Garbage Collection of Timestamped Data in Stampede[*]

Rishiyur S. Nikhil
Compaq Computer Corporation
Cambridge Research Laboratory
One Kendall Square, Bldg. 700
Cambridge MA 02139, USA
nikhil@crl.dec.com

Umakishore Ramachandran
College of Computing
801 Atlantic Drive
Georgia Inst. of Technology
Atlanta GA 30332, USA
rama@cc.gatech.edu

ABSTRACT

Stampede is a parallel programming system to facilitate the programming of interactive multimedia applications on clusters of SMPs.

In a Stampede application, a variable number of threads can communicate data items to each other via *channels*, which are distributed, synchronized data structures containing timestamped data such as images from a video camera. Channels are not queue-like: threads may produce and consume items out of timestamp order; they may produce and consume items sparsely (skipping timestamps), and multiple threads (including newly created threads) may consume an item in a channel. These flexibilities are required due to the complex dynamic parallel structure of applications, to support increased parallelism, and because of real-time requirements.

Under these circumstances, a key issue is the "garbage collection condition": *When can an item in a channel be garbage collected?*

In this paper we specify precisely Stampede's semantics concerning timestamps, and we describe two associated garbage collection conditions– a weak condition, and a more expensive but stronger condition. We then describe a distributed, concurrent algorithm that implements these two GC conditions. We present performance numbers that show little or no application-level performance penalty to using this algorithm for aiding automatic garbage collection in a cluster. We conclude with some remarks about the implementation in the Stampede system.

1. INTRODUCTION

In this paper we describe an unusual garbage collection (GC) problem and its solution.

There is an emerging class of important applications: sophisticated interactive systems that use vision and speech recognition to comprehend a dynamically changing environment and react in real time. Broadly speaking, such a system is organized as a pipeline of tasks processing streams of data, for example starting with sequences of camera images, at each stage extracting higher and higher-level "features" and "events", and eventually responding with outputs. Fig. 1 illustrates such a system. The task structure is usually a dynamic graph, more complex than a simple linear pipeline, and the "streams" are quite unlike FIFO queues. Items in a stream may be produced and consumed by multiple tasks both because the functionality demands it and because tasks may be replicated for greater performance. Although items have a natural ordering based on "time", they need not be produced nor consumed in that order. Items may be produced and consumed sparsely (non-consecutive timestamps). The producing and consuming tasks of a stream may not be known statically, *e.g.,* in the figure the "High-fi" tracker task may only be created when the "Low-fi" task detects something, and more than one "High-fi" task may be created to track multiple targets. In such an environment, *garbage collection* is a tricky issue– when can the resources occupied by an item in a stream be safely reclaimed?

Stampede is a programming system designed and built at CRL to simplify the programming of such applications [5, 4]. A Stampede program consists of a dynamic collection of threads communicating timestamped items through *channels*. Stampede's rules governing time and timestamps admit all the flexibility described in the previous paragraph, but still capture a notion of an overall forward progress of time. In this context, it is possible to determine certain lower time bounds below which all items are guaranteed to be garbage.

This paper describes this system, the GC problem and its solution. We present related work in Sec. 2). In Sec. 3 we describe the computational system: threads, channels, connections, time, timestamps, and operations on these entities. We also describe the GC problem with respect to this system. In Sec. 4 we describe a simple "GC condition", *i.e.,* a predicate that identifies garbage, and discuss its correctness. In Sec. 5 we describe a second GC condition that is more complicated (and expensive), but which is strictly

[*]This work has been funded in part by NSF grants MIPS-9630145 and CCR-9972216, and a research gift from Compaq Cambridge Research Lab.

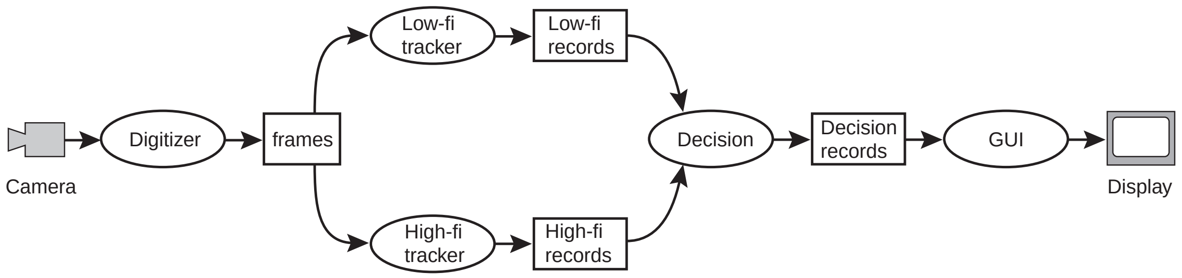

Figure 1: A simple vision pipeline (ovals are tasks, and rectangles are "streams" or "channels")

more powerful (identifies more garbage) and solves a certain "progress" problem associated with the first GC condition. In Sec. 6 we discuss complexity and implementation cost. In Sec. 7 we describe a distributed, concurrent algorithm that implements the two GC conditions. Implementation details, and some performance notes are presented in Sec. 8. Finally, we conclude in Sec. 9 with some remarks about future work.

2. RELATED WORK

The traditional GC problem (on which there is a large literature [7, 3]) 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 tag. Timestamps are simply integers, and threads can compute new timestamps.

Stampede's GC problem is somewhat analogous to the traditional problem of "black-holing space leaks". For example, consider a table (hash table, tree, association list, ...) that contains several objects. As long as the table itself is reachable, traditional GC will not reclaim any object in the table even if there are objects that will never be looked up in the future. "Black-holing" is an analysis and optimization that identifies such objects and reclaims them. Traditionally, black-holing is based on a flow analysis in the compiler. In Stampede, a channel is like a table, and the GC problem is to reason about time in order to identify items in the channel that will never be accessed in the future.

The terminology and the algorithms to be described in this paper are closely related to Global Virtual Time (GVT) calculation in Parallel Discrete Event Simulation [2]. Indeed, our distributed algorithms are inspired by knowledge of work in that area. However, there are many significant differences. PDES systems often work with static task graphs, and timestamps formally encode dependencies, so that tasks are required to process timestamps in timestamp order (optimistic systems have rollback/undo mechanisms if this is violated). In Stampede, task graphs can vary dynamically, timestamps may be consumed and produced sparsely, and timestamps may be consumed and produced out of order (the system does not associate correctness with this ordering).

A concrete context for Stampede would be an information kiosk in a public space such as an airport or shopping mall. Researchers at CRL have prototyped such kiosks and placed them in actual public environments [6, 1], and this is a major motivator and concrete testbed for our work. Similar requirements exist in other systems that use vision/speech and react in real time to a changing environment, such as autonomous vehicles and virtual-reality simulators.

3. THE COMPUTATIONAL MODEL AND THE GC PROBLEM

In this section we describe Stampede's computational model (the "mutator", in classical GC terminology) and its associated garbage collection problem.

3.1 Intuitions behind "time" and "visibility" in Stampede

The Stampede computational model consists mainly of a collection of threads communicating timestamped items via *channels*. Because items in channels are timestamped, we also refer to the collection of channels as *Space-Time Memory* [5]. Threads may be sources (they only produce items), intermediate (both consume and produce items), or sinks (only consume items). An example of a source thread is one that captures image frames from a video camera at regular intervals and puts them into a channel. An example of a sink thread is one that encapsulates the output display. Most threads in a vision system are intermediate threads.

We can think of timestamps as integers that represent discrete points in real time. For example, we may use the sequence numbers of images captured from a camera as timestamps to accompany those images. In intermediate threads we typically want to maintain this association of timestamps with data. For example, a thread may get an image-and-timestamp pair <x,ts> from a channel, perform some computation on it, and put some result <y,ts> into another channel, where y is a detected feature or an annotated image.

It greatly simplifies the programming of intermediate threads if they do not have to manage the progress of time explicitly, even though they may process timestamps out of order. In most cases, if a thread is currently getting items from one or more channels with timestamps ts_1, ts_2, ..., ts_n, then the corresponding item that it outputs typically has a timestamp that is simply derived from the input timestamps, such as picking one of the input timestamps. In the frequent special case of a thread with a single input, the next output timestamp is usually just the current input timestamp.

In general, we want "time" to march forward, *i.e.,* as a thread repeatedly gets increasing timestamps on its inputs, we want the outputs that it produces also to have increasing timestamps. To capture this idea we introduce the notion of the *visibility* for a thread: when a thread is processing inputs with timestamps ts_1, ts_2, ..., ts_n, it is only allowed to output items with timestamps $\geq$ the minimum of the input timestamps. A way to think about this is that the thread derives its "current time" from the timestamps of its current input items, and it is not allowed to output an item that is "earlier" than this.

There are a few more rules to capture this idea. When a thread creates a new thread, the new child thread should not thereby be able to observe or output items in the past w.r.t. the thread-creation event. When a thread attaches a new input connection to a channel it should not be able to observe items in the past w.r.t. the attachment event.

Although the above ideas allow automation of time management, there are a few situations where a thread needs to manage its notion of time explicitly. Source threads of course need to do this, since they have no input timestamps from which to derive their current time. But even an intermediate thread may wish to "reserve the right" to look at older data. For example, a thread may use simple differencing to identify regions in an image where some change has occurred from the previous image (*e.g.,* someone approaches the camera). When it detects a sufficiently interesting difference, it may re-analyze the interesting region of the last d images using a more sophisticated algorithm (*e.g.,* a face detector). Thus, the thread needs to have the ability to go d steps back in time.

The computational model in Stampede, in particular its rules for time and visibility, are motivated by these intuitions. The rules allow "out of order" processing. For example, for performance reasons a task may be parallelized into multiple threads that share common input and output channels. The rate at which these threads process inputs may be unpredictable, so that outputs may be produced out of timestamp order. A downstream thread should also be allowed to consume these results as they arrive, rather than in timestamp order.

3.2 The computation model

Fig. 2 depicts entities and operations in Stampede's computational model. It consists of dynamically created *threads* and *channels*. Input connections (*i_conns*) and output connections (*o_conns*) are explicit associations between channels and threads. A thread can have several i_conns and o_conns, including multiple connections to the same channel. We will use the notation "i_conns(t)" and "o_conns(t)" to refer to the current set of i_conns and o_conns, respectively, of the thread t.

Each thread t has a state variable $\mathrm{VT}(t) \geq 0$, the thread's *virtual time*. Virtual times, and timestamps in the following discussion, are just integers. A legal value for $\mathrm{VT}(t)$ is $+\infty$ (this is typically the case for intermediate threads that do not want to manage time explicitly).

A channel contains zero or more *timestamped* items, written <x,ts>. The exact nature of an item (some application data communicated among threads) is not relevant for this discussion and is depicted as an elliptical blob in the figure. Thus, instead of the more verbose "an item with timestamp ts" we will frequently just say "a timestamp ts". Threads *get* items from channels *via* i_conns, and *put* items into channels *via* o_conns.

In a channel, initially all timestamps are *absent*. When a thread successfully performs the following operation on an o_conn oc:

put(oc, x, ts)

the item at timestamp ts becomes *present* in the channel. The put() operation fails if the timestamp is already present; duplicate timestamps in a channel are not allowed (in this sense, a channel is just like a stream, where there would be a unique item at each time index). The put() operation may also fail because ts is an illegal timestamp; we will describe this situation under visibility rules (Sec. 3.2.1)).

While the *absent* and *present* states are associated with a timestamp in a channel and are independent of connections to threads, there is also a *per i_conn* view of items in a channel, with associated state (for which we use capital letters). When an item is *absent* in a channel, it is also ABSENT w.r.t. any i_conn to that channel. When a timestamp becomes *present* in an channel, it is initially UNSEEN w.r.t. any i_conn to that channel. A thread may perform the following operation over an i_conn ic:

<x,ts> = get (ic)

If successful, this returns an item and its associated timestamp; the item then becomes OPEN w.r.t. ic. Note, that a thread cannot get a timestamp more than once over ic (it can still get the same timestamp over a different i_conn). In the real Stampede system, get() has parameters that specify which item to return (specific timestamp, wildcards, LATEST, ...), but those details are unimportant for this discussion.

Finally, the thread can perform:

consume (ic, ts)

and the item becomes CONSUMED w.r.t. ic. In summary, in the per i_conn view of the channel, a timestamp transitions monotonically:

$$\text{ABSENT} \longrightarrow \text{UNSEEN} \longrightarrow \text{OPEN} \longrightarrow \text{CONSUMED}$$

There is one short-cut to the state transitions just described.

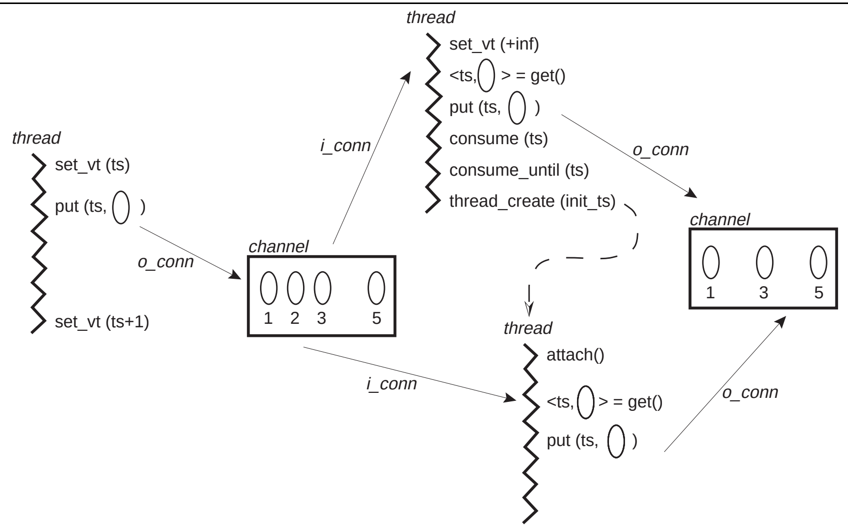

Figure 2: Entities and operations in Stampede's computational model

A thread can also perform:

consume_until (ic, ts)

in which case *all* timestamps $\leq$ ts immediately transition into the CONSUMED state w.r.t. ic, irrespective of their previous state. In particular, even ABSENT and UNSEEN items can thus be consumed, and this is effectively an assertion by the thread that it is skipping over such timestamps and will never perform a get() on any timestamp $\leq$ ts in the future (even if an *absent* item in the channel becomes *present* in the future).

Note that, unlike dequeueing an item from a queue, get() does not remove an item from a channel; it just returns the value (or a copy) of the item. The item can be got multiple times over multiple i_conns.

3.2.1 Visibility rules

We now discuss five "visibility" rules governing a thread's operations.

V1: A thread t can always get() *any* UNSEEN item on any of its i_conns, regardless of VT(t).

This is part of the ability to process items out of order. Next, we define the instantaneous visibility VIS(t) of a thread t as the min of VT(t) and the timestamps of any items it currently has OPEN on its i_conns.

$$\begin{aligned}\mathrm{VIS}(t) = \min\ (\ &\mathrm{VT}(t),\\ &\min\ [\ \mathrm{ts} \mid \mathrm{ic} \leftarrow \mathrm{i\_conns}(t),\\ &\qquad c = \mathrm{channel\_of(ic)},\\ &\qquad \mathrm{x} \leftarrow \mathrm{items}(c),\\ &\qquad \mathrm{state(x,ic)} = \mathrm{OPEN}\ ]\ )\end{aligned}$$

Conceptually, this is the "current time" at which the thread is currently working. Again, because of out-of-order processing, VIS(t) need not increase monotonically. A thread can get an item on an i_conn with timestamp ts < VIS(t) (because of *V1*); in this case, VIS(t) will decrease to ts.

V2: The put() operation in an o_conn oc is limited by visibility (cannot put an item "in the past"):

put(oc, x, ts) *provided* ts $\geq$ VIS(t)
timestamps < VIS(t) are considered
illegal for put() operation on o_conn oc

V3: A thread t can change its own virtual time VT(t) to any time in its current visibility:

set_vt(vt) *provided* vt $\geq$ VIS(t)

i.e., the thread can only move its VT(t) forward w.r.t. VIS(t). Note that VT(t) can decrease, since VIS(t) may be lower than VT(t) due to a currently OPEN item with a lower timestamp.

V4: When a thread t creates a new thread t', its initial virtual time is bounded below by VIS(t):

thread_create(vt') $\rightarrow t'$ *provided* vt' $\geq$ VIS(t)
Then, VT(t') is initialized to vt'

V5: When a thread t creates a new i_conn ic to an existing channel c, the timestamps it may get over ic are limited by VIS(t):

attach(c) $\rightarrow$ ic
The new i_conn is initialized with an implicit
consume_until(ic, VIS(t)-1)

Rules *V4* and *V5* ensure that a child thread can never perform operations "in the past" w.r.t. the parent thread.

3.3 The GC problem

At first glance, the consume() operation may seem odd in a discussion of automatic garbage collection. In traditional GC there is no analogous operation. However, remember that, here, items are "named" by integer timestamps on which the program can do arbitrary arithmetic. Thus, even with a strong type system that distinguishes timestamps from integers, it would be impossible to examine the state of the computation and automatically determine the set of unnameable timestamps. (This is why traditional safe GC'd languages disallow arbitrary pointer arithmetic.) Thus, it is necessary to have a consume() operation by which a thread declares that it has finished examining the result of a previous get(); it is not a deallocation operation.

Conceptually, each <x,ts> entry, once put into a channel, remains forever. However, visibility rules may imply that a particular timestamp can never be got in the future by any thread over any i_conn. If so, the resources occupied by such an item can be reclaimed. The Stampede GC problem, therefore, is:

> Identify all timestamps in all channels that no thread can possibly get() in the future (and recycle the resources occupied by those items).

Why is this non-trivial? When a thread performs consume(ic,ts), this says nothing about items with lower timestamps, since out-of-order get's are allowed. When an item in a channel has been consumed over all i_conns to that channel, it may still not be garbage because a thread may attach a new i_conn to the channel over which the item is still UNSEEN, and can therefore be got. If a timestamp is absent from a channel, it could be that the producer of that item just hasn't got around to producing it yet, but it could also be that no thread is capable any longer of producing that item because of the visibility rules. Thus, it requires some non-local reasoning, taking into account the current state of all the threads and connections, to determine that a particular timestamp in a channel is garbage.

Note that when an item is GC'd, its timestamp does not revert to *absent* in the channel; it is still conceptually *present*, *i.e.,* no thread can do a put() operation with that timestamp on this channel. In practice we do not need to retain the state associated with GC'd items because visibility rules preclude any attempt to access (for either put() or get()) any item with a GC'd timestamp.

4. MINKV: A GC CONDITION BASED ON VIRTUAL AND KEEP TIMES

A "GC Condition" is a predicate that identifies garbage, *i.e.,* when applied to an item, it specifies whether or not that item is garbage. Of course, a GC condition must be *safe* (when it says it is garbage, it really is garbage), but it is usually conservative (when it says it is not garbage, what it means is that we don't know). We initially assume that the predicate examines the *instantaneous entire state of the system* while it is not in the middle of any Stampede operation. We can always force the system into such a state by not starting any new Stampede operation and allowing those already in progress to complete. For those Stampede operations that have already been initiated and that might block within the operation (certain variants of get() and put()), we can model them using polling loops, and then just wait until a thread is at a neutral point in the polling loop. This assumption can be directly translated into a "stop the world" operation in the Stampede runtime system.

Define the auxiliary concept of KT(ic) (*keep time*) of an i_conn ic as the smallest timestamp in the associated channel such that all lower timestamps have been *explicitly* CONSUMED w.r.t. ic, using consume() or consume_until() calls:

$$\mathrm{KT(ic)} = \mathrm{ts}_k \quad \text{s.t. } \mathrm{ts}_k \text{ is not CONSUMED w.r.t. ic } (\textit{i.e.,} \text{ is ABSENT/UNSEEN/OPEN}), \text{ and } \forall\ 0 \le \mathrm{ts} < \mathrm{ts}_k, \text{ ts is CONSUMED w.r.t. ic}$$

Some observations:

- A thread can no longer get a timestamp $<$ KT(ic) over the i_conn ic.
- Since the (per i_conn) state of an item monotonically advances up to the CONSUMED state, this implies that KT(ic) can only monotonically increase.
- Executing consume_until(ic,ts) implies that KT(ic) $\ge$ ts+1 (after this operation KT(ic) can be strictly $>$ ts+1 if the thread has already consumed items at ts+1, ts+2, ...)
- If the thread has skipped over a timestamp (it is ABSENT/UNSEEN), KT(ic) cannot advance past that timestamp unless the thread performs consume_until().

The minKV GC condition is based on the minimum of thread virtual times and i_conn keep times:

Theorem 4.0 (minKV condition):
An item with timestamp $<$ minKV is garbage, where

$$\begin{aligned} \mathrm{minV} &= \min\ [\ \mathrm{VT}(t) \mid t \leftarrow \text{all threads}\ ] \\ \mathrm{minK} &= \min\ [\ \mathrm{KT(ic)} \mid t \leftarrow \text{all threads}, \\ &\qquad\quad \mathrm{ic} \leftarrow \mathrm{i\_conns}(t)\ ] \\ \mathrm{minKV} &= \min\ (\mathrm{minK}, \mathrm{minV}) \end{aligned}$$

Central idea in correctness proof

Essentially formalizes and details the intuitions described above. minKV $\le$ VIS(t) for all threads t. VIS(t) is a lower bound on any puts and gets that can be performed in the future by t or any of its descendants due to thread creation, *unless* there is or appears an UNSEEN item in one of their i_conns. But there can be no UNSEEN items below minKV. □

5. MINO: A GC CONDITION BASED ON OBSERVABLE TIMES

There is a problem with the "progress" of the minKV condition in the presence of sparse production and consumption of items, illustrated in Figure 3. In the i_conn view ic, KT(ic) = 2 because of the ABSENT item at timestamp 2. This will hold minKV to 2, and no items at 2 or above can be GC'd. However, if we examine the global state of the system (suggested by the rest of the figure), we can see that it is impossible for any thread ever to put an item at timestamp 2 into that channel. All threads have higher VT's, and there

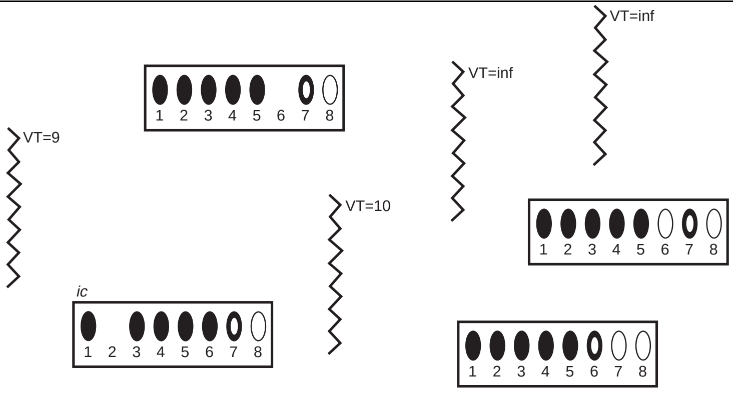

Figure 3: A problem with minKV (missing timestamps are ABSENT; white items are UNSEEN; donuts are OPEN; black items are CONSUMED)

is no UNSEEN item in any channel that will allow their VIS values to drop to 2. Thus, it is safe to consider timestamp 2 as if it were CONSUMED everywhere. In fact, it is safe to regard all timestamps below 6 as if they are CONSUMED.

The minO GC condition formalizes this (the name is derived from the idea that the minimum "observable" timestamp may be higher than minKV). Define an entry <x,ts> to be *fully consumed in channel* c if it has been consumed on all i_conns to the channel, *i.e.,*

$\forall$ ic $\leftarrow$ i_conns(c), <x,ts> is CONSUMED w.r.t. ic

The minO condition is:

Theorem 5.0 (minO condition):

```
An item with timestamp < minO is garbage
where
  if (minV ≤ minK)
  then minO = minK
  else minO is the smallest timestamp s.t.
    minK ≤ minO ≤ minV, and
    minO is not fully consumed in some channel, and
    ∀ ts s.t. minK < ts < minO,
    ∀ c ← all channels,
      either ts is absent in c,
      or ts is fully consumed in c
```

Sketch of correctness proof

Basically formalizes the reasoning given in the example above. □

6. COMPLEXITY AND COST OF IMPLEMENTATION

The complexity of computing the minKV condition is:

O(#threads + #i_conns $\times a$)
where,
a = Average # items in a channel

because we need the VT of each thread and the KT of each i_conn, which in turn requires examining the per i_conn state of all the items in the channel. In our implementation we maintain the KT of each i_conn incrementally, *i.e.,* we update KT(ic) every time we do a consume() or consume_until() on ic. Since KT's have already been computed, the actual work done during the computation of minKV is:

O(#threads + #i_conns)

The complexity of computing the minO condition is:

O(#threads + #i_conns + $b \times c$)
where,
b = Average # items between minKT and minVT
c = Average # i_conns per channel

because we need the VT of each thread and the KT of each i_conn, and we need to check the "consumed-ness" of each item between minKT and minVT in each channel, w.r.t. all i_conns to that channel.

The minO condition is more expensive. Further, minO is only useful in the presence of some sparse production and consumption patterns. In many applications, even if puts and gets are sparse, using consume_until() instead of consume() fills any gaps in the consumption pattern, allowing minKV to advance past absent items, so that all garbage is caught by the minKV condition. The strategy in our Stampede implementation (see Sec. 8) is to run the minKV computation frequently, and the minO computation less frequently.

7. A DISTRIBUTED, CONCURRENT ALGORITHM FOR GC CONDITIONS

It is trivial to implement a "stop the world" algorithm for minKV and minO computations from the above specifications of the GC conditions. We now describe a distributed and concurrent algorithm for computing these conditions.

By "distributed" we mean that each thread and each channel belongs to one of N *Address Spaces* (AS's) which are

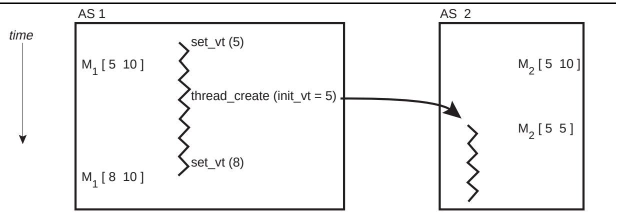

Figure 4: Non-monotonicity of ASminKV(j)

identified by integers 1..N. They may actually be implemented in a distributed manner; this is just a definitional convenience indicating that each thread's/channel's state is available in one AS. We will focus only on the minKV and minO computations; the exact details of how channels are accessed across address spaces, or how threads are forked across address spaces, are not relevant for the present discussion. Suffice it to say that threads in an AS can transparently access channels across address spaces. An address space represents a memory protection boundary. Thus to achieve address space transparency for channels, messages have to exchanged among address spaces. Such low-level messages are simply vehicles for implementing the Stampede computation model and are invisible to application threads executing within the confines of the Stampede computation model. These messages are asynchronous and have unknown latencies. However, we assume that messages between a particular pair of address spaces are delivered in order.

By "concurrent" we mean that, unlike a "stop the world" algorithm, the application computation (the "mutator") may proceed in parallel with the computation of the minKV and minO values, and the subsequent reclamation of storage associated with the items that can be considered garbage (the "collector").

7.1 A Distributed, Concurrent Algorithm for Computing minKV

Define ASminKV(j) as the minimum of thread virtual times and i_conn keep times, restricted to threads on address space j and their i_conns:

```
ASminKV ($j$) = min (ASminK, ASminV)
where,
    ASminV = min [ VT($t$) | $t \leftarrow$ threads_of_AS($j$) ]
    ASminK = min [ KT(ic) | $t \leftarrow$ threads_of_AS($j$),
                            ic $\leftarrow$ i_conns($t$) ]
```

Each AS can compute its own ASminKV. Then, minKV is simply the minimum of all the individual ASminKV's, but of course there is no instantaneous way to do this distributed computation.

Let us suppose that ASminKV(j) increases monotonically (in fact it does, *almost*; we will address the exception shortly). Then, it is easy for each AS to maintain a conservative approximation of minKV: The key insight for the correctness of this algorithm is that the array M_j at address space AS j is always a conservative snapshot of the ASminKV's of all the address spaces. Armed with this insight, it is fairly straightforward to prove that assuming that each ASminKV(j) increases monotonically, then this approxMinKV is a conservative approximation to ($<$ than) the real minKV, and therefore timestamps below this value can be garbage collected safely.

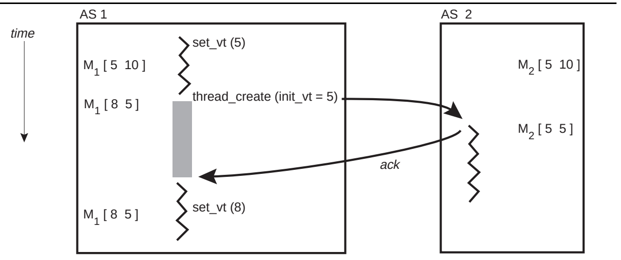

Figure 5: Solution to the problem of non-monotonicity of ASminKV(j)

```
Distributed Computation of approxMinKV:
  On each AS $j$
    Create an array $M_j$[1..N].
    Initialize $M_j$[1..N] with initial VT
          (usually 0) of the application.
    Loop forever:
    Compute $M_j[j]$ = ASminKV ($j$)
    Send array $M_j$ to all remaining AS's
    Upon receipt of array $M_i$ from each
          remaining AS $i$,
          Incorporate $M_i$ into $M_j$ as follows:
              $M_j[i] = M_i[i]$,
                    (believe his knowledge of himself)
              Ignore $M_i[j]$
                    (ignore his estimate of me)
              $\forall k, k \neq j, k \neq i$
              $M_j[k]$ = min($M_j[k]$, $M_i[k]$)
                    (unify our estimates of others)
    approxMinKV = min ($M_j$[1..N])
```

Unfortunately, there is one situation where ASminKV(j) can decrease. This is illustrated in Fig. 4. Suppose AS 1 has a thread that has set its VT to 5, and suppose ASminKV(1)=5. Suppose AS 2 has ASminKV(2)=10. And, suppose that the M[] arrays on both AS's are in sync, both therefore having the values M[1]=5 and M[2]=10. Now suppose the thread on AS 1 creates a thread on AS 2 with initial VT=5. On AS 2, M_2[2] will decrease to 5 (this is the non-monotonic change). Then, before AS 2 communicates the new M_2[] to AS 1, let the thread on AS 1 increase its VT to 8, so that ASminKV(1) advances to 8, so that M_1[1] advances to 8. Now we have a problem– the minimum of array M_1[] on AS 1 is now 8, and we would erroneously treat everything below 8 as garbage. But, the new thread on AS 2 could attach to any channel on AS 1 and legally access timestamps 5, 6, and 7, *i.e.,* they are not garbage.

The solution to this problem is illustrated in Figure 5. As soon as the thread on AS 1 does a remote thread_create(), we set M_1[2] to the minimum of its previous value and the initial

VT of the new thread (here, that number is 5). We also wait to receive an acknowledgment of the thread creation from AS 2 (in Stampede, we wait for an ack anyway because remote thread creation can fail and we must wait for this status; other threads on AS 1 may continue running). This time interval is depicted by the shaded box in the figure. In this interval, we *discard* any arriving messages from AS 2 carrying $M_2[]$ (for example, such a message, sent just before the thread creation, may indicate that we should increase $M_1[2]$ back to 10). However, because of message-ordering, any $M_2[]$ message from AS 2 that arrives after the ack is guaranteed to incorporate the initial VT of the new thread. Thus, $M_1[]$ and $M_2[]$ together safely accommodate the non-monotonicity.

Formally, we need a small modification to the Stampede runtime system, and a small modification to the minKV computation:

```
On AS $j$
  Runtime system:
    Initialize an array of integers flag[1..N] to zero.
    When a thread attempts to create a thread on
        AS($i$) with initial VT ts,
      Increment flag[$i$]
      $M_j[i]$ = min ($M_j[i]$, ts)
      Send the request message to AS($i$)
      Receive the response message (ack) from AS($i$)
      Decrement flag[$i$]

  Computation of MinKV
    ... modified so that when flag[$i$] > 0, ignore $M_i$
        msgs from AS $i$ ...
```

The protocol of incrementing and decrementing flag[i] is needed to take care of the situation where multiple threads in one address space may simultaneously perform remote create_thread operation. We have only conveyed the flavor of the solution above, omitting many synchronization details for brevity.

7.2 A Distributed, Concurrent Algorithm for Computing minO

The strategy here is similar to the minKV case. We denote the minO computation restricted to channels on AS j by ASminO(j). This number also increases monotonically, except for two situations:

- When a thread does a put() into a remote channel
- When a thread does an attach() creating a new i_conn to a remote channel.

These potential non-monotonicities are treated just like the potential non-monotonicity during remote thread creation in the minKV case.

7.3 Garbage Collection

These algorithms basically compute a *lower bound* for timestamp values that is valid in the entire system, with minKV giving a more conservative lower bound compared to minO. GC itself is trivial once this lower bound is computed. Simply reclaim all channel items whose timestamps are lower than this lower bound. This step is entirely a local operation to each address space. Note that both the distributed algorithm for determining the lower bound as well as the actual reclamation of channel items proceed concurrently with execution of the application program.

8. IMPLEMENTATION, STATUS, AND PERFORMANCE NOTES

Stampede was originally implemented on a cluster of 4-processor 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. To enable the automatic garbage collection, a *daemon* thread is created in each address space. The daemon periodically wakes up (a tunable parameter) and executes the distributed concurrent algorithms sketched in the previous section to compute minKV and minO values. Basically, the daemon in each address space computes the local values of minKV and minO, and sends a message to all the other address spaces enclosing the newly computed values. There is no need to synchronize the execution of these daemons in the different address spaces for this computation. Each address space can then determine the global value of minKV and minO from the local calculation and the messages it receives from its peers as described in the earlier section.

How well do the algorithms presented in the previous section perform? To answer this question, we conduct the following experiment: A driver thread (D) continually puts items into a channel (d), and an echo thread (E) gets and consumes these items from the channel. The echo thread continually puts a response message in a channel (e), which is gotten and consumed by the driver thread. The above ping-pong loop is executed a large number of times. The channels have a fixed capacity (100 items in this experiment). In one set-up, we GC an item as soon as it is consumed, and call this set-up reference-count based GC. Clearly this reference-count based set-up needs no distributed coordination to determine when and what items become garbage. In the second set-up, we let our distributed algorithm determine when and what items can be GC'ed through the minKV and minO computations. The metric for comparison between these two set-ups is the average round-trip time observed for the ping-pong loop. In other words, we want to see if the distributed GC leads to an increase in the execution time of the loop. The results are summarized in Fig. 6 for three different configurations of the driver and echo threads, and the channels. The results (comparing the last two columns in Fig. 6) show that the there is hardly any difference in round-trip times for the ping-pong loop for the two set-ups. This is because the actual work associated with GC is done by a daemon thread in each address space running concurrently with the application threads, and the cluster nodes are 4-way SMP boxes.

9. CONCLUSION

Config #	Configuration	Datasize (bytes)	Normalized Round-trip time for ping-pong loop	
			Ref-count GC (%)	Distributed GC (%)
1	Driver and Echo threads on one node and the channels on a second node	128	100	100.9
		4096	100	95.4
2	Driver and its output channel on one node Echo and and its output channel on a second node	128	100	96.3
		4096	100	99.3
3	Driver, Echo, and both channels all on the same node	128	100	101
		4096	100	104

Figure 6: Round-trip time for ping-pong loop test. The measured times for the distributed garbage collector are normalized with respect to those observed for the reference-count based garbage collector.

Stampede is a parallel programming system aimed at simplifying the programming of interactive applications that use vision and speech. This is an increasingly important class of applications that are computationally demanding and dynamic and complex in structure. Within this context there is an interesting and unusual GC problem, orthogonal to the traditional GC problem. In this paper we have outlined this problem, described declarative conditions to identify garbage, and described a distributed, concurrent algorithm for computing these conditions. This algorithm has been implemented in Stampede, and the Stampede system itself is in use at CRL, Georgia Tech, and Rice University.

The algorithms in this paper (and in our implementation) involve looking at all threads, all i_conns and all channels to calculate global minimums which are then used uniformly for GC on all channels. Our colleague K. Knobe has observed that we can exploit the "threads-and-channels graph" to segment these analyses, limiting them to smaller regions. The threads-and-channels graph is analogous to the traditional procedure call-graph in compilers, *i.e.,* it is a graph which conservatively estimates which threads can create which other threads and which threads can attach to which channels, based on a static analysis of the application code. For example, if this graph indicates that a certain region of the graph R_1 is always "upstream" from another region R_2, then we can independently identify and collect garbage in R_1 without considering R_2. This idea is to be explored further in our future work. Another area of future work is consider communication link and node failures in implementing the Stampede computation model.

10. REFERENCES

[1] A. D. Christian and B. L. Avery. Digital Smart Kiosk Project. In *ACM SIGCHI '98*, pages 155–162, Los Angeles, CA, April 18–23 1998.

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

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.