow of items through the system. In particular, as a timestamped item is transferred from node N1 to node N2, we update the forward guarantee at node N2 and the backward guarantee at node N1. This enables continual and aggressive identification of dead timestamps.

2.2.1 Forward processing

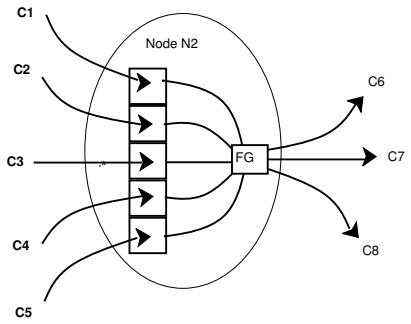

Figure 5: **ForwardGuaranteeVec**

Figure 5 provides an example for the components involved in this processing In this example, node N2 has input connections, C1-C5 and output connections C6-C8. Each node maintains a vector of forward guarantees ($ForwardGuaranteeVec$). There is a slot in this vector for each input connection, in this case C1-C5. Slot C_i of the vector holds the last forward guarantee communicated to the node over C_i. These are simply the timestamp markers associated with these connections. Forward processing at a node N involves computing the MIN of the elements of this vector and maintaining it as the $ForwardGuarantee$ for this node N, labeled FG in the figure.

2.2.2 Backward processing

Figure 6 provides an example for the components involved in this processing In this example, node N1 has input connections, C6-C8 and output connections C1-C5. Each node, maintains a vector of backward guarantees ($BackGuaranteeVec$). There is a slot in this vector for each output connection, in this case C1-C5. Slot C_i of the vector holds the last backward guarantee communicated to the node over C_i. These are once again the timestamp markers associated with these connections. Backward processing at a node N involves computing the MIN of the elements of this vector and maintaining it as the $BackwardGuarantee$ for this node N, labeled BG in the figure.

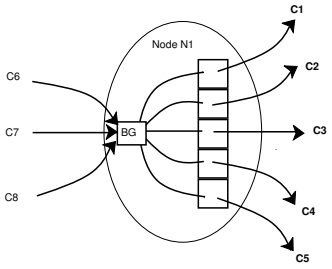

Figure 6: **BackwardGuaranteeVec**

BackwardGuarantee for node N identifies dead timestamps for that node. If the node is a channel, items in the channel with timestamps that are dead can be removed as garbage. Timestamps that arrive at a channel where they have been previously determined to be dead are *dead on arrival* and need not be placed in the channel. If the node is a thread, dead timestamps that have not yet been computed by that thread are dead computations and need not be computed.

2.3 Optimization

Sections 2.1 and 2.2 provide a basic framework for identifying dead timestamps. In this subsection we will present an optimization to this basic framework.

2.3.1 Transfer Functions

The basic framework uses the application specified task graph and the properties of the connections to generate the forward and backward guarantees. We can go further and use additional knowledge about the application to more aggressively expose dead timestamps. For example, it is conceivable that not all input connections to a thread node play a role in determining the timestamps on one of its output connection. If this application knowledge were to be made available to the forward and backward processing algorithms, then the guarantees produced would be more optimistic.

The machinery used to capture this application knowledge is *Transfer functions*. A forward transfer function is defined for each "out" connection from a node, and a backward transfer connection is defined for each "in" connection to a node. $\mathcal{T}_f$ and $\mathcal{T}_b$ indicate the forward and backward transfer functions respectively. $\mathcal{T}_f(\mathrm{C}_{out}) = \{C1_{in}, C2_{in}, \ldots Cn_{in}\}$ where node N is the (unique) source of the output connection C_{out} and $\{C1_{in}, C2_{in}, \ldots Cn_{in}\}$ is a subset of the input connections of N such that the timestamps put to C_{out} are determined only by the connections in this set. A connection, C_i for example, might not be in this set if C_i is a dependent connection or if timestamps for some output connection other than C_{out} are determined by C_i. $\mathcal{T}_b(\mathrm{C}_{in}) = \{C1, C2, \ldots Cn\}$ where node N is the (unique) target of the input connection C_{in} and $\{C1, C2, \ldots Cn\}$ is a subset of the input and output connections of N such that relevant timestamps for N are determined only by connections in this set.

For a thread node, the forward and backward transfer functions for connections incident at that node are determined by the thread code itself (and assumed to be made available in some form to the runtime system). For a channel node, the forward transfer function for any "out" connection is the set of all input connections; the backward transfer function for any "in" connection is the set of all input and output connections incident at that channel node.

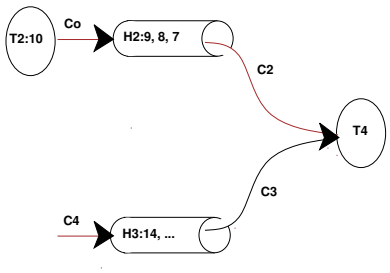

Figure 7: **Dead timestamp elimination example**

These transfer functions are used by the forward and backward processing algorithms to generate tighter bounds for dead timestamps. This is illustrated via an example. In Figure 4, assume that input connection C2 depends on C3. Thus $C3 \in \mathcal{T}_b(C2)$, but $C2 \notin \mathcal{T}_b(C3)$. Let T4 get the latest timestamp from C3 (say this is t); it then executes a *get* from C2 for the same timestamp t. Figure 7 shows a dynamic state of Figure 4. The highest timestamp on channel H3 is 14. H2 contains timestamps 7, 8 and 9. T2 is about to compute timestamp 10. When T4 gets timestamp 14 from C3 it will then wait for timestamp 14 from C2. The backward transfer function will help backward processing at node T4 to compute the backward guarantee on C2 as 14, thus allowing H2 to eliminate timestamps less than 14 as garbage (*i.e.,* timestamps 7, 8, and 9); this in turn will tell T2 to eliminate as dead computations, thread steps that produce timestamps 10, 11, 12 and 13.

The backward and forward processing algorithms incorporate these transfer functions and are summarized in Figure 8.

3 Implementation Issues

Stampede was originally implemented (Compaq CRL) on a cluster of 4-way Alpha SMPs interconnected by Memory Channel and running Tru64 Unix. Since then it has been ported to clusters of x86-Linux, x86-Solaris, StrongArm-Linux, and x86-NT boxes. Stampede facilitates the creation of any number of address spaces in each node of the cluster, and threads within each address space. The channels can be created in any of the address spaces and have system-wide unique IDs allowing transparent access to them by a thread running anywhere in the cluster. This implementation allows an item to be flagged as garbage in channels either via an explicit reference count associated with it, or via a transparent garbage collection

At a Transfer across a Connection C from Node N1 to Node N2
(a *put* from thread N1 to channel N2 or a *get* from channel N1 to thread N2)

Forward Processing:

- Considering transfer functions
  $ForwardGuaranteeVec_{N2}[C] = \min_{C_{in} \in \mathcal{T}_f(C)} ForwardGuaranteeVec_{N1}[C_{in}]$.
- Considering monotonicity
  $ForwardGuaranteeVec_{N2}[C] =$
  $\max(\min_{C_{in} \in \mathcal{T}_f(C)} ForwardGuaranteeVec_{N1}[C_{in}], current_C)$
  where $current_C$ is the latest (largest) timestamp to cross monotonic connection C.

Backward Processing:

- Considering local guarantees at connection C.
  $BackwardGuarantee^C_{N2} =$
  $\max(MonoGuarantee^{C\prime}_{N2}, ForwardGuaranteeVec_{N2}[C\prime])$
- Considering transitive guarantees across C.
  $BackwardGuarantee^C_{N2} = \min_{C_{out} \in \mathcal{T}_b(C)} backwardGuaranteeVec_{N2}[C_{out}]$
  where C_{out} are output connections from N2.
- Considering both local and transitive guarantees.
  $BackwardGuarantee^C_{N2} = \max(CurrentMax^{C\prime}_{N2}, CurrentMin^C_{N2})$

Figure 8: **Forward and Backward Processing Algorithms**

algorithm.

We have completed implementation of the dead timestamp identification algorithm described in the earlier section. This new implementation allows a node (which can be a channel or a thread) to propagate timestamp values of interest forward and backward through the dataflow graph (of channels and threads) that represents the application. The new implementation assumes that the application dataflow graph is fully specified at application startup (*i.e.,* static).

Forward propagation is instigated by the runtime system upon a put/get operation on a channel. For example, when a thread does a put on a channel, a lowerbound value for timestamps that that thread is likely to generate in the future is enclosed by the runtime system and sent to the channel. Similarly upon a get from a channel, the runtime system calculates a lowerbound for timestamp values that could possibly appear in that channel and piggybacks that value on the response sent to the thread.

Backward propagation is similarly instigated by put/get operations. In fact, backward propagation is likely to be more beneficial in terms of performance due to the properties of monotonicity and dependence on other connections which we described in Section 2.1. These properties come into play during a get operation on a channel. We have extended the Stampede API to enable a thread to enquire the forward and backward guarantees so that it may incorporate these guarantees in its computation.

There is very minimal application level burden to use the extended implementation of Stampede. Specifically, the application has to provide a few handler codes that the runtime system can call during execution to determine the forward and backward transfer functions for a given connection, the monotonicity and the dependence (if any) of a given connection on other ones.

Compared to the original implementation the new one offers two specific avenues for performance enhancement. First it provides a unified framework for both eliminating unnecessary computation from the thread nodes and the unnecessary items from the channel nodes as compared to the old one which does only the latter. Secondly, the new one allows getting rid of items from the channels more aggressively compared to the old one using the application level guarantees of monotonicity and dependence for a connection.

4 Performance Results

The Stampede cluster system supports three different garbage collection strategies: a simple reference count based garbage collector (REF), a transparent garbage collector (TGC), and the new dead timestamps based garbage collector (DGC). In REF, an application thread explicitly encodes the reference count when it does a *put* operation. The item is garbage collected when the reference count goes to zero. In TGC, the runtime system computes a global virtual time (GVT) using a distributed algorithm [8], which runs concurrent with the application. Subsequently, in each node of the cluster all items with timestamps lower than GVT are garbage collected. The GVT value thus computed is necessarily a safe lower bound for timestamps not needed by any thread. Clearly, REF is the most aggressive in terms of eliminating garbage as soon as it is recognized, while TGC is the most conservative. Neither REF nor TGC offer any help for removing dead computations. DGC is intended to

help eliminate both dead computations and dead items. However, in this study we show the relative performance of the three techniques with respect to garbage collection only.

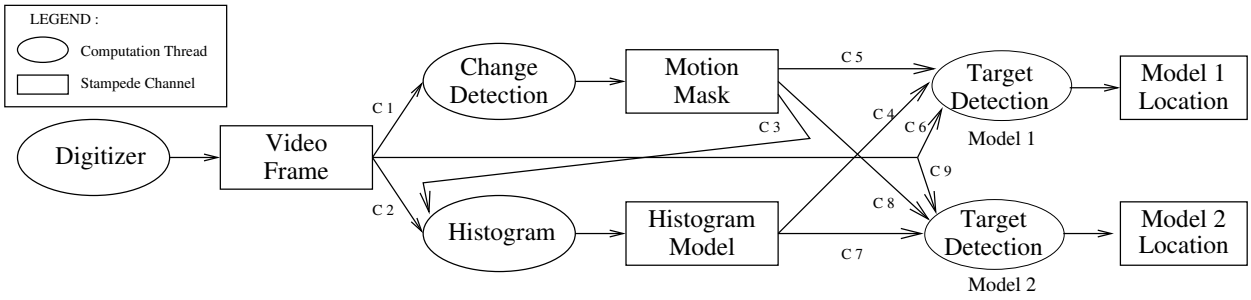

Figure 9: **Color Tracker Task Graph**

Connection	*Dependent Connections*
$C3$	$C2$
$C4$	$C5,\ C6$
$C7$	$C8,\ C9$

Figure 10: **Input Connection Dependency (monotonic).** These are the monotonic dependencies between the input connections of the tracker application illustrated in figure 9.

We use a real-time color-based people tracker application developed at Compaq CRL [12] for this study. Given a color histogram of a model to look for in a scene, this application locates the model if present. The application task graph and its connection dependencies are shown in Figures 9 and 10 respectively. As we mentioned earlier in Section 2, these connection dependencies are provided by the application and used by the dead timestamp identification algorithm to compute the forward and backward guarantees. Any number of models can be tracked simultaneously by cloning the target detection thread shown in the figure and giving each thread a distinct model to look for in the scene. The digitizer produces a new image every 30 milliseconds, giving each image a timestamp equal to the current frame number, The target detection algorithm cannot process at the rate at which the digitizer produces images. Thus not every image produced by the digitizer makes its way through the entire pipeline. At every stage of the pipeline, the threads get the latest available timestamp from their respective input channels. To enable a fair comparison across the three GC algorithms, the digitizer reads a pre-recorded set of images from a file; two target detection threads are used in each experiment; and the same model file is supplied to both the threads. Under the workload described above, the average message sizes delivered to the digitizer, motion mask, histogram, and target detection channels are 756,088, 252,080, 1,004,904, and 67 bytes respectively.

We have developed an elaborate measurement infrastructure that helps us to accumulate the memory usage as a function of time in the Stampede channels, and the latency for Stampede put/get/consume operations during the execution of the application. A post-mortem analysis program generates metrics of interest. Details of this measurement infrastructure are outside the scope of this paper.

The metrics for evaluating the three different strategies are the following: *memory footprint*, *space-time*, *currency*, and *latency per relevant timestamp*. Memory footprint is the amount of memory used by the application as a function of real time. This metric is indicative of the instantaneous memory pressure of the application. Space-time is the product of the memory usage and time for which a particular chunk of memory is used. This can be considered as the integral of the memory footprint over time. Currency is the value of relevant timestamp (that made its way through the entire pipeline) as a function of real time. This metric is indicative of the real-time performance of the application. The higher the currency value for a given real-time the better the performance. Latency is the elapsed time for a relevant timestamp to make it through the entire pipeline. This metric is the determinant of the real-time performance of the application.

The experiments are carried out on a cluster of 17, 8-way 550 MHz P-III Xeon SMP machines with 4GB of main memory running Redhat Linux 7.1. The interconnect is Gigabit Ethernet. The Stampede runtime uses a reliable messaging layer called CLF implemented on top of UDP. We use two configurations. In one configuration all the threads and channels shown in Figure 9 execute on one node within a single address space. This configuration represents one extreme, in which all computation is mapped onto a single node, and does not require the run-time system to use the messaging layer. In the second configuration the threads and channels are distributed over 5 nodes of the cluster. This configuration represents the other extreme, where threads and channels do not share the same address space. In this scenario, the messaging layer (CLF), as well as the physical network latencies, come into play. CPU resources, however, are not shared.

Figures 11 and 12 show latency per processed timestamp reaching the end of the application pipeline. Although the latency has increased for DGC due to inline execution of transfer functions on puts and gets, the percentage increase is only marginal (2.7% and 0.5% compared to TGC, 3.2% and less than 0.1% compared to REF for 1-node and 5-node configurations respectively). However, the memory footprint of the application as shown in Figure 13 is very much in favor of DGC. Furthermore, Figures 11 and 12 show a low mean and standard deviation for memory usage compared to both TGC and REF. The low memory usage compared to TGC is expected due to the aggressive nature of our algorithm. However, the performance advantage compared to REF is quite interesting. REF makes local decisions on items in a channel once the consumers have explicitly signaled a set of items to be garbage. DGC has an added advantage over REF in that it propagates guarantees to upstream channels thus enabling dead timestamps to be identified much earlier, resulting in a smaller footprint compared to REF. This trend can also be seen in the space-time metric column of Figures 11 and 12.

Figure 14 shows the currency metric for the three GC algorithms, for the first (1-node) configuration. The y-axis is the value of the timestamp that reaches the end of the pipeline and the x-axis is the real time. The higher the currency value for a given real time the better, since this is indicative of how recent the processed information is with respect to real time. The dead-timestamp based GC algorithm gives almost the same currency (Figure 14) despite the small increase in latency we observed earlier (average latency column in Figures 11 and 12). The currency metric results for the second (5-node) configuration is almost indistinguishable for the three algorithms and hence not shown in the paper. An interesting question for future work is investigating how the three algorithms behave in a

$Config\ 1:$ 1 *node*	*Total frames*	*Timestamps successful (relevent)*	*Average Latency (ms)*	*Mean memory usage (kB)*	*Memory usage STD*	*Total space − time usage (kB ∗ ms)*
DGC	4802	584	505,594	16,913	17,439	2,380,869,326
TGC	4801	600	491,946	24,043	24,787	3,402,019,615
REF	4802	595	489,610	23,755	24,606	3,229,459,927

Figure 11: **Metrics (1-node).** Performance of the three GC algorithms for the tracker application with all the threads executing within a single address space on one node. All experiments were run for the same period of time. Transparent GC (TGC) and Reference Counting (REF) on average consume around 40% more memory than dead-timestamps based GC (DGC). The space-time usage of TGC is 42.9% and that of REF is 35.6% greater than DGC. On the other hand, DGC is 2.7% and 3.2% slower in terms of average latency than TGC and REF, repectively.

$Config\ 2:$ 5 *nodes*	*Total frames*	*Timestamps successful (relevent)*	*Average Latency (ms)*	*Mean memory usage (kB)*	*Memory usage STD*	*Total space − time usage (kB ∗ ms)*
DGC	5509	599	557,502	28,096	28,194	4,842,436,910
TGC	5489	600	554,584	36,911	37,381	6,276,379,274
REF	5510	599	556,964	32,764	33,111	5,606,728,841

Figure 12: **Metrics (5-node).** Performance of three GC algorithms for the tracker application with the threads distributed on 5 nodes of the cluster. All configurations were run for the same period of time. Transparent GC (TGC) and Reference Counting (REF) on average consume respectively 31.4% and 16.6% more memory than dead-timestamps based GC (DGC). The space-time usage of Transparent GC is 29.6% and that of REF is 15.8% greater than DGC. On the other hand, the average latency of DGC is only 0.5% and 0.1% slower than that of TGC and REF, repectively.

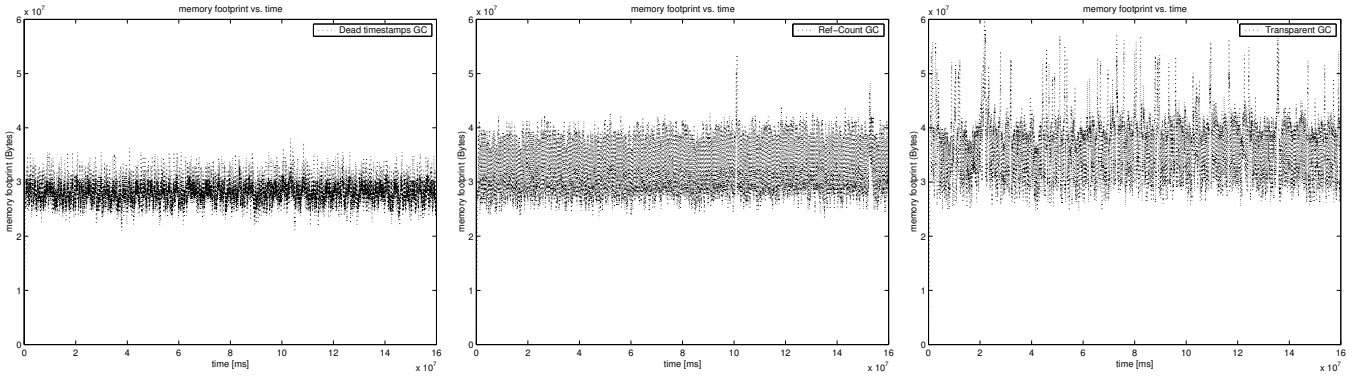

Figure 13: **Memory Footprint.** The three graphs represent the memory footprint of the application (distributed over 5 nodes) for the three GC algorithms: DGC-Dead timestamps (left), REF-Reference Counting (center), and TGC-Transparent (right). We recorded the amount of memory the application uses on every allocation and deallocation. All three graphs are to the same scale, with the y-axis showing memory use (bytes x 10^7), and the x-axis representing time (milliseconds). The graphs clearly show that DGC has a lower memory footprint than the other two. In addition, it deviates much less from the mean, thereby requiring a smaller amount of memory during peak usage.

resource constrained environment.

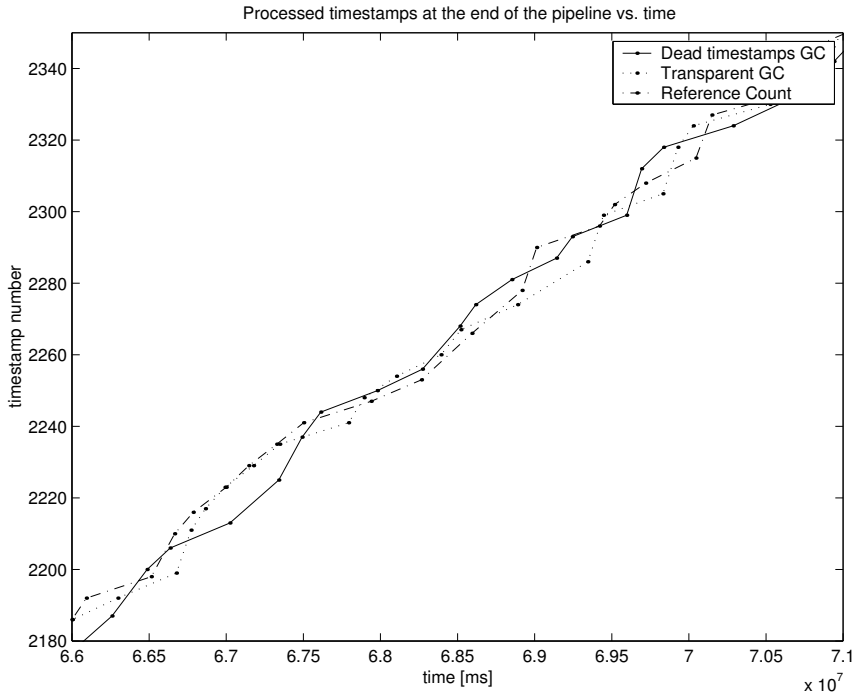

Figure 14: **Currency of Processed Timestamps (1-node).** There is no substantial difference in the currency of processed timestamps using any of the three GC algorithms.

We noted earlier that the new GC algorithm can aid both in eliminating dead items from channels and dead computations from threads. Clearly, dead computations can only result if later stages of the pipeline indicate during execution their lack of interest for some timestamp values to earlier stages. Thus for dead computation elimination one or more of the following conditions need to hold: (1) variability in processing times for items, (2) higher processing time for at least one earlier stage of the pipeline, (3) dependences on connections that change with time, and (4) variability in resource availability over time. The first three are properties of the application and workload, while the fourth is a property of the computational infrastructure. These properties do not hold in the application, the workload, and the hardware environment used in this study. We are currently exploring possible scenarios for illustrating the performance advantage of dead computation elimination.

5 Related Work

The traditional GC problem (on which there is a large body of literature [13, 6]) concerns reclaiming storage for heap-allocated objects (data structures) when they are no longer "reachable" from the computation. The "name" of an object is a heap address, *i.e.,* a pointer, and GC concerns a transitive computation that locates all objects that are reachable starting with names in certain well-known places such as registers and stacks. In most safe GC languages, there are no computational operations to generate new names (such as pointer arithmetic) other than the allocation of a new object. Stampede's GC problem is

an orthogonal problem. The "name" of an object in a channel is its timestamp, *i.e.,* the timestamp is an index or a tag. Timestamps are simply integers, and threads can compute new timestamps.

The problem of determining the interest set for timestamp values in Stampede has similarity to the garbage collection problem in Parallel Discrete Event Simulation (PDES) systems [3]. However, the application model that Stampede run-time system supports is less restrictive. Unlike Stampede, PDES systems require that repeated executions of an application program using the same input data and parameters produce the same results [4]. To ensure this property, *every* timestamp must *appear* to be processed *in order* by the PDES system. A number of synchronization algorithms have been proposed in the PDES literature to preserve this property. Algorithms such as Chandy-Misra-Bryant (CMB) [1, 2] process the timestamps strictly in order, exchanging null messages to avoid potential deadlocks. There is no reliance on any global mechanism or control. Optimistic algorithms, such as Time Warp [5], assume that processing a timestamp out of order by a node is safe. However, if this assumption proves false then the node rolls back to the state prior to processing the timestamp. To support such roll back, the system has to keep around *state*, which is reclaimed based on calculation of a Global Virtual Time (GVT). The tradeoff between the conservative (CMB) and optimistic (Time Warp) algorithms is space versus time. While the former is frugal with space at the expense of time, the latter does the opposite. These systems do not have the notion of dead computation elimination (because every timestamp must be processed).

On the other hand, the Stampede programming model does not require in-order execution of timestamps, nor does it require that every timestamp be processed. Consequently, Stampede does not have to support roll backs. If nothing is known about the application task graph, then similar to PDES, there is a necessity in Stampede to compute GVT to enable garbage collection. The less restrictive nature of the Stampede programming model allows conception of different types of algorithms for GVT calculation. Our earlier work [8] described algorithms for such transparent garbage collection. In this paper, we developed the algorithmic machinery to enable garbage collection based entirely on local events with no reliance on any global mechanism. This is somewhat akin to the approach taken by the (CMB) algorithm for PDES. Yet, we show in Section 4, that our algorithm is almost as good in terms of latency as our earlier GVT based transparent garbage collection, while accruing all the benefits of space reduction. The key to the success of this new algorithm is access to an application level task graph, as well as properties of the connections.

Dead code elimination is a common optimization technique in compilers for high level languages. However, we are not aware of any other work that provides a unified framework and implementation for eliminating dead computations and dead items at runtime.

6 Conclusions

Stampede is a cluster programming system for interactive stream-oriented applications such as vision and speech. Space management (in the form of eliminating unnecessary items) and time management (in the form of eliminating unnecessary computations) are crucial to enhance the performance of such applications. Stampede provides threads and channels as

computational abstractions for mapping the dataflow graph of the application to the cluster. In this paper, we have proposed a novel unified framework for dynamically eliminating both dead computations and dead items from such a computational pipeline. The framework defines a simple and intuitive machinery for applications to specify the properties of the computational pipeline. This information is used by the runtime to dynamically generate guarantees (lower bounds on timestamp values) to the threads and channels, that are then used in the dynamic elimination of dead items and computations. Stampede system has been implemented and runs on several different cluster platforms. Experimental results (on a cluster of a 17 node, 8-way 550 MHz P-III Xeon SMP-Linux interconnected by Gigabit Ethernet) are presented comparing the new algorithm with the previously available techniques for garbage collection in Stampede. The results show that, depending on the mapping of the threads and channels to the nodes of the cluster, the memory footprint of the new algorithm for a color-based vision tracker application is reduced anywhere from 16% to 40% compared to previous techniques. Future work includes more elaborate experimentation to illustrate the dead computation elimination capabilities of the unified framework.

7 Acknowledgments

A number of people have contributed to the Stampede project. Rishiyur Nikhil, and Jim Rehg contributed to the space-time memory abstraction that is at the heart of Stampede. The color-based tracker used in the performance study of this paper was developed by Jim Rehg. In addition, Bert Halstead, Chris Jeorg, Leonidas Kontothanassis, and Jamey Hicks contributed during the early stages of the Stampede project. Dave Panariti developed a version of CLF that runs transparently on Alpha Tru64 and Microsoft NT. All the members of the "ubiquitous presence" group at Georgia Tech continue to contribute to the Stampede project. Sameer Adhikari, Arnab Paul, Bikash Agarwalla, Matt Wolenetz, Phil Hutto, Durga Devi Mannaru, Russ Keldorph, Anand Lakshminarayanan, Namgeun Jeong, Yavor Angelov, and Ansley Post deserve special mention.

References

[1] R. E. Bryant. Simulation of Packet Communication Architecture Computer Systems. Technical Report MIT-LCS-TR-188, M.I.T, Cambridge, MA, 1977.

[2] K. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computation. *Communications of the ACM*, 24:198–206, 1981.

[3] R. M. Fujimoto. Parallel Discrete Event Simulation. *Comm. of the ACM*, 33(10), October 1990.

[4] R. M. Fujimoto. Parallel and distributed simulation. In *Winter Simulation Conference*, pages 118–125, December 1995.

[5] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[6] R. Jones and R. Lins. *Garbage Collection : Algorithms for Automatic Dynamic Memory Management*. John Wiley, August 1996. ISBN: 0471941484.

[7] K. Knobe, J. M. Rehg, A. Chauhan, R. S. Nikhil, and U. Ramachandran. Scheduling constrained dynamic applications on clusters. In *Proc. SC99: High Performance Networking and Computing Conf*, Portland, OR, November 1999. Technical paper.

[8] R. S. Nikhil and U. Ramachandran. Garbage Collection of Timestamped Data in Stampede. In *Proc.Nineteenth Annual Symposium on Principles of Distributed Computing (PODC 2000), Portland, Oregon*, July 2000.

[9] R. S. Nikhil, U. Ramachandran, J. M. Rehg, R. H. Halstead, Jr., C. F. Joerg, and L. Kontothanassis. Stampede: A programming system for emerging scalable interactive multimedia applications. In *Proc. Eleventh Intl. Wkshp. on Languages and Compilers for Parallel Computing (LCPC 98), Chapel Hill NC*, August 7-9 1998.

[10] U. Ramachandran, R. S. Nikhil, N. Harel, J. M. Rehg, and K. Knobe. Space-Time Memory: A Parallel Programming Abstraction for Interactive Multimedia Applications. In *Proc. Principles and Practice of Parallel Programming (PPoPP'99), Atlanta GA*, May 1999.

[11] J. M. Rehg, K. Knobe, U. Ramachandran, R. S. Nikhil, and A. Chauhan. Integrated task and data parallel support for dynamic applications. *Scientific Programming*, 7(3-4):289–302, 1999. Invited paper, selected from 1998 Workshop on Languages, Compilers, and Run-Time Systems.

[12] J. M. Rehg, M. Loughlin, and K. Waters. Vision for a Smart Kiosk. In *Computer Vision and Pattern Recognition*, pages 690–696, San Juan, Puerto Rico, June 17–19 1997.

[13] P. R. Wilson. Uniprocessor garbage collection techniques, Yves Bekkers and Jacques Cohen (eds.). In *Intl. Wkshp. on Memory Management (IWMM 92), St. Malo, France*, pages 1–42, September 1992.