A Comparative Study of Stampede Garbage Collection Algorithms

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

Nissim Harel [*]
nissim@cc.gatech.edu

Kathleen Knobe [†]
knobe@hp.com

Umakishore Ramachandran [*]
rama@cc.gatech.edu

Abstract

Stampede is a parallel programming system to support interactive multimedia applications. The system maintains temporal causality in such streaming real-time applications via *channels* that buffer *timestamped* items. A Stampede application is a coarse-grain dataflow pipeline of these timestamped items. Not all timestamps are *relevant* for an application output due to the differential processing rates of pipeline stages. Therefore, *garbage collection (GC)* is crucial for Stampede runtime performance. Three GC algorithms are currently available in Stampede. In this paper, we ask the question how far off these algorithms are from an *ideal garbage collector*, one in which the memory usage is exactly equal to that which is required for buffering only the relevant timestamped items in channels. This oracle, while unimplementable, serves as an empirical lower-bound for memory usage. We then propose optimizations that will help us get closer to this lower-bound. Using an elaborate measurement and post-mortem analysis infrastructure in Stampede, we evaluate the performance potential for these optimizations. A color-based people tracking application is used for the performance evaluation. Our results show that these optimizations reduce the memory usage by over 60% for this application over the best GC algorithm available in Stampede.

1 Introduction

Emerging applications such as interactive vision, speech, and multimedia collaboration require the acquisition, processing, synthesis, and correlation (often *temporally*) of streaming data such as video and audio. Such applications are good candidates for the scalable parallelism available in clusters of SMPs. In a companion paper [10] we discuss the need for higher level data abstractions to match the unique characteristics of this class of applications. *Stampede* [9] is a parallel programming system for enabling the development 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*[1].

A Stampede computation with threads and channels is akin to a coarse-grain dataflow graph, wherein the nodes are threads and channels and the links are the connections among them. Threads can be created to run anywhere in the cluster. Channels can be created anywhere in the cluster and have cluster-wide unique names. 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. Every item on a channel is uniquely indexed by a *timestamp*. Typically in a Stampede computation, a thread will *get* an item with a particular timestamp from an input connection, perform some processing[2] 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 runtime system of Stampede takes

[*]College of Computing, Georgia Institute of Technology

[†]HP Labs - Cambridge Research Laboratory

[1]Stampede also provides another cluster-wide data abstraction called *queues*. Queues also hold timestamped data items and differ in some semantic properties from the channels. From the point of view of the focus of this paper these differences are immaterial and hence we will not mention them in the rest of the paper.

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

care of the synchronization and communication inherent in these operations, as well as managing the storage for items put into or gotten from the channels. 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.

In a Stampede computation, the creation of threads, channels, and connections to channels are all dynamic. Since the channels hold timestamped data there is an issue as to when to get rid of data from channels that are no longer needed by any thread (current and future). We refer to this issue as the *garbage collection* problem in Stampede. The traditional GC problem [12, 7] concerns reclaiming storage for heap-allocated objects (data structures) when they are no longer "reachable" from the computation. On the other hand, Stampede's GC problem deals with determining when timestamped items in channels can be reclaimed. The runtime system determines that a specific timestamp (which is not a memory pointer but an index or a tag) will not be used anymore. Thus storage associated with all items that are tagged with this timestamp can be reclaimed. Stampede prescribes a simple set of rules for timestamp values that can be associated with an item by a thread (or its children). Further it imposes a discipline of programming that requires each thread to mark an item (or a set of items) on a channel as garbage once the thread is finished using that item by issuing a *consume* operation for that item. Using information from all consuming threads, the runtime system discerns when items can be garbage collected from channels.

In a companion paper [8], we have presented a distributed garbage collection algorithm for Stampede that does not use any application-level knowledge. In a more recent paper [5], we have introduced a new algorithm, called *dead-timestamp based garbage collection (DGC)*, that uses an application-level task graph to make runtime decisions on the interest set for timestamps. We show that using such application-level knowledge can result in a significant space advantage (up to 40%) compared to the earlier algorithm that does not use such knowledge.

In this paper, we ask the question how far off are these GC algorithms from an *ideal*? We define ideal as the case where the memory usage is exactly that which is required for processing relevant timestamps. While implementation of such an oracle is infeasible, it nevertheless serves as an empirical lower-bound for memory usage. We then propose two optimizations to the DGC algorithm that are expected to get us closer to this lower-bound:

1. The first optimization is to instantly propagate globally, information on irrelevent timestamps, to the entire task graph.

2. The second optimization is to buffer only the most recent items in a given channel. This optimization gives the *producer* of items some measure of direct control over garbage collection. The intuition is that downstream computations in interactive applications need only the most recent items and will skip over earlier items.

The rest of the paper is organized as follows. We discuss related work in section 2. Section 3 presents a summary of algorithms that are currently available in Stampede for garbage collection. In section 4, we present the proposed enhancements to the DGC algorithm. The measurement infrastructure in Stampede that allows us to gather the performance data for comparing these algorithms are presented in section 5. We introduce the definition for Ideal GC in section 5.1. Moving on to section 6, we discuss the performance of the current algorithms as well as the proposed enhancements with respect to the ideal. We present concluding remarks in section 7.

2 Related Work

The traditional GC problem (on which there is a large body of literature [12, 7]) 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 [6], 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 a 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.

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 like the one described in [8]. In [5] we have proposed yet another algorithm that uses application-level knowledge enabling garbage collection based entirely on local events with no reliance on any global mechanism.

3 Algorithms for Garbage Collection In Stampede

There are three competing mechanisms for GC in Stampede: a *REFerence count based garbage collector (REF)*, a *Transparent Garbage Collector (TGC)* [8], and a *Dead timestamp based Garbage Collector (DGC)* [5]. Each algorithm represents a specific point in the tradeoff between information needed to be provided to the runtime system and the corresponding aggressiveness for eliminating garbage.

- **Reference Count Based Garbage Collector:**

  This is the simplest garbage collector. As the name suggests, a thread associates a reference count with an item when it *puts* it in a channel. The item is garbage collected when the reference count goes to zero. Clearly, this garbage collector can only work if the consumer set for an item is known *a priori*. Hence this algorithm works only for static graphs where all the connections are fully specified.

- **Transparent Garbage Collector:**

  The TGC algorithm [8] maintains two *state variables* on behalf of each thread. The first is the *thread virtual time*, and the second is the *thread keep time*. The former gives the thread an independent handle on timestamp values it can associate with an item it produces on its output connections. The latter is the minimum timestamp value of items that this thread is still interested in getting on its input connections. Thread keep time advances when an item is consumed, and thread virtual time advances when a thread performs a *set virtual time* operation. The minimum of these two state variables gives a lower bound for timestamp values that this thread is interested in. TGC is a distributed algorithm that computes a *Global Virtual Time (GVT)* that represents the minimum timestamp value that is of interest to any thread in the entire system. So long as threads advance their respective virtual times, and consume items on their respective input connections the GVT will keep advancing. All items with timestamps less than GVT are guaranteed not to be accessed by any thread and can therefore be safely garbage collected. This algorithm is entirely transparent to the specifics of the application, and hence needs no information about the application structure.

- **Dead Timestamps Based Garbage Collector**

  The DGC algorithm [5] is inspired by the observation that even in dynamic applications, it may be possible to discern all the *potential* channel, thread, and connection creations by analyzing the application code. Under such circumstances, it is possible for the application programmer to supply the runtime system with a task graph that is a maximal representation of the application dynamism. A task graph (see Figure 1) for a Stampede application is a bipartite directed graph with *nodes* that represent either *threads*, which perform a certain computation, or *channels*, which serve 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 DGC algorithm 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

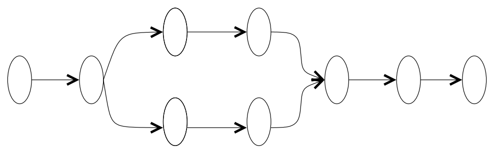


Figure 1: **An abstract task graph**

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. DGC use the application supplied task graph to propagate timestamp guarantees among the nodes. The 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. Similarly, the backward guarantee for a connection identifies timestamps that are dead on that connection. Using these guarantees, a node can *locally* separate live timestamps from dead ones, and garbage collect items with dead timestamps.

4 Enhancements to the DGC Algorithm

The three GC algorithms have been implemented in the Stampede system. In the case of DGC, the forward and backward propagation of timestamp guarantees are instigated by put/get operations on channels. In a companion paper [5], we experimentally compared the performance of these three GC algorithms in the context of a real-time color-based people tracker application that was developed at Compaq CRL [11]. We use *memory footprint* - the amount of memory used for buffering timestamped items in the Stampede channels by the application as a function of real time - as the metric for performance evaluation. This metric is indicative of the instantaneous memory pressure of the application. We showed that the memory footprint of the DGC algorithm for the color-based people tracker application is reduced anywhere from 16% to 40% compared to the other two algorithms.

In this section, we consider optimizations that can further reduce the memory footprint of the application. Although the first optimization is specific to DGC and concerns the global propagation of the timestamp guarantees, the second optimization is based on a more general technique (KLnU) that is applicable to any GC algorithm.

4.1 Out-of-Band Propagation of Guarantees (OBPG)

Our current implementation of DGC propagates forward and backward guarantees by piggy-backing them with the put/get operations. The limitation to this approach is that a node guarantee cannot be propagated along the graph until either (a) a *put* is performed by the thread node, or (b) a *get* is performed on the channel node. A more aggressive garbage collection would be possible if these guarantees are made instantly available to all nodes. It is conceivable to use *out of band* communication among nodes to disseminate these guarantees. However, there will be a consequent increase in runtime system overhead for such communication. To understand the above trade-off, the first optimization evaluates the following hypothetical question: *How can the memory footprint be reduced by instantly disseminating the node guarantees to all other nodes in the graph?* Clearly, instantaneous propagation is not practically feasible, nevertheless such inquiry helps us understand the potential performance limit for such an optimization.

4.2 Attributed Channels

- **Keep Latest ’n’ Unseen** The second optimization stems from the fact that in real-time applications, downstream computations are mostly interested only in the *latest* item produced by an upstream computation. Coupled with this fact is our earlier observation that upstream computations are lighter than the downstream ones, resulting in a large number of items becoming irrelevant. For example, in [5], we show that in the people tracker application only one in eight items produced by the digitizer thread reaches the end of the pipeline. The proposed optimization is to associate an *attribute* with a channel that allows it to discard all but the *latest* item. The basic idea is that when a producer puts a new item, the channel immediately gets rid of items with earlier timestamps if they have not been gotten on *any* of the connections. We refer to this set of earlier timestamps as the *dead set*. Since a Stampede application depends on timestamp causality, this optimization will not allow an item with an earlier timestamp to be garbage collected even if *one* connection has gotten that item from this channel. We generalize this attribute and call it *keep latest n unseen (KLnU)*, to signify that a channel keeps only the last n items. The value of n is specified at the time of channel creation and can be different for different channels. Further, it should be clear that this attribute gives local control for a channel to garbage collect items that are deemed irrelevant and is in addition to whatever system-wide mechanism may be in place for garbage collection (such as DGC).

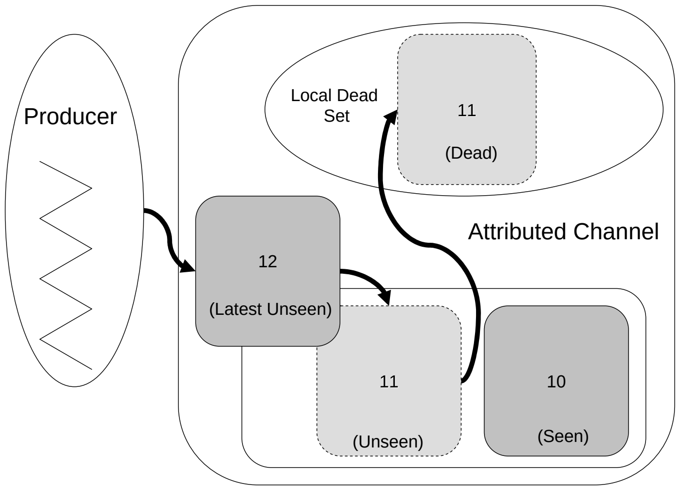
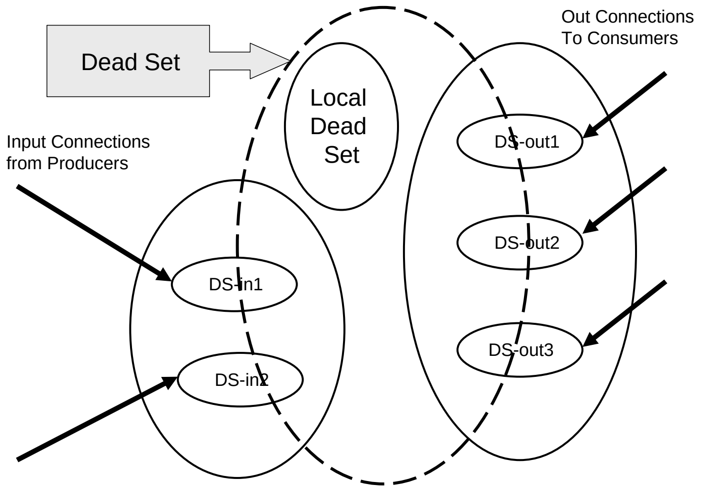


Figure 2: **An Attributed Channel.** Implementing the KLnU optimization where n = 1.

Figure 2 shows how attributed channels work. The producer thread has just produced an item with timestamp 12. Items with timestamps 10 and 11 are already present in the channel. Item with timestamp 10 has been gotten by a consumer, while 11 has not been gotten by any consumers so far. Thus upon put of item with timestamp 12, timestamp 11 can be added to the dead-set of this channel while 10 cannot be.

- **Propagating Dead Sets**

  Similar to the forward and backward guarantee propagation in the basic DGC algorithm, we consider propagating the *local* dead-set information from a node to other nodes in the graph. We call this optimization *Propagating Dead Sets (PDS)*.



Dead Set
Out Connections To Consumers
Local Dead Set
Input Connections from Producers
DS-out1
DS-out2
DS-out3
DS-in1
DS-in2

Figure 3: **Propagation of Dead Sets.** The arrows indicate the direction of flow of dead sets into the given node.

Figure 3 shows the state of a node in a given task graph. This node will receive dead-set information from all the out edges due to backward propagation. Similarly, it will receive dead-set information from all the in edges due to forward propagation. The dead-set for a node is computed as the *union* of the local dead-set and the *intersection* of the dead-set information of all the in and out edges incident on that node.

In [5], we introduced the notion of dependency among input connections incident at a node. This information allows timestamp guarantees to be *derived* for dependent connections. For e.g., if connection A to a channel is dependent on connection B to another channel, and if the guarantee on B is T (that is no timestamps *less than* T will be gotten on B), then the guarantee on A is T as well. In a similar vein, we propose dependency among output connections. For e.g., let A and B be out connections from a given channel, and let A be dependent on B. If the timestamp guarantee on B is T (that is no timestamps *less than* T will be gotten on B), then the guarantee on A is T as well.

Channel Name	*Out Connection*	*Dependent Connections*	*Channel Name*	*Out Connection*	*Dependent Connections*
Video Frame	$C1$	$C6, C9$	*Motion Mask*	$C3$	$C5, C8$
Video Frame	$C2$	*no dep.*	*Motion Mask*	$C5$	*no dep.*
Video Frame	$C6$	$C1$	*Motion Mask*	$C8$	*no dep.*
			Hist. Model	$C4$	*no dep.*
			Hist. Model	$C7$	*no dep.*

Figure 4: **Out Connection Dependency** The depencies stated here are for the real-time color people-tracker pipeline illustrated in figure 5

Figure 4 shows the connection dependency for the color-tracker pipeline (figure 5). As can be noticed from the table, for the video frame channel the out connections to the histogram and target-detection threads are dependent on the out connection to the change-detection thread. This dependency implies that timestamps in the dead-set of the change-detection thread, would never be gotten by either the histogram or the tracker threads. Therefore, the dead-set from the change-detection thread serves as a shorthand for the dead-sets of all three out connections emanating from the video frame channel. Thus the dependent connection information allows faster propagation of dead-set information to neighboring nodes during get/put operations.

- **Out-of-Band Propagation of Dead Sets (OBPDS)**

  The PDS optimization assumes that the dead-set information is propagated only upon get/put operations by neighboring nodes. In section 4.1 we observed the potential for more aggressive garbage collection if local information is disseminated more globally. This optimization investigates the performance potential for out-of-band communication of the the dead-set information to *all* nodes in the graph.

The KLnU attribute to a channel has applicability to any GC algorithm. However, the PDS, and OBPDS optimizations are specific enhancements to the DGC algorithm since they require application knowledge.

5 Methodology

Stampede is implemented as a runtime library on top of standard platforms (x86-Linux, Alpha-Tru64, x86-Solaris). The three GC algorithms mentioned in section 3 have been implemented in Stampede. The TGC algorithm is implemented in the Stampede runtime by a daemon thread in each address space that periodically wakes up and runs the distributed GVT calculation algorithm. As we mentioned earlier, for the DGC algorithm forward and backward propagations are instigated by the runtime system upon a put/get operation on a channel.

We have developed an elaborate measurement infrastructure that helps us to accumulate the memory usage as a function of time in the Stampede channels. Events of interest are logged at run-time in a pre-allocated memory buffer that is flushed to disk at the successful termination of the application. Some of the interesting events that are logged include, memory allocation times, channel put/get times, and memory free (or GC) times. A post-mortem analysis program then uses these logged events to generate metrics of interest such as the application's mean memory footprint, channel occupancy time for items, latency in processing etc.

5.1 Ideal GC

We define an *Ideal Garbage Collector (IGC)* as one in which the memory footprint of the application is exactly equal to that needed to buffer only *relevant* items of the application. Thus the memory footprint recorded by IGC represents a lower bound for an application.

5.2 Simulating IGC and the Proposed Optimizations

We use the runtime logs and post-mortem analysis technique to simulate IGC as well as answer the *what if* questions raised in section 4. This strategy is akin to using trace-driven simulation to predict the expected performance of an architectural idea in system studies. Essentially, we use the runtime logs as a trace. The events such as get/put/consume are all recorded in the logs with their respective times of occurrence. The logs contain the GC times for all item on all channels. However, either for simulating IGC or for studying the effect of the optimizations we simply ignore these log entries. Instead, using the logs and the specific optimization being investigated, the post-mortem analysis program *computes* the times when the items will be garbage collected in the presence of that optimization. For IGC we use the logs and postem-mortem analysis to accumulate the memory usage for only the relevant items.

6 Performance of GC Algorithms

We use a real-time color-based people tracker application developed at Compaq CRL [11] 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 is shown in figure 5.

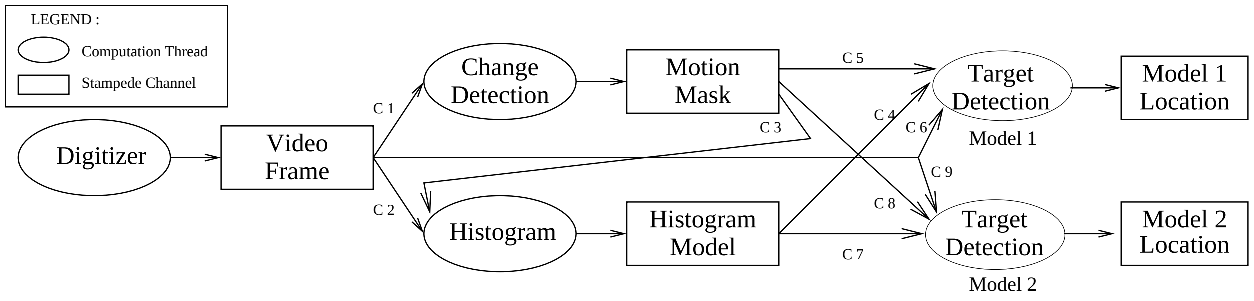


Figure 5: **Color Tracker Task Graph**

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. Each thread gets the latest available timestamp from its input channel. To enable a fair comparison for all the proposed optimization strategies, 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 756088, 252080, 1004904, and 67 bytes respectively.

We use *memory footprint* metric to evaluate the proposed optimization strategies. As we mentioned in section 4, memory footprint is the amount of memory used for buffering timestamped items in the channels by the application as a function of real time, and is indicative of the instantaneous memory pressure exerted by the application. To associate real numbers with the memory footprint, we also present the *mean memory usage* of the application.

All experiments are carried out on a cluster of 17, 8-way 550 MHz P-III Xeon SMP machines, each with 4GB of main memory running Redhat Linux 7.1 and interconnected with Gigabit Ethernet. The Stampede runtime uses a reliable messaging layer called CLF implemented on top of UDP. We conduct our experiments using two configurations. In the first configuration all threads and channels shown in figure 5 execute on one node within a single address space. Thus there is no need for inter-node communication in this configuration. In the second configuration, threads and channels are distributed over 5 nodes of the cluster. The channel into which a thread ‘puts’ items is colocated with that thread in the same address space. Due to the distribution of threads in different address spaces, the messaging layer as well as the network latencies play a part in determining the performance of the application. CPU resources, however, are not shared.

6.1 Performance of TGC, REF and DGC

In our earlier work [5], we presented a comparison of the DGC, REF, and TGC algorithms in the context of this same application. Figure 7 and figure 6 summarize the results of this comparison. The primary conclusion from that work is that although the latency for processing a relevant timestamp through the entire pipeline increased for DGC due to the in-line execution of transfer functions on puts and gets, the percentage increase was nominal (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 [5]). However, the mean memory usage and standard deviation for memory usage for DGC were both much lower that that of either TGC or REF. Figure 7 also shows the simulated results for IGC. The IGC result is really that of an oracle that knows the set of relevant timestamps exactly. It gives a lower bound for memory usage and serves to point out the disparity of realizable implementations (such as DGC) from the ideal. As can be seen, the mean memory usage of DGC (which is the best performer of the three GC techniques) is still 429% (on 1AS config.) with respect to that of IGC. This is the reason for exploring further optimization strategies.

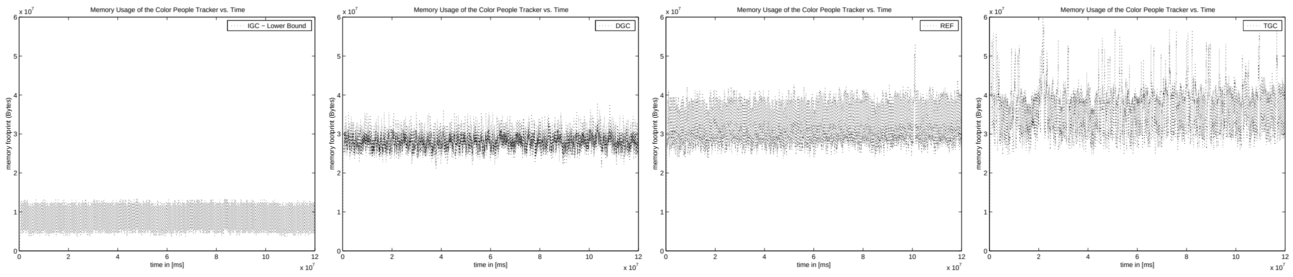


Figure 6: **Memory Footprint.** The four graphs represent the memory footprint of the application (distributed over 5 nodes) for the three GC algorithms and the additional Ideal: (left to right)(a) Ideal Garbage Collector (IGC), (b) DGC-Dead timestamps GC, (c) REF-Reference Counting, (d) TGC-Transparent. 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 both REF and TGG but still higher than IGC. In further comparison with REF and TGC, DGC deviates much less from the mean, thereby requiring a smaller amount of memory during peak usage.

	$Config\ 1:1\ node$				$Config\ 2:5\ nodes$			
	$Average$ $Latency$ (ms)	$Mean$ $Memory$ $usage\ (B)$	$Memory$ $usage$ STD	$\%$ $w.r.t.$ IGC	$Average$ $Latency$ (ms)	$Mean$ $Memory$ $usage\ (B)$	$Memory$ $usage$ STD	$\%$ $w.r.t.$ IGC
TGC	491,946	24,696,845	6,347,450	616	554,584	36,848,898	5,698,552	444
REF	489,610	23,431,677	6,247,977	585	556,964	32,916,702	4,745,092	397
DGC	505,594	17,196,808	4,143,659	429	557,502	28,118,615	2,247,225	339
IGC	N/A	4,006,947	1,016,427	100	N/A	8,292,963	2,903,339	100

Figure 7: **Metrics (1 and 5 node configurations).** Performance of three GC algorithms and IGC is given for the color people-tracker application. The fifth and the last column also give the percentage of *Mean Memory usage* with respect to IGC.

6.2 Performance of the Proposed Optimizations

In this subsection we consider the performance potential of each of the proposed optimizations. As we explained in section 4, the KLnU optimization has general applicability. However, in this study we use the DGC algorithm as the base for evaluating all the performance enhancements.

- **Performance of Out-of-Band Propagation of Guarantees (OBPG)**

  Figure 8 (c) shows the results for this optimization. As we mentioned earlier in section 4, to assess the performance limit for this optimization, zero time is assumed for the out-of-band dissemination of the timestamp guarantees. Comparing the results with those for DGC (figure 8 (b)), it can be seen that the peak memory usage is lower with this optimization. However, the difference is not significant. The reason is two-fold both of which stem from the nature of this particular application. First, later stages in the pipeline do more processing than earlier stages. Therefore, backward guarantees are more useful in hastening garbage collection than forward guarantees. Second, the task graph is highly *connected* (see figure 5). Later threads (such as the target detection thread) need input directly from earlier channels (such as the video frame channel). Thus GC events from later stages of the pipeline are directly getting fed back to earlier stages reducing the benefits of the OBPG optimization.

- **Performance of Keep Latest n Unseen (KLnU)**

  For this optimization, we associate the KLnU attribute (*n = 1*, i.e., a channel buffers only the most recent item) with all the channels in the color tracker pipeline. Figure 9 (a) and figure 10 show the effect of the KLnU optimization over DGC. Compared to DGC (unoptimized) there is only a modest (16% - 26% reduction in memory usage) improvement due to this optimization. This is surprising since we expected that this optimization will allow each channel to be more aggressive in eliminating garbage locally. Recall that even though the buffering is limited to just the most recent item, a channel still cannot GC earlier items that have been gotten by at least one out connection to preserve timestamp causality. The surprisingly small reduction in memory usage is a consequence of this fact.

- **Performance of Propagating Dead Sets (PDS)**

  Figure 9 (c) and (d) show the results of the PDS optimization. Compared to DGC (unoptimized) the reduction in memory usage is large (59% - 55%). The significant improvement due to this optimization is quite surprising at first glance. This optimization combines two effects: first, nodes propagate the dead-set information forwards and backwards using the application task graph; second, a channel aggressively incorporates the incoming dead-set information using the dependency information on its out connections. Analysis of our runtime logs reveal that it is the latter effect that aids the effectiveness of this optimization. For example, the video frame channel can use the dead-set information that it receives from the change-detection thread immediately without waiting for similar notification from the histogram and target-detection threads.

- **Performance of Out-of-Band Propagation of Dead Sets (OBPDS)**

  This optimization is similar to OBPG, with the difference that dead-set information is disseminated out-of-band instead of timestamp guarantees. As in OBPG, we assume that this dissemination itself takes zero time to assess the performance limit of this optimization. Figure 10 shows a small reduction (approximately 500KB) in memory usage compared to DGC (unoptimized). As we observed with the OBPG optimization, the relatively little impact of this optimization is due to the connectedness of this particular application task graph.

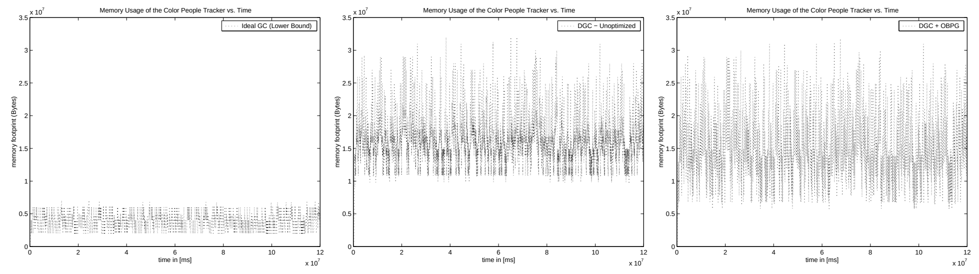


Figure 8: **Memory Footprint.** The tracker application was run on 1 Address Space (on a single node). The memory footprint graphs above show results for (from left to right) : (a) Ideal GC - Lower Bound (IGC), (b) the unoptimized DGC implementation, (c) DGC optimized with Out-of-Band Propagation of Guarantees (OBPG)

Overall, the combined effect of the proposed optimizations is a reduction in memory usage of 62% on 1 AS, and 65% on 5AS, respectively, compared to DGC (unoptimized).

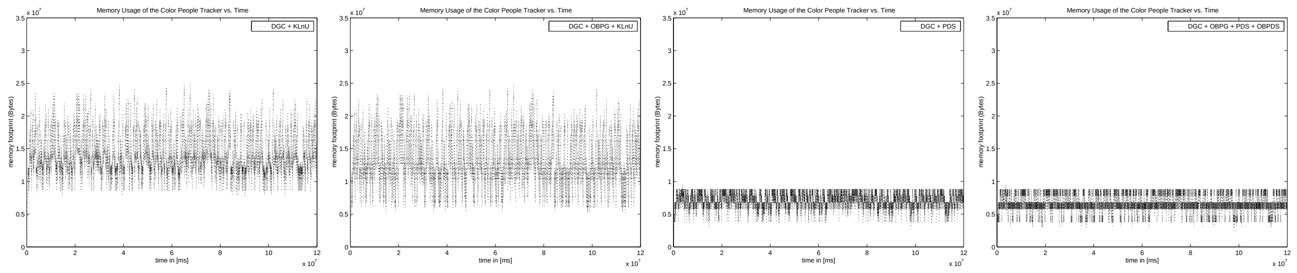


Figure 9: **Memory Footprint.** The tracker application was run again on 1 Address Space (on a single node). The memory footprint graphs above show results for: (from left to right) (a) DGC with KLnU , (b) DGC with OBPG and KLnU optimizations, (c) DGC with PDS, and (d) DGC with OBPG and OBPDS

	$Config\ 1 : 1\ node$				$Config\ 2 : 5\ nodes$			
	$Mean\ Memory\ usage\ (B)$	$Memory\ usage\ STD$	$\%\ w.r.t.\ DGC$	$\%\ w.r.t.\ IGC$	$Mean\ Memory\ usage\ (B)$	$Memory\ usage\ STD$	$\%\ w.r.t.\ DGC$	$\%\ w.r.t.\ IGC$
$DGC - Unoptimized$	17,196,808	4,143,659	100	429	30,359,907	2,486,790	100	366
$DGC + OBPG$	15,509,662	4,774,496	85	387	22,076,617	5,609,878	73	266
$DGC + KLnU$	14,480,981	3,222,947	84	361	22,500,410	2,181,028	74	271
$DGC + OBPG + KLnU$	13,201,657	3,764,269	77	329	16,456,518	4,396,111	54	198
$DGC + PDS$	7,016,269	981,811	41	175	13,770,987	2,910,143	45	166
$DGC + OBPG + OBPDS$	6,497,603	1,235,326	38	162	10,556,471	3,226,764	35	127
$IGC\ Lower - Bound$	4,006,947	1,016,427	23	100	8,292,963	2,903,339	27	100

Figure 10: **Metrics (1 and 5 node configurations).** Performance of different GC algorithms with combinations of different optimizations is presented for the tracker application. Percentage *Mean Memory usage* of optimizations with respect to that of Unoptimized DGC and IGC are also presented in the figure.

6.3 Summary

We considered a number of potential optimizations to enhance the performance of garbage collection. Of the ones considered, the PDS scheme is implementable while OBPG and OBPDS are mainly to understand the limits to performance of implementable strategies. The performance benefits of the PDS optimization is sensitive to several factors that are application-specific: number of attributed channels, the value of n for the KLnU attribute, the connectedness of the task graph, and the dependencies among the in/out connections to nodes in the graph.

Our performance study using the color-based people tracker application revealed some counter-intuitive results. First, disseminating the timestamp guarantee of DGC or the dead-set information of PDS to all nodes did not result in substantial savings. In hindsight, this seems reasonable given the connectedness of the tracker task graph. Second, the KLnU optimization in itself was not sufficient to get substantial reduction in memory usage. The propagation of the dead-set information, and use of the dependency information on the out connections of channels was the key to achieving most of the performance benefits.

There is a danger in generalizing the expected performance benefit of these optimizations simply based on the results of one application. Nevertheless, it appears that knowledge of the dependency information on the out connections of channels is a crucial determinant to the performance potential of these optimizations. A question worth investigating is the performance potential for incorporating out connection dependency in the original DGC algorithm. Another question worth investigating is the extent to which the TGC and REF algorithms will benefit in the presence of attributed channels.

7 Concluding Remarks

Stampede is a cluster programming system for interactive stream-oriented applications such as vision and speech. A Stampede application is composed of threads that can execute anywhere in the cluster, communicating timestamped items via channels (also allocable transparently cluster-wide). An important function performed by the Stampede runtime system is garbage collection of timestamped data items that are no longer needed by any threads in the application. In real-time interactive applications for which Stampede is targeted, it is common for threads to work with the most recent items in a channel and skip over earlier timestamps. It is therefore crucial for the performace of the Stampede runtime system, and the computation pipeline running above it, that garbage collection of such irrelevent (skipped over) items from channels occur efficiently.

Our earlier work has proposed a *distributed* transparent garbage collection algorithm. In a more recent work, we have proposed a new algorithm that uses an application task graph to *locally* compute guarantees on timestamps that are needed by a given node in the graph and propagate such guarantees to other nodes. In this paper, we have first quantified how far off are the memory usages in these existing algorithms from an *ideal garbage collector*, one that buffers exactly the items that are processed fully in the task graph. Armed with this knowledge, we propose optimizations that help us get closer to the empirical limit suggested by the ideal garbage collector. The first optimization is to make the timestamp guarantees of our previous algorithm available to all the nodes in the graph. The second optimization is to buffer only a few of the most recent items in each channel. This optimization gives local control to the producer of items to declare some timestamps as *dead* in a channel and allows dissemination of this information to other nodes in the graph. The proposed optimizations are evaluated with the measurement and post-mortem analysis infrastructure available in Stampede using a color-based people tracker application. The results show over 60% reduction in the memory usage compared to our most aggressive garbage collection algorithm that is based on timestamp guarantees.

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] N. Harel, H. A. Mandviwala, K. Knobe, and U. Ramachandran. Dead timestamp identification in stampede. In *The 2002 International Conference on Parallel Processing (ICPP-02)*, Aug. 2002. To Appear.

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

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

[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, 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.

[12] 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.