Space-Time Memory: A Parallel Programming Abstraction for Interactive Multimedia Applications

Umakishore Ramachandran[1] Rishiyur S. Nikhil[2] Nissim Harel[1] James M. Rehg[2] Kathleen Knobe[2]

[1] College of Computing, Georgia Insitute of Technology
[2] Compaq Computer Corporation, Cambridge Research Laboratory (CRL)

Abstract

Realistic interactive multimedia involving vision, animation, and multimedia collaboration is likely to become an important aspect of future computer applications. The scalable parallelism inherent in such applications coupled with their computational demands make them ideal candidates for SMPs and clusters of SMPs. These applications have novel requirements that offer new kinds of challenges for parallel system design.

We have designed a programming system called *Stampede* that offers many functionalities needed to simplify development of such applications (such as high-level data sharing abstractions, dynamic cluster-wide threads, and multiple address spaces). We have built Stampede and it runs on clusters of SMPs. To date we have implemented two applications on Stampede, one of which is discussed herein.

In this paper we describe a part of Stampede called *Space-Time Memory* (STM). It is a novel data sharing abstraction that enables interactive multimedia applications to manage a collection of time-sequenced data items simply, efficiently, and transparently across a cluster. STM relieves the application programmer from low level synchronization and data communication by providing a high level interface that subsumes buffer management, inter-thread synchronization, and location transparency for data produced and accessed anywhere in the cluster. STM also automatically handles garbage collection of data items that will no longer be accessed by any of the application threads. We discuss ease of use issues for developing applications using STM, and present preliminary performance results to show that STM's overhead is low.

1 Introduction

Emerging application domains such as interactive vision, animation, and multimedia collaboration display dynamic scalable parallelism. Due to their high computational requirements, they are good candidates for executing on parallel architectures. SMPs and clusters of SMPs are attractive platforms for exploiting the inherent scalable parallelism of such applications. There are some aspects of these applications that set them apart from scientific applications that have been the main target of high performance parallel computing in recent years. First, time is an important attribute in such emerging applications due to their interactive nature. In particular, they require the efficient management of temporally evolving data. For example, a stereo module in an interactive vision application may require images 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. Second, 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.

To address these problems we have developed an abstraction for parallel programming called **Space-Time memory (STM)** – a dynamic concurrent distributed data structure for holding time-sequenced data. STM addresses the common parallel programming requirements found in most interactive applications, namely, buffer management, inter-task synchronization, and meeting soft real-time constraints. These facilities are useful for this application class even on an SMP. However, in addition, our system provides the STM abstraction transparently across clusters. Currently, STM runs on a cluster of Alpha SMPs (running Digital Unix) interconnected by Memory Channel. We have used STM to implement the vision tracking component of an interactive multimedia application called the *Smart Kiosk*.

The key contributions of this paper are:

- the presentation of the STM abstraction for parallel programming, and its implementation;
- a demonstration of ease of use, using this abstraction for programming interactive multimedia applications, and

College of Computing, Georgia Institute of Technology, Atlanta GA 30332, USA; {rama,nissim}@cc.gatech.edu
Compaq Computer Corporation, Cambridge Research Laboratory, One Kendall Square, Bldg 700, Cambridge MA 02139, USA; {nikhil,rehg,knobe}@crl.dec.com

- a preliminary performance study using this abstraction on a cluster of SMPs. In particular, we show that STM's significant programming advantage (over, say, direct message-passing) incurs only low performance overheads.

We begin by giving the application context, in Sec. 2. In Sec. 3, we enumerate the parallel programming requirements engendered by interactive multimedia applications. The Space-Time Memory abstraction, and the unusual garbage collection problem in this class of applications are discussed in Sec. 4. Ease of use of STM is demonstrated via programming examples in Sec. 5. Subsequently, we discuss design rationales (Sec. 6) and present related work (Sec. 7). Preliminary performance results of STM are presented in Sec. 8 and concluding remarks are in Sec. 9.

2 Application Context

The *Smart Kiosk* is a new type of public computer device under development at Compaq's Cambridge Research Lab [22, 4]. It is located in public spaces such as a store, museum, or airport and is designed to interact with multiple people in a natural, intuitive fashion. For example, we envision Smart Kiosks that entertain passers-by while providing directions and information on local events. The kiosk may initiate contact with customers, greeting them when they approach and acknowledging their departure.

A Smart Kiosk may employ a variety of input and output devices for human-centered interaction: video cameras, microphones, infrared and ultrasonic sensors, loudspeakers, and touch screens. Computer vision techniques are used to track, identify and recognize one or more customers in the scene [17]. A future kiosk will use microphone arrays to acquire speech input from customers, and will recognize customer gestures. Synthetic emotive speaking faces [21] and sophisticated graphics, in addition to Web-based information displays, are currently used for the kiosk's responses.

We believe that the Smart Kiosk has features that are typical of many emerging scalable applications, including mobile robots, smart vehicles, intelligent rooms, and interactive animation. These applications all have advanced input/output modes (such as computer vision), very computationally demanding components with dynamic structure, and real-time constraints because they interact with the real world.

Fig. 1 shows the software architecture of a Smart Kiosk. The input analysis hierarchy attempts to understand the environment immediately in front of the kiosk. At the lowest level, sensors provide regularly-paced streams of data, such as images at 30 frames per second from a camera. In the quiescent state, a blob tracker does simple repetitive image-differencing to detect activity in the field of view. When such an activity is detected, a color tracker can be initiated that checks the color histogram of the interesting region of the image, to refine the hypothesis that an interesting object (*e.g.,* a human) is in view. If successful, this in turn can invoke higher-level analyzers to detect faces, human (articulated) bodies, *etc.* Still higher-level analyzers look for gaze, gestures, and so on. Similar hierarchies can exist for audio and other input modalities, and these heirarchies can merge as multiple modalities are combined to further refine the understanding of the environment. See [17] for details.

3 Application Programming Requirements

The parallel structure of the Smart Kiosk is highly dynamic. The environment in front of the kiosk (number of customers, and their relative position) and the state of its conversation with the customers affect which threads are running, their relative computational demands, and their relative priorities (*e.g.,* threads that are currently part of a conversation with a customer are more important than threads searching the background for more customers).

A major problem in implementing this kind of application is "buffer management". This is illustrated in the simple vision pipeline shown in Fig. 2. The *digitizer* produces digitized images every 30th of a second. 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 things contribute to complexity in buffer management as we move up to higher levels:

- Threads may not access their input datasets 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. The conversation may even result in cancelling activities initiated earlier, so that they no longer need their input data items. Consequently, producer-consumer relationships are hints and not absolute, complicating efficient data sharing especially in a cluster setting.
- Datasets from different sources need to be combined, correlating them temporally. For example, stereo vision combines data from two or more cameras, and stereo audio combines data from two or more microphones. Other analyzers may work multi-modally, *e.g.,* by combining vision, audio, gestures and touch-screen inputs.
- 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.
- Since computations performed on the data increase in sophistication 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 will necessarily be used at the higher levels. As a result, the datasets become temporally sparser and sparser at higher levels of processing because they correspond to higher- and higher-level hypotheses of interesting events. For example, the lowest-level event may be: "a new camera frame has been captured", whereas a higher-level event may be: "John has just

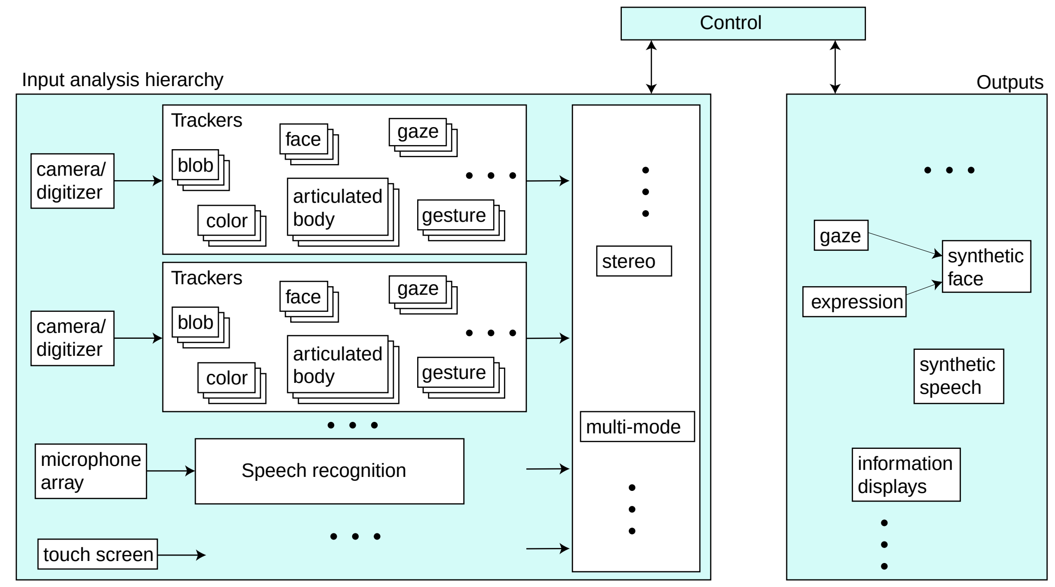


Figure 1: Software architecture of the Smart Kiosk.

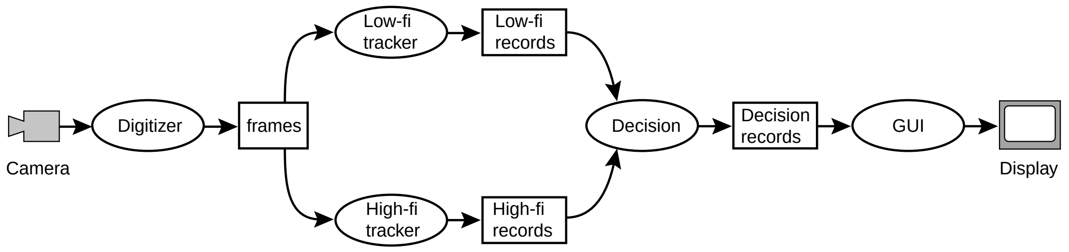


Figure 2: A simple vision pipeline.

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 algorithmic features bring up 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).

4 Space-Time Memory

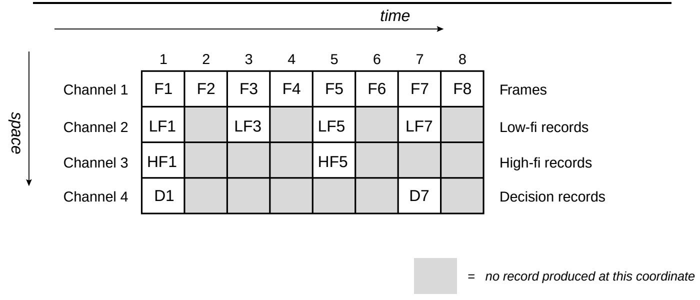
CRL's *Stampede* project addresses the parallel programming requirements posed by such interactive multimedia applications. Stampede allows the creation of multiple address spaces in the cluster and an unbounded number of dynamically created application threads within each address space. The threading model within an address space is basically *pthreads* (POSIX threads) [8]. Stampede provides high-level data sharing abstractions that allow threads to interact with one another without regard to their physical locations in the cluster, or the specific address spaces in which they execute.

A novel component of Stampede is Space-Time Memory (STM), a distributed data structure that addresses the complex "buffer management" problem that arises in managing temporally indexed data items as in the Smart Kiosk application. Traditional data structures such as streams, queues and lists are not sufficiently expressive to handle the requirements enumerated in the previous section.

STM can be envisioned as a two-dimensional table. Each row, called a *channel*, has a system-wide unique id. A particular channel may be used as the storage area for an activity (e.g. a digitizer thread producing digitized camera images) to place the time-sequenced data records that it produces. Every column in the table represents the temporally correlated output records of activities that comprise the computation. For example, in the vision pipeline in Fig. 2, the digitizer produces a frame F_t with a timestamp t. The Low-fi tracker produces a tracking record LF_t analyzing this video frame. The decision module produces its output D_t based on LF_t. These three items are on different channels of the STM and may be produced at different real times, but they are all temporally correlated and occupy the same column t in the STM. Similarly, all the items in the next column of STM have the timestamp $t + 1$. Fig. 3 shows an example of how the STM may be used to orchestrate the activities of the vision processing pipeline introduced in Fig. 2. The rectangular box at the output of each activity in Fig. 2 is an STM channel. The items with timestamp 1 (F_1, LF_1, HF_1, and D_1) in each of the four boxes in Fig. 2 is a column in the STM.

4.1 The API

The Space-Time memory API has operations to create a channel dynamically, and for a thread to *attach* and *detach* a channel. Each attachment is known as a *connection*, and a thread may have multiple connections to the same channel. Fig. 4 shows an overview of how channels are used. A thread can *put* a data item into a channel *via* a given output connection using the call:



	1	2	3	4	5	6	7	8	
Channel 1	F1	F2	F3	F4	F5	F6	F7	F8	Frames
Channel 2	LF1	▨	LF3	▨	LF5	▨	LF7	▨	Low-fi records
Channel 3	HF1	▨	▨	▨	HF5	▨	▨	▨	High-fi records
Channel 4	D1	▨	▨	▨	▨	▨	D7	▨	Decision records

Figure 3: Mapping the vision pipeline to STM.

```
spd_channel_put_item (o_connection, timestamp,
                      buf_p, buf_size, ...)
```

The item is described by the pointer **buf_p** and its **buf_size** in bytes. A channel cannot have more than one item with the same timestamp, but there is no constraint that items be put into the channel in increasing or contiguous timestamp order. Indeed, to increase throughput, a module may contain replicated threads that pull items from a common input channel, process them, and put items into a common output channel. Depending on the relative speed of the threads and the particular events they recognize, it may happen that items are placed into the output channel out of order. Channels can be created to hold a bounded or unbounded number of items. The **put** call takes an additional flag that allows it either to block or to return immediately with an error code if a bounded output channel is full.

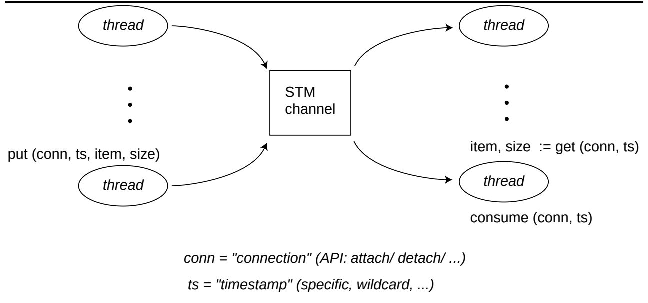

thread
thread
put (conn, ts, item, size)
STM channel
thread
thread
item, size := get (conn, ts)
consume (conn, ts)
conn = "connection" (API: attach/ detach/ ...)
ts = "timestamp" (specific, wildcard, ...)

Figure 4: Overview of Stampede channel usage (relationship of a channel to threads)

A thread can *get* an item from a channel *via* a given connection using the call:

```
spd_channel_get_item (i_connection, timestamp,
                      & buf_p, & buf_size,
                      & timestamp_range, ...);
```

The **timestamp** can specify a particular value, or it can be a wildcard requesting, for example, the newest/oldest value currently in the channel, or the newest value not previously gotten over any connection. As in the **put** call, a flag parameter specifies whether to block if a suitable item is currently unavailable, or to return immediately with an error code. The parameters **buf_p** and **buf_size** can be used to pass in a buffer to receive the item or, by passing NULL in **buf_p**, the application can ask Stampede to allocate a buffer. The **timestamp_range** parameter returns the timestamp of the item returned, if available; if unavailable, it returns the timestamps of the "neighboring" available items, if any.

The put and get operations are atomic. Even though a channel is a distributed data structure and multiple threads on multiple address spaces may simultaneously be performing operations on the channel, these operations appear to all threads as if they occur in a particular serial order.

The semantics of put and get are copy-in and copy-out, respectively. Thus, after a put, a thread may immediately safely re-use its buffer. Similarly, after a successful get, a client can safely modify the copy of the object that it received without interfering with the channel or with other threads. Of course, an application can still pass a datum by reference– it merely passes a reference to the object through STM, instead of the datum itself. The reference can be any Stampede notion of object references.

Puts and gets, with copying semantics, are of course reminiscent of message-passing. However, unlike message-passing, these are location-independent operations on a distributed data structure. These operations are one-sided: there is no "destination" thread/process in a put, nor any "source" thread/process in a get. The abstraction is one of putting items into and getting items from a temporally ordered collection, concurrently, not of communicating between processes.

4.2 Garbage Collection

In dealing with timestamped data in this application domain we encounter an unusual notion of garbage collection, where "reachability" concerns timestamps and not memory addresses. If physical memory were infinite, STM's put and get primitives would be adequate to orchestrate the production and access to time-sequenced data in any application. However, in practice it is necessary to garbage collect data items that will no longer be accessed by any thread. When can we reclaim an item from a timestamp-indexed collection? The problem is analogous to the classical "space leak" situation where, whenever a table is reachable from the computation, no item in that table can be garbage collected on the basis of reachability alone, even if there are items that will never be accessed subsequently in the computation. A complication is the fact that application code can do arithmetic on timestamps. Timestamp-based GC is orthogonal to any classical address-based GC of the STM's host language. This section discusses the guarantees provided by the STM for producing and accessing time-sequenced data, and the guarantees that the application must provide to enable garbage collection.

To enable garbage collection of an STM item, the API provides a consume(*connection,timestamp*) operation by which the application declares to STM that this item is garbage from the perspective of a particular connection. STM can safely garbage collect an item once it has determined that the item can no longer be accessed through any existing connection *or any future connection* to this channel. So the discipline imposed by STM on the application programmer is to get an item from a channel, use it, and mark it as consumed. An object X in a channel is in one of three states with respect to each input connection ic attaching that channel to some thread. Initially, X is "unseen". When a get operation is performed on X over connection ic, then X is in the "open" state with respect to ic. Finally, when a consume operation is performed on the object, it transitions to the "consumed" state. We also say that an item is "unconsumed" if it is unseen or open. The contract between the runtime system and the application is as follows: The runtime system guarantees that an item will not be garbage collected at least until it has been marked consumed on all the connections that have access to it. An application thread has to guarantee to mark each item on its input connections as consumed. The consume operation can specify a particular object (*i.e.,* with a particular timestamp), or it can specify all objects up to and including a particular timestamp. In the latter case, some objects will move directly into the consumed state, even though the thread never performed a get operation on them.

Similarly, there are rules that govern the timestamp values that can be associated with items produced by a thread on a connection. A thread can associate a timestamp t with an item it produces so long as this thread has an item X with timestamp t currently in the open state on one of its input connections. This addresses the common case (*e.g.,* the Low-fi tracker thread in Fig. 2) where a thread gets an item from its input connection, processes it, produces a timestamped output (correlated to the timestamp of the item it is processing, possibly even the same timestamp) as a result of the processing, and marks the item consumed. We say that the output item *inherits* the timestamp of the input item.

However, there are situations where timestamped output may have to be generated without getting an item from the STM. This is in general true for application "source" threads that have no input connections (*e.g.,* the digitizer thread in Fig. 2, with the corresponding code fragment shown in Fig. 6), or a root thread in a task connectivity graph that drives the whole computation. For this purpose, the STM maintains a state variable for each thread called *virtual time.* An application may choose any application-specific entity as the virtual time. For example, in the vision pipeline (Fig. 2), the frame number associated with each camera image may be chosen as the virtual time. The current *visibility* of a thread is the minimum of its virtual time and the timestamps of any items that it currently has open on any of its input connections. When a thread puts an item, it can give it any timestamp $\geq$ its current visibility. When a thread creates a new thread, it can initialize the child thread's initial virtual time to any value $\geq$ its own current visibility. When a thread creates a new input connection to a channel, it implicitly marks as consumed on that connection all items $<$ its current visibility. A thread can explicitly change its own virtual time to any value $\geq$ its current visibility. In most cases, a thread can set its own virtual time to the special value INFINITY because the timestamps of items it puts are simply inherited from those that it gets.

These rules enable the runtime system to transitively compute a global minimum ts_{min}, which is the minimum of:

- virtual times of all the threads, and
- timestamps of all unconsumed items on all input connections of all channels.

This is the smallest timestamp value that can possibly be associated with an item produced by any thread in the system. It is impossible for any current thread, or any subsequently created thread, ever to refer to an object with timestamp less than the global minimum. Thus, all objects in all channels with lower timestamps can safely be garbage collected. Stampede's runtime system has a distributed algorithm that periodically recomputes this value and garbage collects dead items. To ensure that this global minimum advances and thus garbage collection is not stymied a thread must guarantee that it will advance its virtual time, for which STM provides an API call.

The `consume` call is reminiscent of reference counting. However, this is misleading because the number of consumers of an item is unknown– a thread may skip over items on its input connections, and new connections can be created dynamically. These interesting and subtle issues, as well as our distributed, concurrent garbage collection algorithm are described in greater detail in a separate paper [16].

4.3 Synchronization with Real-time

The virtual time and timestamps described above with respect to STM are merely an indexing system for data items, and do not in of themselves have any direct connection with real time. For pacing a thread relative to real time, Stampede provides an API for loose temporal synchrony that is borrowed from the Beehive system [20]. Essentially, a thread can declare real time "ticks" at which it will re-synchronize with real time, along with a tolerance and an exception handler. As the thread executes, after each "tick", it performs a Stampede call attempting to synchronize with real time. If it is early, the thread waits until that synchrony is achieved. It if is late by more than the specified tolerance, Stampede calls the thread's registered exception handler which can attempt to recover from this slippage. Using these mechanisms, for example, the digitizer in the vision pipeline can pace itself to grab images from a camera and put them into its output channel at 30 frames per second, using absolute frame numbers as timestamps.

5 Programming Examples

In this section, we show some STM programming examples. Fig. 5 shows the relationship of an application thread to the STM abstraction. The only interaction it has with the other threads in the application is via the STM channels it is connected to on the input and output sides. Other than the specific calls to the STM to get, put, or consume an item, the thread executes its sequential algorithm.

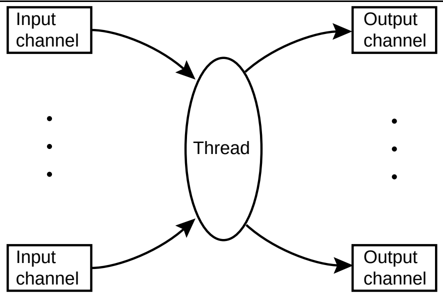


Figure 5: Relationship of an application thread to STM.

For the vision pipeline in Fig. 2, we present code fragments for the digitizer thread and a tracker thread in Figs. 6 and 7, respectively.

Digitizer thread

```
...
/* create an output connection to an STM channel */
oconn = spd_attach_output_channel(video_frame_chan)
/* specify mapping between vt tick and elapsed real-time */
spd_tg_init(TG_DIGITIZE, 33)
/* frame'count will be used as the virtual time marker
   for the digitizer */
frame_count = 0
while (True) {
  frame_buf = allocate_frame_buffer()
  frame_buf ← digitize_frame()

  /* put a timestamped output record of the frame */
  spd_channel_put_item(oconn, frame_count, frame_buf)

  /* advance digitizer's virtual time */
  frame_count++

  /* announce digitizer's new virtual time to STM */
  spd_set_virtual_time(frame_count)

  /* synchronize digitizer's virtual time with real-time */
  spd_tg_sync_vt_with_rt(TG_DIGITIZE)
}
```

Figure 6: Digitizer code using the STM calls.

Tracker thread

```
...
/* announce to STM that the thread's virtual time is
    +infinity for the purposes of garbage collection */
spd_set_virtual_time(+infinity)
/* create an inout connection to the STM channel for
   getting video frames from the digitizer */
iconn_frame = spd_attach_input_channel(video_frame_chan)
/* create an output connection to an STM channel for
   placing tracker output records */
oconn = spd_attach_output_channel(model_location_chan)
while (True) {
  location_buf = allocate_location_buffer()

  /* get the most recent frame produced by the digitizer,
     and record its timestamp in T_k */
  (frame_buf, T_k) = spd_channel_get_item(iconn_frame,
                                          STM_LATEST_UNSEEN)

  /* tracker algorithm for detecting target model
     in video frame */
  location_buf ← detect_target(frame_buf)

  /* put the location of the detected target in STM channel
     corresponding to tracker's output records */
  spd_channel_put_item(oconn, T_k, location_buf)

  /* mark the video frame consumed */
  spd_channel_consume_items_until(iconn_frame, T_k)
}
```

Figure 7: Tracker code using the STM calls.

It can be seen from the code fragments that the extent of application modification required to use the STM is small and localized to the specific regions where a thread would need to communicate with its peers under any parallel programming regime. More importantly, all such communication and synchronization are encapsulated in the get, put, and consume calls. The threads never have to synchronize explicitly with other threads, nor do they have to know the existence of other threads in the applications. All that a particular thread needs to know is the names of the channels it should expect inputs from and the channels to which it should send its outputs (see Fig. 5). Thus STM relieves the application programmer from low level synchronization and buffer management. Moreover, the virtual time and timestamp mechanisms of the STM provide a powerful facility for the application programmer to temporally correlate disparate data items produced at different real times by different threads in a complex application.

Space limitations prevent us from presenting more elaborate programming examples here. The program represented by the code fragments in Figs. 7 and 6 could perhaps have been written using straight message-passing, except that the STM code is still simpler because of its location-independence (producer and consumer need not be aware of each other), and because the consumer has the capability of transparently skipping inputs (using the `STM_LATEST_UNSEEN` flag in its get call). A more elaborate example would involve dynamic thread and channel creation, dynamic attachments to channels, multiple producers and consumers for a channel with complex production and consumption patterns *etc.*. These features, along with STM's automatic garbage collection, would be difficult to reproduce with message-passing code.

The vision group at CRL has adopted the Stampede system as its development platform. In addition to the Smart Kiosk system we described in this paper, Stampede is also being used in another application called image-based rendering [10, 18].

6 Design Rationale

In designing the STM abstraction, we have attempted to keep the interface simple and intuitive. We provide the reasoning behind some of the design choices we made along the way:

- **Virtual versus Real timestamps:** Despite the fact that the primary intent of this abstraction is to support interactive applications, we chose an application-derived quantity to be used as timestamps. We did not see any particular benefit to using real time for temporal correlation. Besides, it was not clear how the runtime could make correlations (using real-time) between independent streams that may use different sampling rates on input data (*e.g.,* voice versus video). We chose to allow the application to specify the mapping of the virtual time ticks to real time, and use that relationship purely for scheduling the threads (*i.e.,* pacing an individual thread's activity) and not for temporal correlation.
- **Virtual Time Management:** As mentioned in Sec. 4.2 a "source" thread (with no input connections) must manage its virtual time explicitly, purely for the purpose of garbage collection, whereas most other threads implicitly inherit time based on what is available on their input connections. A more complex and contrived alternative would have been to let source threads make input connections to a "dummy" channel whose items can be regarded as "time ticks".
- **Connections to Channels:** A design choice is to allow operations directly on channels instead of via explicit connections, thus simplifying the API. However, (1) from an application perspective, this is limiting since a thread loses the flexibility to have multiple connections to the same channel. Such a flexibility would be valuable for instance if a thread wants to create a debugging or a monitoring connection to the same channel in addition to the one that it may need for data communication. While the same functionality could be achieved by creating a monitoring thread, we think that connections are a more intuitive and efficient way to achieve this functionality. (2) From a performance perspective, connections can play a crucial role in optimizing communication especially in a cluster setting by providing a hint to the runtime system as to who may be potential consumers for a data item produced on a channel (so that data can be communicated early).
- **Garbage Collection:** STM provides transparent garbage collection by performing reachability analysis on timestamps. In a cluster, this could be quite expensive since the put and get operations on a channel are location transparent, and can be performed by threads anywhere in the cluster that have connections to that channel. The alternative would have been to associate a reference count and garbage collect an item as soon as its reference count goes to zero. However, in some dynamic applications a producer may not know how many consumers there may be for an item it produces (consider, for example, the Digitizer in Fig. 2). As a compromise we allow a put operation to specify an optional reference count (a special value indicates that the consumer count is unknown to the producer). The runtime employs two different algorithms. The first algorithm uses reference counts. A second algorithm based on reachability analysis is run less frequently to garbage collect items with unknown reference counts.

7 Related Work

The STM abstraction may be viewed as a form of structured shared memory. In this sense it is related to recent distributed shared memory systems (such as Cashmere [13], Shasta [19], and Treadmarks [11]). DSM systems typically offer the same API as any hardware SMP system and therefore are too low level to simplify programming of the complex synchronization and communication requirements found in interactive multimedia applications (as mentioned earlier, STM is useful even on an SMP). Further, from a performance perspective DSM systems are not particularly well-suited for supporting applications with highly dynamic sharing characteristics.

There have been several language designs for parallel computing such as Linda [1], Orca [2], and Cid [15]. The data sharing abstractions in these languages are at a lower level than STM; of course, they could be used to implement STM.

Temporal correlation of independent data streams is a key distinguishing feature of our work from all prior work. The work most closely related to ours is the Beehive [20] software DSM system developed by one of the authors and his colleagues at the Georgia Institute of Technology. The delta consistency memory model of Beehive is well-suited for applications that have the ability to tolerate a certain amount of staleness in the global state information. Beehive has been used for real-time computation of computer graphical simulations of animated figures. STM is a higher level structured shared memory that can use the lower-level temporal synchronization and consistency guarantees of Beehive.

The idea of Space-Time memory has also been used in optimistic distributed discrete-event simulation [9, 5]. The purpose and hence the design of Space-Time memory in those systems is very different from ours. In those systems, Space-Time memory is used to allow a computation to roll-back to an earlier state when events are received out of order. In this paper, we have proposed Space-Time Memory as the fundamental building block around which the entire application is constructed.

8 Performance

In addition to simplifying programming, STM has the potential to provide good performance on clusters, for several reasons. First, synchronization and data transfer are combined in STM, permitting fewer communications. Second, connections provide useful hints for optimizing data communication across clusters. Third, sharing across address spaces is orchestrated via the STM abstraction which can therefore optimize it in a more targeted manner than the indiscriminate sharing that can occur in a DSM system for dynamic applications.

In this section we provide some preliminary performance numbers for STM.

8.1 Platform and Limits

Before we look at Stampede performance it is useful to examine the performance that we hope to achieve and the performance limits of our platform.

The Stampede system is implemented on a cluster of AlphaServer 4100's (SMPs with four 400 MHz EV5 ALpha processors each) interconnected by Memory Channel, running Digital Unix 4.0. The vision tracking component of the Smart Kiosk [17] and an image-based rendering application [10] have been implemented on top of Stampede[18].

A typical digitizing video camera (with associated frame grabbers, *etc.*) delivers 30 images per second, where each image has 320×240 pixels and each pixel has 24 bits of color. Thus, each frame has 230400 Bytes, the total bandwidth is 6.912 MegaBytes/sec, and the inter-frame latency is 33.33 milliseconds. We will use B, msecs, usecs and MB/s as abbreviations for Bytes, milliseconds, microseconds and MegaBytes per second, respectively.

STM is built on top of CLF, our homegrown low level packet transport layer. CLF provides reliable, ordered point-to-point packet transport between Stampede address spaces, with the illusion of an infinite packet queue. It exploits shared memory within an SMP, and any available network between SMPs, including Digital Memory Channel [6], Myrinet [3], Tandem's ServerNet [7], and if none of these are available, ordinary UDP over a LAN. In our tables below, we show numbers for Memory Channel and for UDP running over a 100 Mbit/s FDDI LAN (max 12.5 MB/s).

Minimum one-way end to end latencies achievable under CLF are shown in Table 8, for various packet sizes up to 8152 Bytes, the MTU or maximum packet size under CLF. The minimum latencies (for 8 Bytes) for shared memory and Memory Channel are somewhat high because CLF itself is multi-threaded (in-order to preserve the illusion of an infinite queue) and so communication involves some number of synchronizations and context switches (truly "raw" latencies would be less than 5 usecs).

Maximum bandwidths achievable under CLF are shown in Table 9, for various packet sizes. The rightmost column assumes that a sender waits for an acknowledgement from a receiver after sending an image-worth of data (230400 Bytes), and hence has somewhat lower bandwidths than the column to its left, which does not involve such a wait. Any code that involves some coordination per image frame (as in STM) cannot exceed this limit.

Communication	Packet size (Bytes)				
medium	8	128	1024	4096	8152
Shared Memory (within an SMP)	17	20	37	90	164
Memory Channel (between SMPs)	19	23	42	147	293
UDP/LAN (between SMPs)	227	280	441	983	1423

Figure 8: Minimum one-way send/receive latencies (in usecs) for CLF, for miscellaneous packet sizes.

Comm.	Packet size (Bytes)					
medium	8	128	1024	4096	8152	8152*
Shared Memory (within SMP)	2.3	26.3	76.3	93.5	95.9	87
Memory Channel (betw. SMPs)	2.3	21.8	35.5	37.0	37.1	32.6
UDP/LAN (betw. SMPs)	0.13	1.95	10.1	11.9	12.0	10.5

Figure 9: Max bandwidths (in MB/s) for CLF, for miscellaneous packet sizes. Rightmost column (*) assumes an ack after every 230400 Bytes (size of an image).

8.2 STM performance

We designed various microbenchmarks to measure the basic costs of the frequent STM API calls: put, get and consume.

Fig. 10 shows the average one-way latencies for various payload sizes up to 8112 Bytes (the maximum STM payload in one CLF packet). The experiment sets up a producer thread in one address space that puts items into a channel and a thread in another address space that gets and consumes these items from the channel. We measure the total latency from before the put until after the consume. Depending on whether the channel data structure is co-located with the producer or the consumer, this could take two, four or more round-trip communications. In addition, these operations will involve a number of thread synchronizations and context switches (because manipulating a channel is done with a lock, and remote channel requests are handled by a server thread). Keeping this in mind, the STM numbers are in line with the one-way latencies in Fig. 8. Also, these latencies are still well below the 33 msec frame rate of video camera.

Communication medium	Packet size (Bytes) 8	128	1024	4096	8112
Memory Channel (between SMPs)	242	254	352	607	900
UDP/LAN (between SMPs)	449	487	691	1357	2078

Figure 10: Minimum STM one-way latencies (in usecs) for a *put* on one address space and a *get* and *consume* on another address space, with the channel co-located with the consumer.

Fig. 11 shows bandwidths for image-size payloads (230400 Bytes). In column A, a producer on one address spaces does repeated puts, and a consumer on another address space does repeated gets and consumes. Because of the synchronization between puts and gets and consumes, the data is moved in bursts, one item at a time. The bandwidths are thus much less than the raw CLF bandwidths of the rightmost column of Fig. 9, although they are still comfortably above the basic camera image rate of 6.912 MB/s. In column B, there are two producers on two different address spaces and two consumers on another address space. In this case, one consumer can be involved in data movement while the other consumer is involved in synchronization with its producer, and vice versa, thus increasing total bandwidth seen by the two consumers. We can see that these total bandwidths approach the raw CLF bandwidths of Fig. 9.

Communication medium	Configuration A	B
Memory Channel (between SMPs)	21.7	31.0
UDP/LAN (between SMPs)	8.9	11.2

Figure 11: Maximum total STM bandwidth (in MB/s) at the consumer address space for image-sized payloads (230400 Bytes). Column A has one producer (doing *put*'s) on one address space and one consumer (doing *get*'s and *consume*'s) on another address space. Column B has two producers on two address spaces and two consumers on a third address space.

9 Concluding Remarks

The full implementation of the Stampede system is complete and we are now embarking on more detailed performance analysis and tuning of the system. In particular we would like to use information about the current connections to a channel to pre-emptively send data towards consumers, thereby improving latency and bandwidth through the channel. This must be done with caution because consumers can skip timestamps, so sending such data would be wasted work. We are also porting Stampede to run under Windows NT and over MPI [14], and working towards releasing it to researchers outside CRL.

There are several directions for future research. In complex applications such as the Smart Kiosk, efficient resource management to ensure that critical activities are completed in a timely manner is crucial. A companion paper [12] discusses support for integrating task and data parallelism in such dynamic applications. It explores optimal latency-reducing schedules for task- and data-parallel decompositions. We are looking at several new applications for Stampede in this class of interactive, realistic multimedia applications, both at CRL and at Georgia Institute of Technology.

References

[1] S. Ahuja, N. Carriero, and G. David. Linda and Friends. *IEEE Computer*, 19(8):26–34, August 1986.

[2] H. E. Bal, A. E. Tanenbaum, and M. F. Kaashoek. Orca: A Language for Distributed Programming. *ACM SIGPLAN Notices*, 25(5):17–24, May 1990.

[3] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet – a gigabit-per-second local-area network, Draft 16 Nov 1994.

[4] 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.

[5] K. Ghosh and R. M. Fujimoto. Parallel Discrete Event Simulation Using Space-Time Memory. In *20th International Conference on Parallel Processing (ICPP)*, August 1991.

[6] R. Gillett. MEMORY CHANNEL Network for PCI: An Optimized Cluster Interconnect. *IEEE Micro*, pages 12–18, February 1996.

[7] R. W. Horst and D. Garcia. Servernet SAN I/O Architecture. In *Hot Interconnects Symposium V, Kresge Auditorium, Stanford University, Stanford CA*, August 21-23 1997. See also www.servernet.com.

[8] IEEE. Threads standard POSIX 1003.1c-1995 (also ISO/IEC 9945-1:1996), 1996.

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

[10] S. B. Kang. A Survey of Image-based Rendering Techniques. Technical Report CRL 97/4, Cambridge Research Lab., Digital Equipment Corp., August 1997.

[11] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. Winter Usenix*, 1994.

[12] K. Knobe, J. M. Rehg, A. Chauhan, R. S. Nikhil, and U. Ramachandran. Dynamic Task and Data Parallelism Using Space-Time Memory. In preparation.

[13] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, S. Dwarkadas, and M. Scott. VM-Based Shared Memory on Low-Latency Remote-Memory-Access Networks. In *Proc. Intl. Symp. on Computer Architecture (ISCA) 1997, Denver, Colorado*, June 1997.

[14] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, May 1994. www.mpi-forum.org.

[15] R. S. Nikhil. *Cid*: A Parallel "Shared-memory" C for Distributed Memory Machines. In *Proc. 7th. An. Wkshp. on Languages and Compilers for Parallel Computing (LCPC), Ithaca, NY, Springer-Verlag LNCS 892*, pages 376–390, August 8–10 1994.

[16] R. S. Nikhil and U. Ramachandran. Garbage Collection of Timestamped Data in *Stampede*, January 1999. (submitted for publication).

[17] 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.

[18] J. M. Rehg, U. Ramachandran, R. H. Halstead, Jr., C. Joerg, L. Kontothanassis, and R. S. Nikhil. Space-Time Memory: A Parallel Programming Abstraction for Dynamic Vision Applications. Technical Report CRL 97/2, Digital Equipment Corp. Cambridge Research Lab, April 1997.

[19] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proc. 7th. ASPLOS, Boston MA*, October 1996.

[20] A. Singla, U. Ramachandran, and J. Hodgins. Temporal Notions of Synchronization and Consistency in Beehive. In *9th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1997.

[21] K. Waters and T. Levergood. An Automatic Lip-Synchronization Algorithm for Synthetic Faces. *Multimedia Tools and Applications*, 1(4):349–366, Nov 1995.

[22] K. Waters, J. M. Rehg, M. Loughlin, S. B. Kang, and D. Terzopoulos. Visual Sensing of Humans for Active Public Interfaces. In R. Cipolla and A. Pentland, editors, *Computer Vision for Human-Machine Interaction*, pages 83–96. Cambridge University Press, 1998.