Space-Time Memory: A Parallel Programming Abstraction for Dynamic Vision Applications

James M. Rehg   Umakishore Ramachandran   Robert H. Halstead, Jr.
Christopher F. Joerg   Leonidas Kontothanassis   Rishiyur S. Nikhil
Sing Bing Kang

Cambridge Research Laboratory
Technical Report Series

CRL 97/2
April 1997

Cambridge Research Laboratory

The Cambridge Research Laboratory was founded in 1987 to advance the state of the art in both core computing and human-computer interaction, and to use the knowledge so gained to support the Company's corporate objectives. We believe this is best accomplished through interconnected pursuits in technology creation, advanced systems engineering, and business development. We are actively investigating scalable computing; mobile computing; vision-based human and scene sensing; speech interaction; computer-animated synthetic persona; intelligent information appliances; and the capture, coding, storage, indexing, retrieval, decoding, and rendering of multimedia data. We recognize and embrace a technology creation model which is characterized by three major phases:

Freedom: The life blood of the Laboratory comes from the observations and imaginations of our research staff. It is here that challenging research problems are uncovered (through discussions with customers, through interactions with others in the Corporation, through other professional interactions, through reading, and the like) or that new ideas are born. For any such problem or idea, this phase culminates in the nucleation of a project team around a well articulated central research question and the outlining of a research plan.

Focus: Once a team is formed, we aggressively pursue the creation of new technology based on the plan. This may involve direct collaboration with other technical professionals inside and outside the Corporation. This phase culminates in the demonstrable creation of new technology which may take any of a number of forms - a journal article, a technical talk, a working prototype, a patent application, or some combination of these. The research team is typically augmented with other resident professionals—engineering and business development—who work as integral members of the core team to prepare preliminary plans for how best to leverage this new knowledge, either through internal transfer of technology or through other means.

Follow-through: We actively pursue taking the best technologies to the marketplace. For those opportunities which are not immediately transferred internally and where the team has identified a significant opportunity, the business development and engineering staff will lead early-stage commercial development, often in conjunction with members of the research staff. While the value to the Corporation of taking these new ideas to the market is clear, it also has a significant positive impact on our future research work by providing the means to understand intimately the problems and opportunities in the market and to more fully exercise our ideas and concepts in real-world settings.

Throughout this process, communicating our understanding is a critical part of what we do, and participating in the larger technical community—through the publication of refereed journal articles and the presentation of our ideas at conferences–is essential. Our technical report series supports and facilitates broad and early dissemination of our work. We welcome your feedback on its effectiveness.

Robert A. Iannucci, Ph.D.
Director

Space-Time Memory: A Parallel Programming Abstraction for Dynamic Vision Applications

James M. Rehg   Umakishore Ramachandran
Robert H. Halstead, Jr.   Christopher F. Joerg
Leonidas Kontothanassis   Rishiyur S. Nikhil   Sing Bing Kang

April 1, 1997

Abstract

Commercial MIMD computers promise cost-effective parallel processing for computer vision applications, but programming them is time-consuming and obtaining good performance is often difficult. We describe a class of vision applications which are well-suited to multi-threaded implementation on MIMD computers, using the example of a Smart Kiosk.

We present a novel parallel programming abstraction called Space-Time Memory (STM) which is designed to support the dynamic data and control flow requirements of multithreaded computer vision applications. The STM model provides high-level primitives for memory management and synchronization which simplify software development on MIMD computers.

We demonstrate significant speed-ups for two vision applications, color-based tracking and image-based rendering, using an implementation of the STM on a four processor AlphaServer 4100. These experimental results confirm the potential of the STM to provide convenient access to parallel computing resources.

Rehg, Joerg, Kontothanassis, Nikhil, and Kang are with the Cambridge Research Laboratory.
Email: {rehg,cfj,kthanasi,nikhil,kang}@crl.dec.com

Address for Ramachandran:
College of Computing, Georgia Institute of Technology, Atlanta GA 30332, USA.
Email: rama@cc.gatech.edu

Address for Halstead:
Curl Corporation, 4 Cambridge Center, 7th floor, Cambridge MA 02142, USA.
Email: rhh@curl.com

The work described in this report took place at the Cambridge Research Laboratory.

CRL Technical reports are available on the CRL's web page at
http://www.crl.research.digital.com.

Digital Equipment Corporation
Cambridge Research Laboratory
One Kendall Square, Building 700
Cambridge, Massachusetts 02139 USA

1 Introduction

Many computer vision applications exhibit coarse grain parallelism that makes them well-suited to multi-threaded implementation on modern MIMD (Multiple Instruction Multiple Data) parallel computers. For example, in advanced user-interfaces based on vision and speech sensing [18, 26], streams of video and audio data are processed in an independent or loosely-coupled fashion, resulting in significant task level parallelism. Furthermore, for a single data stream such as video there are typically a variety of low-level processing tasks that can be performed independently.

A multithreaded programming model provides an effective way to exploit task level parallelism. In this approach, each task in the application is implemented as a thread, a sequential flow of control analogous to a conventional sequential program. Multiple threads execute concurrently, sharing the available CPU resources and exchanging data and control signals at well-defined points in their execution. The multithreaded programming model is supported by a wide variety of commercially-available MIMD computers, such as the DIGITAL[1] AlphaServer 4100 and Silicon Graphics Origin 2000. These systems provide hardware shared memory support for communication between threads and software support for scheduling thread execution on multiple CPU's.

Unfortunately, implementing a vision application as a multithreaded program is significantly more difficult than writing a conventional sequential program. The largest source of difficulty arises from managing the communication of information between threads and synchronizing their access to shared data. This is particularly true for applications that are highly interactive and dynamic, and process time-varying streams of data at multiple rates.

A vision-based user-interface, for example, might perform background subtraction and low-level feature tracking at video rates, while performing face detection and object recognition much less frequently. Furthermore, the desired performance and rate of processing may change during run-time depending upon the users' actions and the system's goals. For example, a task such as facial expression analysis could range from being the performance bottleneck to not being performed at all. These factors complicate low-level implementation issues such as the allocation and recycling of frame buffers and the synchronization of processing tasks. As we will see in more detail in Section 2, these issues have a significant impact on programming difficulty and serve as a practical barrier to the widespread use of parallel computing in these applications.

This report describes a novel parallel programming abstraction called *Space-Time Memory* (STM) which is designed to support the dynamic data and control flow requirements of multithreaded computer vision applications. The STM model provides high-level primitives for memory management and synchronization which simplify software development on MIMD computers. Since the STM abstraction is architecture-independent, programs that use it have increased portability. In addition, implementations of the STM can be tuned for a particular architecture, increasing application performance without the need for a separate application tuning step.

This report provides an introduction to the STM model and its API. It is targeted

[1]The following are trademarks of Digital Equipment Corporation: Alpha, AlphaServer, Memory Channel, and the DIGITAL logo.

towards potential users of the abstraction. It does not assume in-depth knowledge of parallel programming on the part of the reader. The STM is a component of the *Stampede* system for computing on clusters of parallel machines which is being developed at the Cambridge Research Laboratory (CRL) of Digital Equipment Corporation. More information about Stampede and the implementation of the STM abstraction can be found in [19].

In Section 2 we describe the Smart Kiosk application which motivates our work. We explore some of the difficulties that arise in writing parallel vision code. We describe the design and core functionality of the STM in Section 3. Section 4 presents the results from a series of experiments that demonstrate the performance of the STM on two vision tasks, color tracking and image-based rendering.

2 The Smart Kiosk: A Dynamic Vision Application

The design of the Space-Time Memory abstraction is motivated by the computational requirements of a class of dynamic, interactive computer vision applications. We introduce this class through a specific example: a vision-based user-interface for a Smart Kiosk [26]. A Smart Kiosk is a free-standing computerized device that is capable of interacting with multiple people in a public environment, providing information and entertainment.

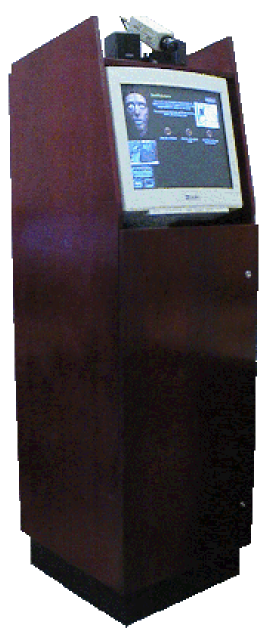


Figure 1: The Smart Kiosk

We are exploring a social interface paradigm for kiosks. In this paradigm, vision and speech sensing provide user input while a graphical speaking agent provides the kiosk's output. A description of the project, including results from a recent public installation, can be found in [4]. A related kiosk application is described in [7].

Figure 1 shows a picture of our most recent Smart Kiosk prototype. The camera at the top of the device acquires images of people standing in front of the kiosk display. The kiosk employs vision techniques to track and identify people based on their motion and clothing color [21]. The estimated position of multiple users drives the behavior of an animated graphical face, called DECface [25], which occupies the upper left corner of the display.

Vision techniques support two kiosk behaviors which are characteristic of public interactions between humans. First, the kiosk greets each person as they approach the display. Second, during an interaction with multiple users DECface exhibits natural gaze behavior, glancing in each person's direction on a regular basis. Future versions of the kiosk will include speech processing and face detection and recognition.

There is currently a great deal of interest in vision- and speech-based user-interfaces (see the recent collections [5, 8]). We believe the Smart Kiosk to be representative of a broad class of emerging applications in surveillance, autonomous agents, and intelligent vehicles and rooms.

A key attribute of the Smart Kiosk application is the real-time processing and generation of multimedia data. Video and speech processing combined with computer graphics rendering and speech synthesis are critical components of a human-centered style of interaction. The number and bandwidth of these data streams results in dramatic computational requirements for the kiosk application. However, there is significant task-level parallelism as a result of the loose coupling between data streams. This can be exploited to improve performance. Unfortunately the complex data sharing patterns between tasks in the application make the development of a parallel implementation challenging.

One source of complexity arises when tasks share streams of input data which they sample at different rates. For example, a figure tracking task may need to sample every frame in an image sequence in order to accurately estimate the motion of a particular user. A face recognition task, in contrast, could be run much less frequently. Differences in these sampling rates complicate the recycling and management of the frame buffers that hold the video input data.

The dynamics of the set of tasks that make up the kiosk application is a second source of complexity. These dynamics are a direct result of the interactive nature of the application. A task such as face recognition, for example, is only performed if a user has been detected in the scene. Thus, whether a task in the application is active or not can depend upon the state of the external world and the inputs the system has received. This variability complicates frame buffer management. In particular, there is a question of which image frames in an input video sequence a newly-activated task should be allowed to see.

2.1 Color-Based Tracking Example

The Smart Kiosk application can be viewed as a dynamic collection of tasks that process streams of input data at different sampling rates. To explore this point further, we focus on a subpart of the Smart Kiosk application that tracks multiple people in an image sequence based on the color of their shirts.

Figure 2 shows the task graph for a color-based person tracking algorithm taken from [21]. It was used in our first Smart Kiosk prototype, which is described in [26]. It tracks multiple people in the vicinity of the kiosk by comparing each video frame against a set of previously defined histogram models of the users' shirt colors. There are four distinct tasks: *digitizer*, *change detection*, *histogram*, and *target detection*, which are shown as elliptical nodes in the diagram. The inputs and outputs for these tasks are shown as rectangles. For example, the *histogram* task reads video frames and writes color histograms. The *target detection* task is based on a modified version of histogram intersection [24].

Figure 3 illustrates the flow of data in the color tracker by following a single image through the task graph. Processing begins at the *digitizer* task, which generates the video frame shown in Figure 3(a). The *change detection* task subtracts a previously

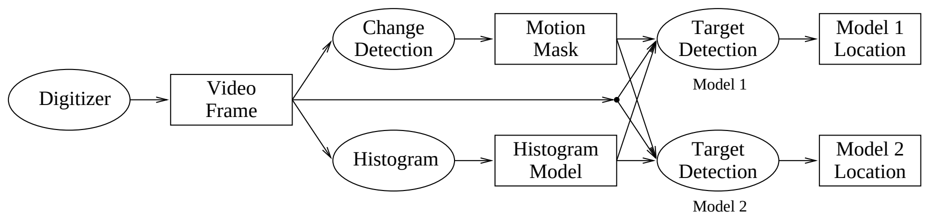


Figure 2: Task graph for the color-based tracker. Ellipses denote *tasks*, implemented as threads. Rectangles denote *channels* which hold streams of data flowing between tasks.

acquired background image from this frame to produce a motion mask (b) showing the foreground objects. Similarly, the *histogram* task produces a histogram model (c) of the video frame (a). Figure (c) shows the red-green projection of a normalized RGB histogram. The *target detection* task compares the image histogram to a previously defined model histogram, resulting in a backprojection image. The sum of the two backprojection images from the two targets is shown in (d). Peaks in the backprojection image correspond to the detected positions of the two subjects (e).

Parallelism at both the task and the data level are visible in the diagram of Figure 2. Task parallelism arises when distinct tasks can be executed simultaneously. It is most obvious in the *change detection* and *histogram* tasks, which have no data dependencies and can therefore be performed in parallel. It is also present in the form of pipelining, where for example the *histogram* and *target detection* tasks can be performed simultaneously on different frames of an image sequence.

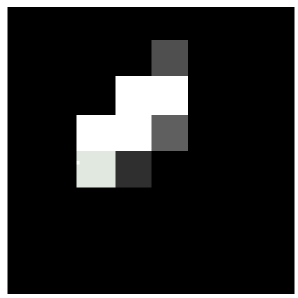
Data parallelism occurs when a single task can be replicated over distributed data. The *target detection* task is data parallel, since it performs the same operation for each color model in the application. The search for a set of models can be performed in parallel by multiple instances of the *target detection* task. For example, Figure 2 illustrates a parallel search for two models. Similarly, data parallelism at the pixel level can be exploited in many image processing tasks, such as *change detection* or *histogram*, by subdividing a single frame into regions and processing them in parallel. In designing a parallel implementation of the color tracker we could focus on task parallelism, data parallelism, or some combination of the two.

Applications such as the Smart Kiosk fit most naturally into a task parallel framework because they are composed of a heterogeneous mixture of relatively independent tasks such as computer graphics rendering, speech, and vision processing. In contrast, it would be difficult to find a single data parallel framework that embraced such a broad range of data types and operations. The Space-Time Memory abstraction supports a task level programming model. We will study the color tracker application from a task-oriented viewpoint in this section. The integration of data parallelism within the task parallel STM framework is addressed in [20].

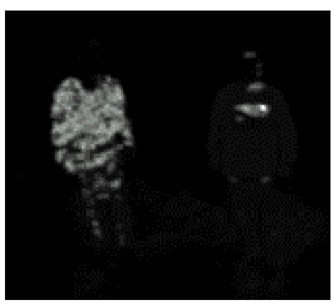
We will examine the color tracker application in Figure 2 through a series of pseudocode implementations. Our focus will be on the implementation of the *channels* for communication between tasks. Each of the rectangular icons in Figure 2 represents a

(a) Video Frame

(b) Motion Mask



(c) Histogram Model



(d) Backprojection Image

(e) Model Locations

Figure 3: Data flow in the color tracker of Figure 2 is illustrated during a search for two models, corresponding to the two figures in the input frame (a). Intermediate results are shown in (b)–(d). The final output is the positions of the detected targets in (e). The motion mask (b) identifies foreground pixels which differ significantly from a background image. The histogram (c) of the input image is compared to the model histograms during search. The backprojection image (d) shows the likelihood that each image pixel belongs to one of the color models (here we show the sum of the two backprojection images). Connected component analysis and peak detection results in the final positions in (e).

channel that supports communication between tasks. Channels can be viewed as sets of buffers for storing sequences of intermediate results. We will examine the complexity that arises in managing these buffers when different tasks access dynamic data at different sampling rates. These examples illustrate some of the difficulties that arise in programming vision applications on MIMD computer systems.

2.2 Sequential Implementation

The first program example we consider is a conventional sequential implementation of the color tracker. For simplicity we will restrict the search to a single model. The four tasks *digitizer*, *change detection*, *histogram*, and *target detection* must exchange three types of data: images, masks, and histograms. The following fragment of pseudocode implements the communication using three global variables, `frame_buf`, `mask_buf`, and `hist_buf`:

```
frame_buf = allocate_frame_buffer()
mask_buf = allocate_frame_buffer()
hist_buf = allocate_histogram_buffer()
Begin loop
  frame_buf ← digitize_frame()
  mask_buf ← detect_change(frame_buf)
  hist_buf ← histogram(frame_buf)
  (x,y) = detect_target(frame_buf, mask_buf, hist_buf)
  ...
End Loop
free(frame_buf)
free(mask_buf)
free(hist_buf)
```

The global variables point to allocated storage. The syntax roughly follows the C language [11], as does the meaning of the assignment operator `=`. The redirection operator ← denotes a function placing its output into a previously allocated buffer.

Function calls inside the inner loop implement the four tasks. The final output of the application is the `(x,y)` location of the detected target in the image (see Figure 3 (e)). This position is computed by `detect_target()` using the input image, motion mask, and image histogram. This function also uses a histogram model of the target, which is omitted for simplicity.

The performance of a particular implementation is constrained by the execution times of its tasks and their data dependencies. Table 4 shows the execution times for the four color tracker tasks in milliseconds. These times were obtained on a 400 MHz AlphaServer 4100. Each task was given the full resources of a single CPU.

The *time-line diagram* in Figure 5 (a) illustrates the interaction between these tasks over time. It is particularly simple in the sequential case. Each task is represented in the diagram by an interval proportional to its execution time. The *application period* shown in the figure is the time interval between two consecutive sets of application outputs. In this case it is simply the sum of the execution times of the individual tasks. Another practical aspect of the sequential implementation is that we can allocate global buffers for the channels at the start of the loop and reuse them in each iteration.

Task	Execution time
digitizer	33 ms
histogram	110 ms
change detection	160 ms
target detection	280 ms

Figure 4: Execution times for the four tasks in the color tracker.

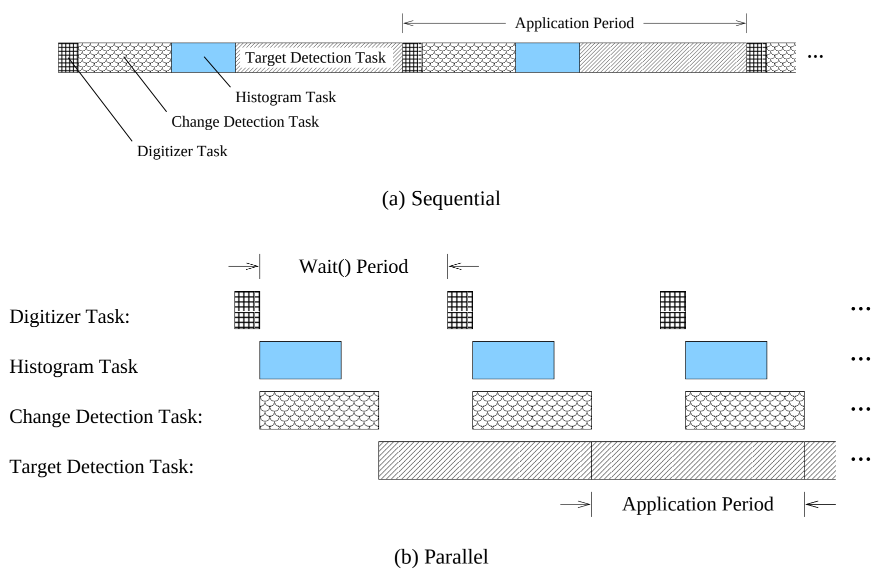


Figure 5: Time-line for sequential and parallel implementations of the color tracker. The throughput and latency is improved in (b) through pipelining and task parallelism.

2.3 Simple Parallel Implementation

We now consider the situation where the *digitizer*, *histogram*, *change detection*, and *target detection* tasks are implemented as separate threads running on different processors in a parallel computer with shared memory. In this situation there is an opportunity to improve performance by exploiting task parallelism. By overlapping the execution of the tasks we can ensure that the slowest task, *target detection*, is never waiting for input data.

The time-line for the simple parallel solution is illustrated in Figure 5 (b). The application period now equals the execution time of the slowest task, as opposed to the sum over all of the tasks. This represents an improvement in the throughput of the application due to pipeline parallelism. Note that the latency[2] is also improved as a

[2]The *latency* of a task or application is defined as the time it takes to produce an output once an input is

result of performing the *change detection* and *histogram* tasks in parallel.

An implementation of the parallel solution depicted in Figure 5 (b) requires two departures from the sequential approach. First, each task becomes a separate thread which must perform mutual exclusion [3] when accessing shared variables to ensure the correctness of the application. We assume the use of a standard thread model such as POSIX [13].

The second departure is an increase in the complexity of buffer management. During pipelining a task has to buffer both the data item it is currently producing and any older items which are being processed in parallel by other tasks. For example, the overlap between the *digitizer* and *target detection* tasks in Figure (b) means that two consecutive video frames will need to be buffered at all times. The following pseudocode implements the buffer management strategy for the *digitizer* and *target detection* tasks:

***digitizer* task**

```
...
Begin Loop i = 1..2
  frame_buf = allocate_frame_buffer()
  enq(reverse_q, frame_buf)
End Loop
Begin Loop
  frame_buf = deq(reverse_q)
  frame_buf ← digitize_frame()
  enq(forward_q, frame_buf)
  wait()
End Loop
```

***target detection* task**

```
Begin Loop
  ...
  frame_buf = deq(forward_q)
  (x,y) = detect_target(frame_buf, mask_buf, ...)
  enq(reverse_q, frame_buf)
  ...
End Loop
Begin Loop i = 1..2
  frame_buf = deq(reverse_q)
  free(frame_buf)
End Loop
...
```

Here we are using a standard bounded queue abstraction to handle the recycling of frame buffers between the two threads. The operations `enq()` and `deq()` denote enqueue and dequeue operations, respectively. These routines can be easily designed to implement mutual exclusion (see [3]). The `wait()` function delays the *digitizer* task by the amount shown in the diagram, to avoid performing unnecessary work. The execution of other three tasks can be regulated solely by the availability of their inputs.

---

available.

In this case, we initialize the `reverse_q` with two buffers that will hold the current and previous digitized frames. When no buffers are available, the call to `deq` will block. All of the synchronization is hidden in the standard `enq()` and `deq()` routines. This example is a two buffer version of the standard producer-consumer problem.

2.4 Complex Parallel Implementation

A realistic vision application will exhibit considerably more complex data sharing patterns than the simple example above. In particular, there will be multiple producers and consumers per channel and more complicated access patterns. The bounded queue abstraction works well in the simple case where every data item is read by exactly one of possibly many consumers. Its extension to an application scenario like the Smart Kiosk is complicated by three factors:

- The number of tasks that will share the data items in a given channel can vary during execution. For example, a video frame may be read by the *change detection* task, the *histogram* task, and zero or more *target detection* tasks, where the exact number depends on the kiosk environment. The rate of execution of these tasks, which determines the rate at which they sample their input data, can also vary.
- Multiple consumers can request the same data item, making a simple queue inappropriate.
- Items may be read and written out of order (i.e., a task may request a specific video frame which is different from the last one produced). Consumers may also want to skip over certain data items in a sequence, and different consumers may want to skip different items.

A secondary complicating issue is that in a cluster setting we cannot depend on the availability of transparent shared memory. Part of the simplicity of the earlier pseudocode implementations stemmed from the fact that shared variables were transparently accessible to threads (i.e., without requiring any additional operations). This is true for threads running on a single SMP with hardware shared memory. But communications between SMP's require a different communication mechanism, such as message-passing, and involve communication delays of a different order of magnitude. Addressing these issues at the application level will dramatically increase the complexity of the code over our simple example and reduce its portability to other hardware platforms.

2.5 Summary

Dynamic vision applications such as the Smart Kiosk have four defining characteristics:

- These applications processes streams of data using a loosely coupled hierarchy of tasks. Tasks in the hierarchy are differentiated by the sampling rates at which they process input data or generate output data.

- The set of tasks is dynamic due to the interactive nature of the application. Tasks can be created and suspended during program execution. The rate at which a task processes input data or generates output data can also change as a function of the system state.
- The set of tasks is heterogeneous. Speech, vision, and rendering algorithms all contribute to the Smart Kiosk application. Even within a single data type such as video the task set can still be quite diverse.
- The target hardware platform can also be heterogeneous, involving machines from multiple vendors linked by a variety of interconnects. We are particularly interested in networked clusters of SMP's such as the DIGITAL AlphaServer 4100.

Implementing such dynamic vision applications on a cluster involves nontrivial parallel programming issues. Addressing these issues at the application level will be time-consuming and will result in complex nonportable code. A better solution is to use a high-level programming model which provides the necessary infrastructure and encapsulates it apart from the application. Unfortunately, commonly-available libraries and run-time systems for parallel systems do not provide adequate primitives for the complex data sharing needs of these dynamic applications. We have developed the Space-Time Memory abstraction to address these needs.

3 Space-Time Memory Abstraction

Space-Time Memory provides a mechanism for sharing time-varying data between threads in a parallel program. A basic element in the model is a *channel*, which holds a single stream of data and supports read and write accesses by multiple threads. In the color-based tracker (see Figure 2) there are five channels which provide storage for digitized frames, motion masks, histogram models, and output records from the two *target detection* threads. Note that each thread writes its results to a channel, and all threads except for *digitizer* read input data from one or more channels.

Each channel holds an ordered sequence of data *items* indexed by *virtual time*, an application-dependent measure of computational progress. Data items can be of arbitrary size and structure. Each item has an associated *time-stamp*, which gives its location within the channel. In the color tracker, progress is measured in video frames, and so the virtual time for the application is frame numbers. In other applications, virtual time might correspond to a sequence of mouse events, or indices in an iteration space.

Time-stamps for a channel item can either be created by a thread or inherited. In our example the *digitizer* thread creates time-stamps, since it originates the flow of data. All of the other threads produce time-stamps for output channel items by inheritance from their input channels. The *change detection* thread, for example, reads a digitized frame with a certain time stamp, T_i, performs pixel-wise background subtraction and thresholding, and then writes the result in the mask channel with time-stamp T_i. Time-stamped items can be read and written in any order, depending upon the needs of each thread.

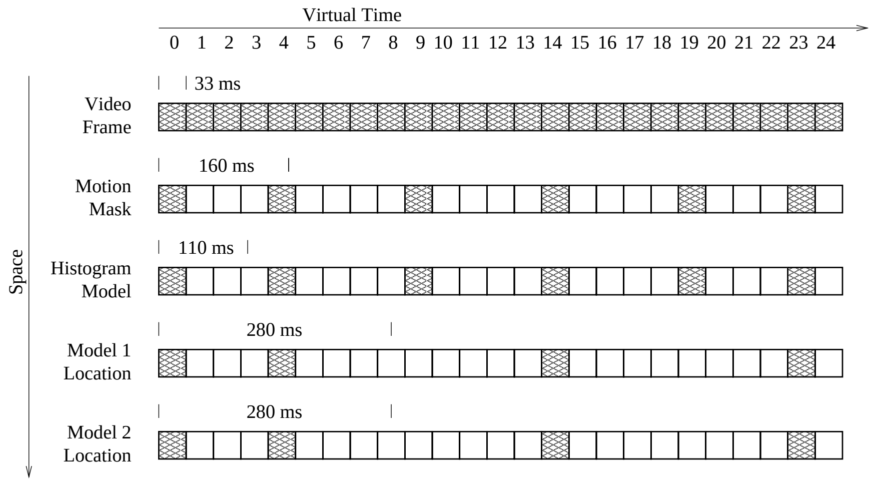


Figure 6: Space-Time Memory array for the channels in the color-based tracker of Figure 2. Cells in the array which contain data items are shaded.

A thread must *attach* to a channel before it can read or write time-stamped data items. When a thread no longer needs a particular channel it can *detach* from it. These operations are analogous to opening and closing a file. Any thread in the application can attach to or detach from any channel at any time. Support for dynamic attachment and detachment is important since threads in the application may not always be active. When the STM is used in a cluster setting, it provides location-independent access. Threads can attach to channels regardless of which node they are currently running on.

The organization of channels within the STM can be viewed as a two dimensional dynamic, concurrent data structure. This is illustrated in Figure 6, which shows a simulation of the contents of the STM during the execution of the color tracker application. The virtual time dimension runs across the top and the space dimension, corresponding to the channels, runs down the side. Threads access shared data at a location in the table by specifying a (channel, time-stamp) pair. The sparseness of the data items in the figure reflects the different sampling rates at which the threads process their inputs.

Data dependencies between threads in the color tracker are easily visible in the figure as a result of time-stamp inheritance. All of the time-stamps used in the color tracker are produced by the *digitizer* thread, which initiates the computation. All other threads write output data items using time-stamps obtained from their inputs. This results in a temporal alignment of occupied cells in the STM for input video frames that pass completely through the application. There are four examples of this in the figure, corresponding to frames 0, 4, 14, and 23.

The production of time-aligned data items is illustrated in more detail in Figure 7 for the case of frame 14 in the image sequence. The figure shows the time-lines for each of the tasks, interspersed with the channels. The time-interval for each task in which time-stamp 14 was processed is shaded. Here we are assuming that each task

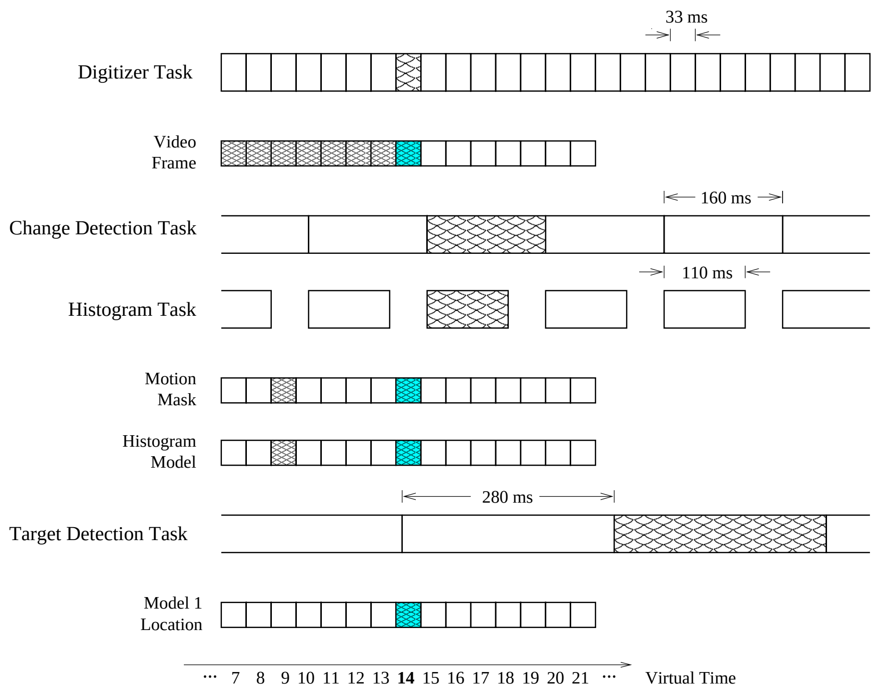


Figure 7: Effect of timing and data dependencies in determining which items are processed is illustrated for time-stamp 14 in Figure 6. The cross-hatched periods are the ones involved in processing item 14.

runs at its maximum rate, given by the timing data in Table 4. The exception is the *histogram* thread which, for simplicity, is delayed so that its execution rate matches *change detection*. Each task selects the most recent data item from its input channel at the start of its computation. This selection process is illustrated in the figure. Each channel is shown in its configuration prior to the start of the shaded interval in which time-stamp 14 is read. The question of which cells in the STM will be occupied is determined by the execution rates of the threads and their data dependencies.

All of the other tasks in the application are dependent upon the *digitizer* task for input data. As a result, the task period for the digitizer defines a natural "unit of real time" for the application. Virtual time provides a powerful abstraction of this conventional notion of a real-time period. For example, if the CPU clock rate doubled, the task execution times in the application would all shrink by one half. The treatment of time-stamps in an STM-based implementation, however, would be unaffected. The application would execute at its new rate, and the run-time system would manage the new pattern of data items in the STM transparently to the application.

The STM is a dynamic data structure. There are four aspects to this. First, new data items are added to the channels continuously and old items are removed as the application progresses. Second, the size of the items stored in a channel can vary during execution. Third, channels can be dynamically created and destroyed. Finally, tasks can attach to and detach from channels as needed.

There are two properties of channels in the STM that distinguish them from conventional dynamic data structures such as variable length arrays. First, channels combine both data storage and synchronization functions in a single primitive, providing a communication substrate for threads in the application. For example, threads that wish to retrieve a time-stamped item from a channel can arrange to be blocked until the item becomes available. This joint functionality is quite convenient. In this respect, channels can be viewed as a generalization of the bounded queue abstraction used in Section 2.3.

The second key property is the flexibility channels provide in reading and writing data items. Threads can read and write data at any sampling rate, and this sampling rate can vary over the course of their execution. Threads can also read and write items out of order. As a result of this flexibility, threads are free to run at arbitrary rates while the STM guarantees the correct buffering and recycling of data. This is especially important in applications like the Smart Kiosk where the computational resources assigned to a thread may change significantly as a function of the world state.

The design of an abstraction such as the STM involves a trade-off between the flexibility and convenience that application programmers desire and the practical considerations of its implementation. For example, some restrictions must necessarily be placed on the application's ability to read and write data items. If threads are allowed to request data items that arrived arbitrarily far back in time, the available system memory would rapidly be exhausted by the accumulation of video frames.

There are two restrictions on reading and writing channel items in the STM which ensure that garbage collection can occur. The first is the *write-once* nature of data items: Once an item with a particular time-stamp has been written, it cannot be modified. It can be read an arbitrary number of times and will be garbage collected when it is no longer needed. The write-once property ensures that a thread never has to read a particular time-stamped item more than once. A second implication is that time-stamps whose data has been garbage collected need not be supported by the STM. This property is consistent with the dynamic nature of data in the STM. For example, a given video frame will not change once it has been digitized, but a new frame will be produced every 33 ms.

The second restriction is that threads cannot write channel items at arbitrary time-stamps, but are restricted to a range of legal values. This range is defined by the *virtual time lower bound* (VTLB), which gives the smallest time-stamp to which a thread may write an item. The abstraction ensures that the VTLB advances as threads process data. This makes garbage collection feasible, since it ensures that any given time-stamped item will eventually pass out of the representation. The VTLB and its relation to garbage collection are discussed further in Section 3.2.

In addition to channels, the STM provides two other more traditional storage mechanisms. The first is a standard queue. The second is a register that holds a single data item. Both can be accessed by multiple threads in a location-independent manner.

3.1 Space-Time Memory API

We will introduce the application programming interface (API) for the STM through pseudocode examples for the *digitizer* and *target detection* threads. In these examples we focus on the use of the STM API to communicate data between tasks. There are five entities that make up the core of the STM abstraction: channels, connections, items, time-stamps, and virtual time. The use of all these entities is illustrated in the following pseudocode for the *digitizer* thread.

```
digitizer task


...
stm_tg_init(TG_DIGITIZE, 280)
oconn = stm_attach_output_channel(video_frame)
frame_count = 0
while {1} {
  frame_buf = allocate_frame_buffer()
  frame_buf ← digitize_frame()
  stm_channel_put_item(oconn, frame_count, frame_buf)
  frame_count++
  stm_set_virtual_time(frame_count)
  stm_tg_sync_to_deadline(TG_DIGITIZE)
}
```

All of the functions beginning with `stm_` are calls to the STM run-time library. For clarity we show only the major arguments required by each function call. Functions that perform synchronization are shown in bold. We do not show any of the thread-level initialization that would occur in a real implementation. The STM is designed to use the Stampede thread model. More information about this model and its API can be found in [19]. We also omit the initialization of STM-specific data structures such as `video_frame`, which describes the STM channel that holds the video frames produced by *digitizer*.

It is instructive to compare the STM pseudocode with the pseudocode for the simple parallel *digitizer* implementation in Section 2.3. The main difference is that the bounded buffers have been replaced by the `video_frame` channel and the `enq()` and `deq()` functions by STM calls. We will explore these differences in more detail below.

The code before the while-loop performs two kinds of initialization. First, it sets up the STM *time group* mechanism which will be used to control the rate of execution of the *digitizer* thread. The call to `stm_tg_init()` creates a time group named `TG_DIGITIZE` with a period of 280 ms. Second, it prepares to write digitized images into the STM. It creates an output connection (`oconn`) by attaching to the `video_frame` channel. This connection will be used to write the digitized frames that the other threads will read. Since the *digitizer* thread does not read from any channels, it must generate its own time-stamps. The initialized `frame_count` variable provides time-stamps by counting the number of frames that have been digitized since the thread was created.

There are three parts to the code inside the `while()` loop. The first part is the standard allocation of an image buffer and the digitization of a frame of video. The second part starts with the call to `stm_channel_put_item()`, which writes the frame buffer into the channel, making it accessible to other threads. The pair of arguments `(oconn, frame_count)` specifies the channel and time-stamp at which to store the data item `frame_buf`. After the item has been written, `frame_count` is advanced and is used to advance the thread's virtual time in `stm_set_virtual_time`. The third part consists of the `sync_to_deadline` call, which regulates the execution of the thread to the specified period of 280 ms.

There are several observations to make about the `put_item` call. First, it is a synchronizing operation. If there is no storage space in the channel available to hold the item, the call will block until space becomes available.[3] This is similar to the behavior of the `enq()` function in the earlier implementation.

The second observation about `put_item` is that the data item can be an arbitrary pointer data structure in C. In this example, `frame_buf` is a pointer to a memory buffer. Support for communicating complex objects made up of multiple pointers is also available, and is described in [19].

The explicit call to `stm_set_virtual_time` is required in most threads such as *digitizer* which create their own time-stamps. This call sets the *thread virtual time*. Through this call, the thread keeps the STM informed about its progress through virtual time. The STM uses the thread virtual time to update the thread's virtual time lower bound (VTLB) and regulate garbage collection. For threads that read time-stamped items, the STM can update the VTLB implicitly and an explicit call is unnecessary. The interaction between virtual time and garbage collection is explored in more detail in Section 3.2.

In the *digitizer* pseudocode from Section 2.3, the function `wait()` was called to regulate the execution of the *digitizer* thread. This serves two purposes. First, it prevents *digitizer* from writing frames which none of the down-stream threads can process due to their execution periods. In comparing Figure 5(b) and 7 it is clear that *digitizer* is doing unnecessary work in the latter.

The second purpose of `wait()` is to ensure that *digitizer* is invoked as "late" as possible, so the most recently available image data is provided to downstream tasks. An alternative, for example, would be to rely on the blocking property of `deq()` and the limited number of buffers to regulate execution. This has the disadvantage of blocking *digitizer* after it has acquired its most recent image rather than before.

The STM uses time groups to provide a general framework for loosely synchronizing the execution of a task to a pre-specified real-time period. When `stm_tg_sync_to_deadline()` is called, it checks the current execution time of the thread against its desired period and suspends the thread if necessary. This mechanism is described in more detail in Section 3.3.

We now examine the pseudocode for the *target detection* thread:

```
target detection task

...
iconn_frame = stm_attach_input_channel(video_frame)
iconn_mask = stm_attach_input_channel(motion_mask)
iconn_hist = stm_attach_input_channel(histogram_model)
```

[3]The function can also be called with the option of returning immediately with an error code in the event of a lack of space. This would allow the application to try an alternate strategy.

```
oconn = stm_attach_output_channel(model_location)
while {1} {
  location_buf = allocate_location_buffer()
  (hist_buf, Tk) = stm_channel_get_item(iconn_hist,
                                         STM_LATEST_UNSEEN)
  mask_buf = stm_channel_get_item(iconn_mask, Tk)
  frame_buf = stm_channel_get_item(iconn_frame, Tk)
  location_buf ← detect_target(frame_buf, mask_buf, hist_buf)
  stm_channel_put_item(oconn, Tk, location_buf)
  stm_channel_consume_items_until(iconn_frame, Tk)
  stm_channel_consume_items_until(iconn_mask, Tk)
  stm_channel_consume_items_until(iconn_hist, Tk)
}
```

In this example there are both input and output channel connections. The first call to `get_item` uses the distinguished value STM_LATEST_UNSEEN, which selects the most recent time-stamp the thread has not yet read and returns it as T_k along with the histogram. This calling convention ensures that the thread will always see the most recent data. T_k is used in subsequent get item calls to ensure that a corresponding set of data is retrieved from the STM. It is also used in writing the final output.

The `consume_item` calls are required for garbage collection. They allow the thread to identify data items it no longer wants to access on all of the ports for which it has input connections. The `consume_item_until` function marks all items up to and including the indicated time-stamp as "consumed". This process is examined in more detail in the next section.

3.2 Garbage Collection

Efficient garbage collection is a key attribute of the STM implementation. It allows the programmer to view each channel as an infinite collection of time-stamped items and it allows threads to sample a channel's contents at arbitrary rates. Threads allow the STM to perform garbage collection by *consuming* items which are no longer needed. Threads consume items which they have either read and processed or are no longer interested in reading. An item in a channel can be garbage collected by the STM implementation when every thread attached to that channel for reading has consumed the item.

There are three practical issues that garbage collection in the STM must address. First, garbage collection is most efficient when contiguous sets of time-stamps can be freed. To the extent possible, we want to avoid examining channel data on an item by item basis. Second, in order to provide flexibility to the application, threads should be able to read all of the items that arrive on the channels they are attached to. One difficulty is that items may arrive on a channel out of order, particularly in a distributed implementation. Third, threads should be allowed to dynamically attach to and detach from channels without halting garbage collection.

The STM abstraction includes a set of rules for reading and writing data items to channels. These rules are designed to encourage programs to generate dense sets of used channel items which can be garbage collected efficiently. The only rule for reading is that threads must agree to consume every item on all of the channels they can read from. This rule alone is not sufficient, however, since we must also restrict threads from writing new items arbitrarily far back in time. This is accomplished by the introduction of a set of legal time-stamps for writing, called the *virtual time window*.

The virtual time window specifies the interval of time-stamps that a thread is allowed to write to. It is defined by the *virtual time lower bound* (VTLB), denoted by T_{lb}. Threads can write to any time stamp that is greater than or equal to their VTLB. This definition allows threads to skip arbitrarily far into the future, but limits their ability to write arbitrarily far back in time.

For threads that use inheritance to generate time-stamps, there is a simple expression for T_{lb} as the minimum time-stamp over the set of items that have not yet been consumed. We refer to this specific time-stamp as T_{nc}. Simply taking $T_{lb} = T_{nc}$ would ensure that the VTLB continuously advances as threads consume items, making garbage collection feasible. However, there are threads such as the digitizer which do not read any items from channels. For these threads, and for increased flexibility, the abstraction also allows a thread to control its VTLB indirectly. The complete definition for the VTLB is given by:

$$T_{lb} = \min(T_{nc}, T_*)$$

where a thread can set its virtual time marker, T_* explicitly through the API. In the pseudocode of Section 3.1, the *digitizer* thread uses the STM call `stm_set_virtual_time` to advance its T_*. By fixing T_* to a specific value, a thread could also ensure that garbage collection will not proceed beyond that point. This could be useful for debugging or system monitoring.

Each thread has its own virtual time lower bound, and its own values for T_{nc} and T_*. These values apply across all of the channels the thread is attached to. When threads are created, they usually inherit the VTLB of their parent. Likewise, when threads attach to a channel for reading, they are restricted to items which lie within their current VTLB. Through these conventions the VTLB provides a solution to what would otherwise be a difficult question: deciding which items on a channel a newly attached thread should be allowed to see. A global comparison of the virtual time windows for all of the threads in an application is the basis for STM garbage collection.

3.3 Synchronization with Real-Time

In addition to synchronization and communication between threads, the STM also provides support for loose synchronization between real and virtual time. Threads can specify a desired real-time execution period which the STM will try to enforce. The STM automatically suspends threads which complete before their execution period has ended and generates warnings which can be handled by the application for threads that exceed their period.

The abstraction allows a thread to define a real-time interval, R_I, corresponding to one unit of virtual time. This is done through the function `stm_tg_init()`. In the color tracker example of Section 3.1, the *digitizer* thread set $R_I = 280$ ms, which corresponds to the execution period of the *target detection* thread. This prevented the *digitizer* thread from running more often than necessary to supply the *target detection* thread with data.

Given the current virtual time of a thread, as measured by T_{lb} for example, the STM can predict the expected real-time at the end of the current iteration as $\hat{R} = (T_{lb} - T_{lb}^0) * R_I + R_0$, where R_0 and T_{lb}^0 give the real and virtual times, respectively, at which the thread was created. The actual real-time can be measured and compared to $\hat{R}$ to determine whether the thread is ahead of or behind its desired schedule.

Warnings can be generated when threads exceed their $\hat{R}$, allowing the application to take an appropriate action. In fact, the API allows a thread to register a handler function which is called in the event that the thread misses its deadline. If a thread finishes earlier than $\hat{R}$ it can be suspended, avoiding unnecessary work. Note that this mechanism simply provides a kind of loose synchronization among threads and is distinct from the features that a real-time operating system could provide, for example.

4 Experimental Results

We implemented the STM on a DIGITAL AlphaServer 4100, a four processor SMP with a 400 MHz clock running DIGITAL Unix 4.0. We ran experiments on two sets of applications, color tracking and image-based rendering, to evaluate the effectiveness of the STM. The first part of each experiment was to port an existing sequential program onto the abstraction. This provided an informal test of its ease of use. In both cases we obtained a working STM version of our sequential code within a half-day. While ease of use is a difficult attribute to measure, we believe that more widespread experimentation with the STM will provide additional evidence of its utility. In the second part of each experiment we analyzed the timing performance of the application to understand how much of the available parallelism the STM could exploit.

4.1 Color-Based Tracker

The first set of experiments measured the STM's effectiveness in performing a parallel search over a set of models in the color-based tracking application. For these experiments we modified the task graph of Figure 2 in two ways. First, we employed a modified version of the histogram algorithm that uses the motion mask to avoid adding background pixels to the model. This improves the quality of the estimate. More importantly, this introduces a dependency between the *change detection* and *histogram* tasks, which forces them to execute sequentially.

The second change was to add an additional STM register to the implementation to prevent pipelining. The register provided a control path from a graphical user-interface to the digitizer, making it possible to send one frame at a time through the implementation. These two changes eliminated task and pipeline parallelism from the original task graph. The available parallelism was contained solely in the *target detection* task. This was done to simplify the experimental analysis.

In this experiment we varied the number of *target detection* threads, each of which searched the entire input frame for a single model. We measured the average execution times as the number of models varied from one to eight and the number of processors varied from one to four. The complete set of data is shown graphically in Figure 8.

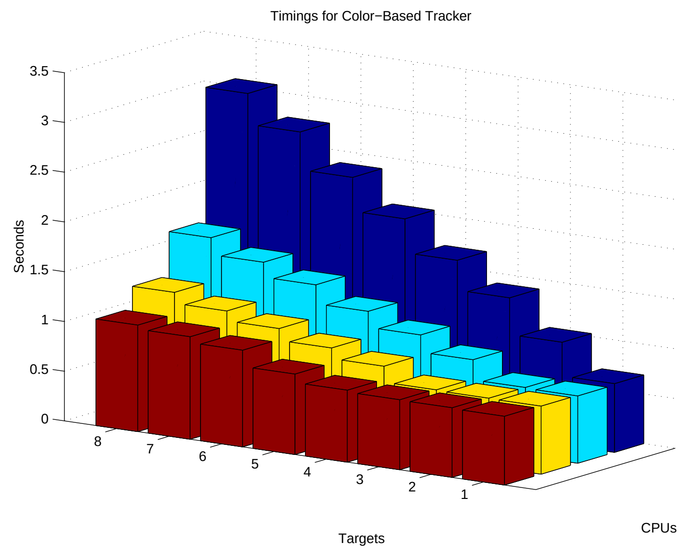


Figure 8: Bar graph of execution times from color-based tracking experiments.

The modified version of the application has a sequential component which includes the digitizer, change detection, and modified histogram threads. Once they have been completed a parallel search over models can occur. Excluding any communication costs, the average time for the sequential task set was 335 ms, and which was approximately equal to the average time for a single *target detection* thread to search for one model. Thus the minimum possible execution time was 670 ms, and could be obtained whenever the number of CPU's is greater than or equal to the number of models. Letting m be the number of models and n the number of CPU's, we see that the bar graph is quite flat in the region $m \leq n$. The average execution time over that region is 693.3 ms.

Given measurements of the sequential and parallel components of the application, we can predict an ideal value for each measurement though the formula: $T_{mn} = mT_p/n + T_s$ where T_s and T_p are the sequential and parallel times, respectively. Predicted performance is plotted against the data as a family of curves in Figure 9. Plots are given for 2, 4, 6, and 8 models with the number of processors varying along the curve. The predicted and measured curves are quite close, and the average error across the points is 52.7 ms. In addition, the measured speed-up from one CPU to four with eight models is 2.87.

The close correspondence between the measured and predicted performance numbers suggests that use of the STM did not introduce significant overhead in this example. The low communication costs are due primarily to the fact that we are exploiting hardware shared memory in the STM implementation and passing images by reference, thereby avoiding unnecessary copying. These promising results suggest that the STM can provide significant performance improvements at a relatively low implementation cost.

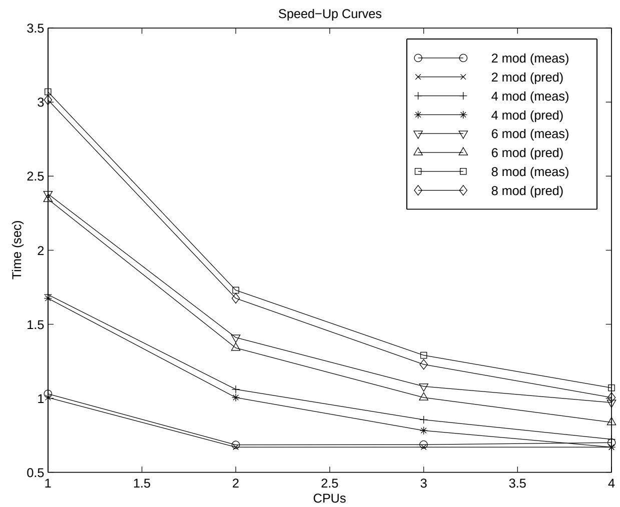


Figure 9: Execution times by number of cpus for 2, 4, 6, and 8 color models.

4.2 Image-Based Rendering

We implemented and tested a second application in the area of image-based rendering (IBR) in order to explore a more traditional form of data parallelism in the STM context. View transfer is the essential idea behind the IBR application: Given two images that have been placed in pixel-wise correspondence and a desired virtual camera viewpoint into the scene, a new image that depicts the scene from the desired viewpoint can be synthesized directly from the input pair [2, 17]. This approach to rendering is interesting because its complexity is a function of image size rather than scene complexity. See Figure 10 for an example of a synthesized image.

There are two main steps in synthesizing a new view through IBR once the correspondences have been obtained off-line: computing the initial transfer and filling in holes through interpolation. We use an Elliptical Weighted Averaging (EWA) technique for interpolation [10], in which adaptively-sized interpolant kernels are applied to image locations with missing texture. Both the view transfer and EWA steps are good candidates for parallelization.

Data parallelism at the pixel-level is the main feature of the IBR application. Each image location can be processed independently of the others, with no sharing of data between neighbors. Our goal in this case was to demonstrate that the STM framework can also be profitably applied to this problem, which is closer to traditional fine-grained parallel vision applications. In this case we divided the image up into a parameterized number of regions, where each region could be processed in parallel. We created a separate thread for each region, and measured the average execution time as the number of regions varied from one to eight and the number of processors varied from one to four. The total set of measurements are plotted in Figure 11.

In addition to the STM implementations, we developed a baseline implementation

(a) View One

(b) View Two

(c) Synthesized view

Figure 10: Input pair (a) and (b) and synthesized third view (c) in an image-based rendering application.

that did not use the STM infrastructure, but instead used lower-level synchronization operations (mutexes) and the hardware shared memory of the AlphaServer. This implementation used four processors and was tested on the same set of regions. These numbers are labeled “HT” (for hand-tuned) in the figure. The same data is displayed as superimposed 2-D plots in Figure 12.

There are several interesting observations we can make regarding the performance numbers for the STM experiments. In the case where there is only a single CPU, total execution time increases slightly with the number of regions. This reflects the additional system-level overhead from context switching as the number of threads grows. Within each of the experiments plotted in Figure 12, the STM implementations show significant speed-ups as the number of image regions is increased until they exceed the number of CPU’s, at which point the speed-up tails off.

The comparison between the cases “4” and “HT” illustrates the additional benefits that can be obtained when a parallel implementation is tightly matched to the computational characteristics of an application. The additional performance in the HT implementation is the result of two optimizations that were not naturally present in the baseline STM implementation. The first optimization exploited the fact that when

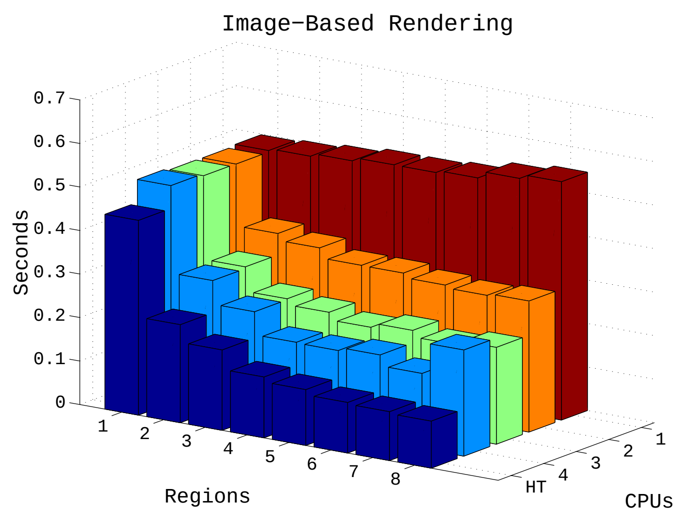


Figure 11: Bar graph of execution times from image-based rendering experiments. The rows labeled 1–4 correspond to the STM implementation with 1–4 processors. The row labeled HT shows the performance of a hand-tuned implementation on 4 processors.

there is no pipelining, the view transfer and EWA steps are sequential. In this case, semaphores can be used to suspend the EWA threads while the view transfer threads are active, and vice versa. This semaphore mechanism is more efficient than the thread yield mechanism which is used in the STM implementation. It ensures that the unused threads consume essentially no system resources.

The second optimization came from exploiting the nonuniformity of the computational requirements in the EWA step. In EWA, the amount of work done in a region is a function of the number of "holes" in the image that must be filled in. Some regions may have no holes, some regions may have many. When the number of threads is equal to the number of regions, some threads do little work but still contribute to the overhead. In the HT implementation there were only four threads, one per CPU, and the regions were scheduled on these threads by the application using a round-robin policy. This explains the additional speed-up in the case where the number of regions increased from four to eight.

There are two lessons from this experiment. The first is the obvious point that code which is hand-tailored to an application will almost always outperform code which uses a more general infrastructure. However, in addition to raw performance we must also consider the ease of development for each implementation. The development of both versions took on the order of hours. However, the STM version was developed by vision researchers who were not experts in programming SMP's. In contrast, the HT application was developed by an expert with a great deal of experience in porting applications to this platform.

The second lesson is that the task parallel STM infrastructure alone cannot capture the full gamut of parallelism which is present in vision applications. We have already

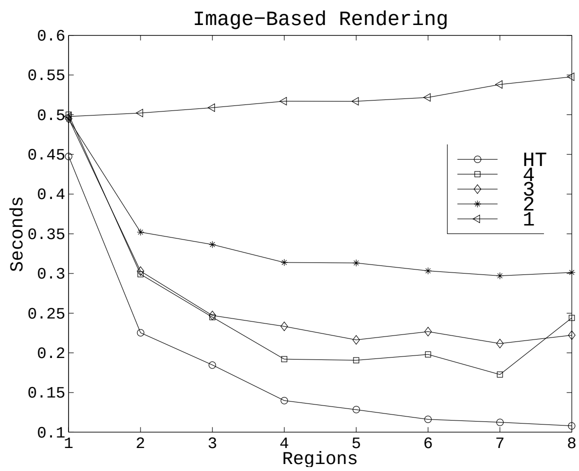


Figure 12: Plots of execution times from image-based rendering experiments. This is the same data as the IBR bar graph.

begun an investigation into more principled schemes for integrating data parallelism into the STM framework. Some preliminary results are described in [20]. We plan to repeat the IBR experiments using our new framework in future work.

Another metric for comparing these implementations is portability. Repeating the STM experiment on a cluster setting would be trivial, as we recently completed an implementation of the STM on a cluster of SMP's in our lab (see [19]). We plan to conduct this experiment in future work. In contrast, the prospects for porting the HT implementation to a cluster are not nearly as promising. Shared memory support in a cluster setting is not widely available, and so a cluster implementation would require significant modifications to the code.

5 Previous Work

During the 1980's and early 1990's there was an explosion of effort in applying parallel computing to computer vision tasks (see the conference series [27]). On the hardware side, the vision community participated in the development of a broad variety of parallel architectures. Representative commercial systems include SIMD machines such as CM-2 [12] and MasPar [15] and systolic/data-flow machines such as the Warp [1]. Experimental parallel computers included pyramid architectures such as the IUA [28] and reconfigurable machines such as PASM [22].

Each of these parallel architectures represented a particular viewpoint on the diverse computational requirements of vision problems. SIMD and data-flow architectures, for example, targeted low-level vision tasks such as histogramming, image smoothing, and convolution. Pyramid machines implemented a hierarchical decomposition of vision problems in hardware. Reconfigurable machines explored the dynamic configuration of processing resources between low and high level vision tasks. This period of research produced considerable insight into the computational requirements of vision problems and the feasibility of high performance parallel solutions. Perhaps the most important lesson was the critical role parallel computing plays in *enabling* the application of computer vision to real-world tasks [6].

Today, commercial MIMD computers seem to be the most cost effective path to wide-spread use of parallel computing. Surprisingly, there has been little investigation into systematic methods for building vision applications on this type of architecture. Of course, the more traditional approach of exploiting fine-grained pixel-level parallelism with specialized architectures is still being pursued. Some examples are [16] and many papers in [27]. While this type of problem is undeniably important, many real-world vision applications will contain a mixture of different types of parallelism, suggesting that an architecture with maximum flexibility could have wider applicability.

In developing STM one of our goals is to make the use of parallel computing by the general vision research community more widespread by developing a software infrastructure that is convenient and easy to use, and which runs on commodity parallel machines and operating systems. A system such as STM will not be able to achieve the peak performance of custom hardware on certain applications, but it may well have a much broader impact.

Portability is another key benefit of a software-only parallel solution such as Space-Time Memory. Applications which use the STM can run on any multiprocessor machine that supports the abstraction, regardless of its underlying memory and communication architecture. The STM is implemented as a C library and does not rely on any special features of a multiprocessor architecture. This simplifies the task of porting it to a new MIMD architecture considerably. By focusing on the portability of our abstraction, we are trying to maximize the chance that it can migrate to new generations of multiprocessor computers.

Within the scalable and parallel computing research community there is a great deal of work that is relevant to STM. The STM abstraction may be viewed as a *structured shared memory*, in comparison to the transparent shared memory that many SMP architectures support. Atomicity for reads and writes in the STM is at the level of channel items. As a consequence, there is considerable opportunity for aggregating and coalescing put operations based on the expected rates of execution of threads communicated to the abstraction. We therefore expect our abstraction to be valuable in devising an efficient implementation for a cluster of SMP's in which there is no hardware support for shared memory. This seems like a promising approach to large scale parallelism.

The most closely related work is the Beehive [23] system developed by the second author and his colleagues at the Georgia Institute of Technology. Beehive is a software distributed shared memory system that provides transparent access to a shared address space across a cluster of Sun workstations. The API provided by Beehive is very similar to standard shared memory programming with the difference that the memory access and synchronization primitives have temporal correctness guarantees. The delta consistency memory model of Beehive is particularly 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. The STM abstraction proposed in this paper 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 [14, 9]. 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.

6 Future Work

There are two compelling avenues for future work. The first is an experimental investigation of the STM on a cluster of SMP nodes. We recently completed a cluster implementation of the STM. There are two avenues of experimental work we would like to pursue. The first is a more precise characterization of the overhead of the STM.

The second avenue is a further investigation of the performance of the color tracking and image-based rendering applications. In a cluster setting questions about the physical location of threads on distinct nodes have an impact in performance. There is also the opportunity to exploit the structure the STM imposes on memory accesses to optimize communication in a cluster setting. A basic question we plan to answer is how overheads differ within and across nodes in the cluster. This will influence the additional speed-ups that are available in a cluster setting.

The second important direction is the integration of task and data parallelism in the STM context. Some preliminary work on this problem is described in [20]. There we present a framework for incorporating data parallelism into a task-oriented description of an application. An important observation about applications like the kiosk is that the optimal division into task and data parallel components must be dynamic. For example, the parallel strategy for the color tracker which delivers the best performance changes with the number of targets being tracked.

Our ultimate goal is a wide-spread distribution of the STM to members of the vision community, and others who may find it useful for their applications. In particular, we would like to quantify the impact of the STM in enabling developers who are not experts in parallel computing to exploit parallelism in their applications. One step towards this goal is to partner with a small number of university research groups who would be interested in adopting the STM as a development platform. A second step is to port the STM implementation to the NT operating system. The STM currently runs on a cluster of AlphaServers under DIGITAL UNIX. An NT port is currently in progress.

7 Conclusions

We have presented a new temporal programming abstraction, called Space-Time Memory (STM). The STM abstraction is tailored to the computational requirements of an emerging class of dynamic, interactive vision applications. We introduced this application class and described its computational properties, using the example of a Smart Kiosk user-interface.

The STM abstraction provides a high level programming model which simplifies the task of sharing time-varying data between threads in a parallel program. We have demonstrated the application of the STM to two vision problems: color-based tracking and image-based rendering. In both cases, the STM delivered significant speed-ups. We believe that the STM can be useful across a broad range of interactive applications, including computer graphics animation and multimedia indexing, retrieval, and editing.

Acknowledgments

The authors would like to thank Kath Knobe for many valuable discussions and for her extensive comments on this manuscript which improved it substantially. Kath, James Hicks, and Mark Tuttle also provided an excellent sounding board for ideas during the development of the STM.

References

[1] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Menzilcioglu, and J. A. Webb. The Warp computer: Architecture, implementation, and performance. *IEEE Transactions on Computers*, 36(12):1523–1538, December 1987.

[2] S. Avidan and A. Shashua. Novel view synthesis in tensor space. In *Conference on Computer Vision and Pattern Recognition*, pages 1034–1040, San Juan, Puerto Rico, June 1997.

[3] M. Ben-Ari. *Principles of Concurrent Programming*. Prentice-Hall, 1982.

[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] R. Cipolla and A. Pentland, editors. *Computer Vision for Human-Machine Interaction*. Cambridge University Press, 1998. In press.

[6] J. D. Crisman and J. A. Webb. The Warp machine on Navlab. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(5):451–465, May 1991.

[7] T. Darrell, G. Gordon, J. Woodfill, and M. Harville. A virtual mirror interface using real-time robust face tracking. In *Proceeding of the Third International Conference on Automatic Face and Gesture Recognition*, pages 616–621, Nara, Japan, April 14–16 1998.

[8] *Proc. of Third Intl. Conf. on Automatic Face and Gesture Recognition*, Nara, Japan, April 14–16 1998. IEEE Computer Society.

[9] K. Ghosh and R. M. Fujimoto. Parallel discrete event simulation using space-time memory. In *20th International Conference on Parallel Processing (ICPP)*, August 1991.

[10] N. Greene and P. Heckbert. Creating raster Omnimax images from multiple perspective views using the Elliptical Weighted Average filter. *IEEE Computer Graphics and Applications*, pages 21–27, June 1986.

[11] S. P. Harbison and G. L. Steele, Jr. *C: A Reference Manual*. Prentice-Hall, 2nd edition, 1987.

[12] D. Hillis. *The Connection Machine*. MIT Press, 1985.

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

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

[15] T. M. Jochem and S. Baluja. A massively parallel road follower. In *Proc. of Comp. Arch. for Machine Perception (CAMP '93)*, New Orleans, LA, December 1993.

[16] T. Kanade, A. Yoshida, K. Oda, H. Kano, and M. Tanaka. A stereo machine for video-rate dense depth mapping and its new applications. In *Computer Vision and Pattern Recognition*, pages 196–202, San Francisco, CA, June 1996.

[17] S. B. Kang. A survey of image-based rendering techniques. Technical Report CRL 97/4, Cambridge Research Lab., Digital Equipment Corp., August 1997.

[18] P. Maes, T. Darrell, B. Blumberg, and A. Pentland. The ALIVE system: Wireless, full-body interaction with autonomous agents. *ACM Multimedia Systems*, Spring 1996.

[19] R. S. Nikhil, U. Ramachandran, J. M. Rehg, K. Knobe, R. H. Halstead Jr., C. F. Joerg, and L. Kontothanassis. *Stampede*: A programming system for emerging scalable interactive multimedia applications. Technical Report CRL 98/1, Digital Equipment Corp. Cambridge Research Lab, Cambridge MA, May 20 1998.

[20] J. M. Rehg, K. Knobe, U. Ramachandran, and R. S. Nikhil. A framework for integrated task and data parallelism in dynamic applications. Technical Report CRL 98/2, Digital Equipment Corp. Cambridge Research Lab, Cambridge, MA, May 1998. To appear in *4th Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers.* Springer-Verlag. In press.

[21] 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.

[22] H. J. Siegal, L. J. Siegel, F. C. Kemmerer, P. T. Muller, H. E. Smalley, and D. Smith. PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Transactions on Computers*, 30:934–947, 1981.

[23] 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.

[24] M. Swain and D. Ballard. Color indexing. *International Journal of Computer Vision*, 7(1):11–32, 1991.

[25] K. Waters and T. Levergood. An automatic lip-synchronization algorithm for synthetic faces. *Multimedia Tools and Applications*, 1(4):349–366, Nov 1995.

[26] K. Waters, J. M. Rehg, M. Loughlin, S. B. Kang, and D. Terzopoulos. Visual sensing of humans for active public interfaces. Technical Report CRL 96/5, Digital Equipment Corp. Cambridge Research Lab, 1996. To appear in *Computer Vision for Human-Machine Interaction*, R. Cipolla and A. Pentland (editors), Cambridge University Press. In press.

[27] C. C. Weems, Jr., editor. *Computer Architectures for Machine Perception*, Cambridge, MA, October 20–22 1997.

[28] C. C. Weems, Jr., S. P. Levitan, A. R. Hanson, E. M. Riseman, D. B. Shu, and J. G. Nash. The image understanding architecture. *International Journal of Computer Vision*, 2:673–688, 1989.

Space-Time Memory: A Parallel Programming Abstraction for Dynamic Vision Applications

James M. Rehg Umakishore Ramachandran Robert H. Halstead, Jr. Christopher F. Joerg Leonidas Kontothanassis Rishiyur S. Nikhil Sing Bing Kang

CRL 97/2

April 1997