Stampede
A programming system for emerging scalable interactive multimedia applications

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

Cambridge Research Laboratory
Technical Report Series

CRL 98/1
May 1998

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

Stampede
A programming system for emerging scalable interactive multimedia applications

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

May 20, 1998

Abstract

Stampede is a programming system for emerging scalable applications on clusters. The goal is to simplify the programming of applications that are interactive (often using vision and speech), that have highly dynamic computation structures, and that must run on platforms consisting of a mix of front-end machines and high-performance back-end servers with a variety of processors and interconnects. We approach this goal by retaining, as far as possible, the well-known POSIX threads model currently in use on SMPs.

Stampede offers cluster-wide threads with optional loose temporal synchrony, and consistent distributed shared objects. A higher-level sharing/ communication mechanism called *Space-Time Memory*, with automatic garbage collection, is particularly suited to the complex buffer management that arises in real-time analysis hierarchies based on video and audio input. In this paper, we describe an example of our target class of applications, and describe features of Stampede that support cluster-based implementations of such applications.

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

There is an emerging class of applications that are computationally very demanding, but which have many features different from the scientific/ engineering applications that have traditionally driven research in parallel processing. An example of this class is a future "Smart Kiosk" for public spaces [14, 11]. It is computationally demanding because it employs sophisticated vision, speech and learning algorithms to track people in front of the kiosk, to recognize them, to gauge facial expressions, gaze and gestures, and to understand their queries. The kiosk's responses may involve sophisticated 3-d graphics, animation and synthesized speech. Being interactive, it must perform these recognition tasks and generate and render responses at sufficient speed to hold up a convincing "conversation". The structure and demands of the computation are dynamic, depending on the current state of the interaction, if any. Such applications are often based on codes originally written in C. If they have been parallelized, it is often for an explicitly parallel SMP model such as POSIX threads.

The computing platform for a kiosk, or for multiple kiosks scattered throughout an airport or railway station, can be quite heterogeneous. The kiosks may contain front-end computers for low-level vision, speech and rendering tasks, while sharing one or more back-end servers for more compute power, for databases, for high-speed Internet access, for maintenance, *etc.* These computers may have different processor architectures and operating systems, different numbers of processors, and interconnection networks of uneven capability.

There is a significant programming difficulty for this application and platform scenario. The dynamic structure and complex sharing patterns of the application by themselves make it difficult to use the message-passing programming model (such as MPI). The dynamic application structure, together with the heterogeneity of the platform makes it infeasible to use a flat/ transparent shared memory programming model.

Stampede is our solution to this programming problem. We refer to the heterogeneous platforms described above as "clusters". Stampede offers cluster-wide threads with optional loose temporal synchrony, and consistent distributed shared objects. A higher-level sharing/ communication mechanism called *Space-Time Memory*, with automatic garbage collection, is particularly suited to the complex buffer management that arises in interactive applications with analysis hierarchies based on video and audio input [12]. One of our general design philosophies is to retain, as far as possible, the traditional POSIX threads paradigm for parallel processing on a single SMP.

In this paper, we describe the Smart Kiosk application in more detail, we describe

---

Address for Nikhil, Rehg, Joerg and Kontothanassis: Digital Equipment Corporation, Cambridge Research Laboratory, One Kendall Square, Bldg. 700, Cambridge MA 02139, USA. Email: {nikhil,rehg,cfj,kthanasi}@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 reported in this paper was done at CRL.)

the features of Stampede that make it suitable for such applications on heterogeneous platforms, and conclude with a description of the current status and plans (we have built a prototype and have begun to run the vision component of the Smart Kiosk on it).

2 The Smart Kiosk: an example target application

The goal of CRL's Smart Kiosk project [3] is to develop a kiosk for public spaces– such as a store, museum, or airport– that interacts with people in a natural, intuitive fashion. A Smart Kiosk may contain a variety of input and output devices: video cameras, microphones, loudspeakers, touch screens, infrared and ultrasonic sensors, *etc.* Two or more cameras may be used to produce stereo images of the scene before the kiosk. Microphone arrays accept stereo speech input from customers. Computer vision techniques are used to track, identify and recognize one or more customers in the scene. The kiosk may initiate and conduct conversations with customers. Recognition of customer gestures and speech may be used for customer input. Synthetic emotive speaking faces and sophisticated graphics, in addition to Web-based information displays, may be used for the kiosk's responses.

We believe that the Smart Kiosk has features that are typical of many emerging scalable applications, including robots, smart vehicles, 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.

Figure 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 (*i.e.,* 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.

The parallel structure of this application 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 application is "buffer management". 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:

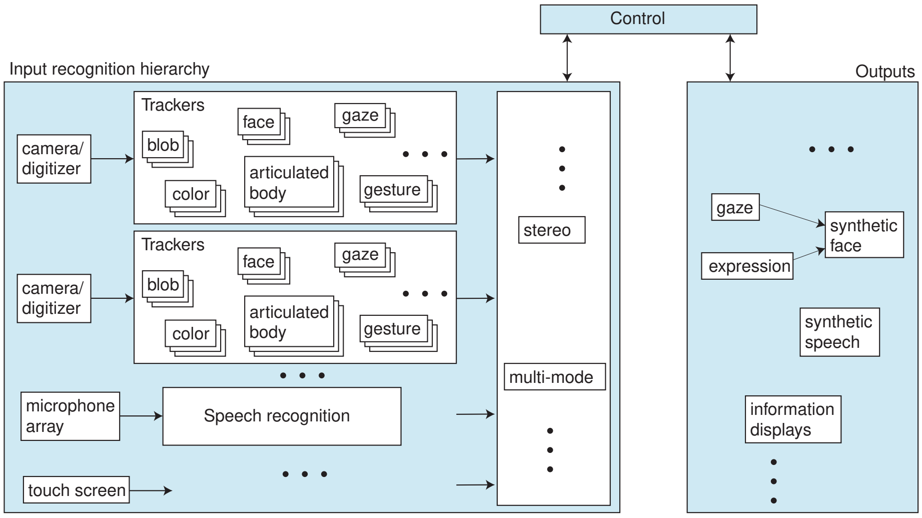

Figure 1: Software architecture of the Smart Kiosk

- The datasets become temporally sparser and sparser, because they correspond to higher- and higher-level hypotheses of interesting events. For exampe, the lowest-level event may be: "a new camera frame has been captured", whereas a higher-level event may be: "John has just pointed at the bottom-left of the screen". Nevertheless, we need to keep track of the "time of the hypothesis" because of the interactive nature of the application.

- Threads may not access their input datasets in a strict stream-like manner. In the interests of conducting a convincing real-time conversation with a human a thread may prefer to receive the "latest" input item available, skipping earlier items. The conversation may even result in cancelling activities initiated earlier, so that they no longer need their input data items.

- 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. Higher-level hypotheses may be generated multi-modally, *i.e.,* by combining vision, audio, gestures and touch-screen inputs.

- Newly created threads may have to re-analyze earlier data. For example, when a thread hypothesizes human presence, this may create a new thread 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.

These algorithmic features bring up two requirements. First, data items must be meaningfully associated with time and, second, there must be some discipline of time, in order to allow reclamation of storage for data items (garbage collection).

Even a single kiosk is computationally demanding (vision, speech, graphics) and scalable (tracking multiple customers and conducting multiple conversations); in addition, multiple kiosks may be installed in a facility, sharing back-end servers for additional compute power, models (color histograms, face models, articulated body models, ...), databases, high-speed Internet access, *etc.*.

The design of Stampede is aimed at making it easier to program such applications on such platforms. An equally important goal is portability, to allow flexibility in the choice of in-kiosk computers, back-end servers, and their interconnection networks.

3 Overview of Stampede

Figure 2 shows an overview of the Stampede programming model. The control model

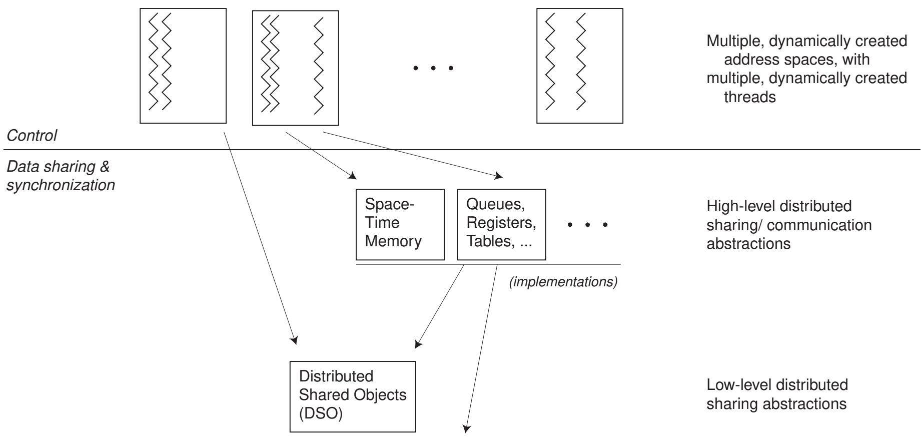

Figure 2: Overview of the Stampede cluster programming system

includes an unlimited number of dynamically created *threads* running in an unlimited number of dynamically created *Address Spaces*. Stampede's threads are an extension of POSIX threads for multiple address spaces.

All threads within an address space can share data using ordinary shared memory (for example, C global static data, malloc'd data, *etc.*). Threads across all address spaces can share data using consistent *Distributed Shared Objects (DSO)*, described in Section 6. DSO is similar to the Midway shared memory system [2], but with a substantially different programmer interface.

Threads across all address spaces can also share/ communicate data using higher-level distributed data structures, the most novel of which is *Space-Time Memory (STM)*, described in Section 5. STM is particularly useful for managing temporally indexed collections of data, as found in the analysis hierarchies of the Smart Kiosk. The figure also illustrates that STM and the other higher-level data structures can be implemented using DSO, or directly using lower-level "raw" communication mechanisms.

Stampede is currently based entirely on C library calls, *i.e.,* it is implemented as a run-time system, with calls from standard C. Many aspects of the calls could be simplified, prettified, hidden completely, or made more robust (with type-checking), by designing language extensions or a new language. Our initial interest is in proving the concepts and quickly bringing up the Smart Kiosk application, whose existing components are written in C. We have some ideas for high-level descriptions of dynamic thread and communication structures (such as those in Figure 1) from which we can automatically compile the actual thread creation and Space-Time Memory calls.

4 Address Spaces and Threads

We chose to make multiple Address Spaces (AS's) visible to the application programmer because we believe that, for our target environment, it is infeasible realistically to provide the illusion of a single, shared address space. In the Smart Kiosk, for example, the application may be split between a front-end machine on the kiosk and one or more back-end servers located in a machine room, and these machines may have different processors and operating systems. In addition, the Smart Kiosk application contains a mixture of components, some written in C and some written in Tcl/Tk. The latter components are not thread-safe, and need to be jacketed in their own address space if we are to avoid a major porting job.

The number of Address Spaces has no direct correlation with the number of physical machines or processors in the system. An Address Space must be contained completely within a single machine (which may be an SMP), and there can be more than one Address Space on a machine. An Address Space stays on the machine on which it is created– it cannot migrate. Address spaces may be created dynamically, although we expect this to be very infrequent (only for dynamically created thread-unsafe computations).

Stampede threads are based on the POSIX "pthreads" model [6]. Execution begins with a single thread at an application-supplied `spd_app_main(argc,argv)` routine. Through recursive thread creation, an application can create an arbitrary number of threads. A Stampede thread always runs entirely within an address space, and does not migrate, once created. Because we are supporting arbitrary C code and libraries, which can involve pointers into the stack, OS-provided handles, *etc.*, migration would be extremely difficult and expensive (if not impossible).

Stampede's `spd_thread_create()` call extends POSIX's `pthread_create()` with a few extra parameters. One of them is an integer that specifies which address space the child thread should run in. This number can be in the range 0 to `(spd_num_ASs-1)`, where `spd_num_ASs` is a Stampede-provided variable equal to the current number of address spaces. Alternatively, a special wild-card argument allows the Stampede runtime system to choose one of the existing address spaces for this thread; this choice may depend, for example, on the current loads on the participating machines. The semantics of thread creation are the same as in POSIX: the parent thread blocks on the creation call until the child thread has been created and is ready to run, no matter which address space it occupies.

Stampede's argument-passing convention during thread creation differs from the POSIX model, because the parent and child threads may be on different address spaces. POSIX thread creation passes only a "one word" argument (coerced to the `(void *)` type) from the parent thread to the root function of the child thread. Larger arguments are passed by reference, by passing a pointer to the real argument in this one word argument. This is adequate in POSIX since threads occupy a single address space. We have found that a simple extension subsumes the POSIX system, with very little intellectual or performance overhead. Stampede thread creation takes an additional integer `arg_size` parameter. When `arg_size` is zero, the usual `(void *)` parameter is passed exactly as in POSIX. When `arg_size` > 0, the `(void *)` parameter is interpreted as a pointer to `arg_size` bytes. These bytes are copied to the destination address space, and the child receives a `(void *)` pointer to this copy. For uniformity, this copy is performed even if the child and parent are on the same address space (so, the child never has to synchronize with the parent to access the copy).

The thread-creation call returns a Stampede thread identifier that is unique across all address spaces in the application. Thread identifiers may be used for thread control and synchronization. For example, if a thread A must wait for another thread B to complete, whether or not they are on the same address space, it can call Stampede's analog to POSIX's `pthread_join()`, supplying the Stampede thread identifier for B.

In summary, in order to simplify porting of existing applications to Stampede, we have sought to retain the POSIX threads model as far as possible, making only the minimal changes necessary in order to extend it to multiple address spaces.

5 Space-Time Memory

Perhaps the most novel aspect 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. To recap the description in Section 2, there are four complicating features: streams become temporally sparser as we move up the analysis hierarchy; threads may not access items in strict stream order; threads may combine streams using temporal correlation, and the hierarchy itself is dynamic, involving newly created threads that may re-examine earlier data.

Traditional data structures such as streams, queues and lists are not sufficiently expressive to handle these features. In addition to the issue of associating data items with time, these features also make garbage collection a challenging problem.

Stampede's Space-Time Memory (STM) is our solution to this problem. The key construct in STM is the *port*, which is a location-transparent collection of objects indexed by time. The API has operations dynamically to create a port, and for a thread to *attach* and *detach* a port. Each attachment is known as a *connection*, and a thread may have multiple connections to the same port. Figure 3 shows an overview of how ports are used. A thread can *put* a data item into a port *via* a given output connection using the call:

```
spd_port_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 port cannot have more than one item with the same timestamp, but there is no constraint that items be put into the port in increasing or contiguous timestamp order. Indeed, to increase throughput, a module may contain replicated threads that pull items from a common

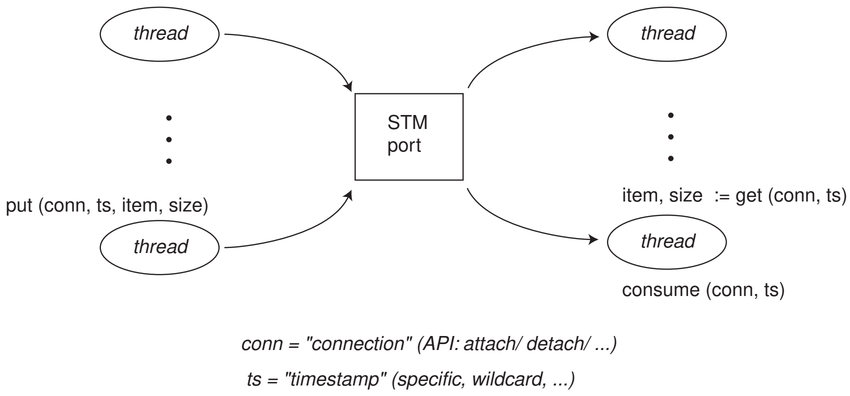

Figure 3: Overview of Stampede ports

input port, process them, and put items into a common output port. Depending on the relative speed of the threads and the particular events they recognize, it may happen that items are placed into the output port "out of order". Ports can be created to hold a bounded or unbounded number of items. The `put` call takes an additional flag that allows it to block or to return immediately with an error code, if a bounded output port is full.

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

```
spd_port_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 the newest/oldest value currently in the port, or the newest value not previously gotten over any connection, *etc.*. 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 port is a distributed data structure and multiple threads on multiple address spaces may simultaneously be performing operations on the port, 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 port 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 a DSO "global pointer" (described in Section 6) or, if the application exploits knowledge about address spaces, it can even be an ordinary C pointer.

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.

5.1 Garbage Collection in STM

The question of garbage collection of items in ports is difficult, in light of the fact that a thread may `get` and `put` items sparsely, and even out of order, and the fact that Stampede threads may fork new threads that revisit old data. Stampede imposes rules on thread times and generation of item timestamps that make garbage collection feasible.

An object X in a port is in one of three states with respect to each input connection ic connecting that port to some thread. Initially, X is "unseen". If the thread performs a `get` operation on X over connection ic, then X is in the "open" state with respect to ic. Finally, the thread can perform a `consume` operation on the object, transitioning it to the "consumed" state. We also say that an item is "unconsumed" if it is unseen or open.

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.

Every thread has a variable called its "virtual time". At each point in time, each thread has a "virtual time lower bound", which is the lesser of:

- its own virtual time, and
- the smallest timestamp of all unconsumed objects in ports to which the thread has input connections (this number of course may vary as new items are put into those ports by other threads).

A thread can change its virtual time to any specific value $\geq$ this lower bound. Alternatively, a thread can set its own virtual time to the special value INFINITY, in which case its virtual time lower bound is determined purely by what is available on its input ports. This strategy is typically adopted by threads that just compute output item timestamps based on input item timestamps.

When a thread `put`'s an object into a port *via* an output connection, it can specify any timestamp $\geq$ its virtual time lower bound (subject, of course, to the normal restriction that two objects in a port cannot have the same timestamp).

Similarly, when a thread creates a new child thread, the parent can specify the child's initial virtual time, using an extra argument in the `spd_thread_create()` call described in Section 4, to any time $\geq$ the parent's virtual time lower bound.

These rules transitively imply a *global* lower bound timestamp ts_{min}, which is the global minimum of:

- virtual times of all the threads, and

- timestamps of all unconsumed items on all input connections of all ports.

It is impossible for any current thread, or any subsequently created thread, ever to refer to an object with timestamp $< ts_{min}$. Thus, all objects in all ports 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.

Although this general-purpose global lower-bound computation eventually picks up all garbage in all ports, there is a common case that accelerates garbage collection. Frequently, a producer thread knows exactly how many consumer threads will consume each item (which may be different from the number of input connections to the port). This information can be passed to Stampede in the form of an additional *reference count* parameter in the `put` call. As soon as that item has been consumed the requisite number of times, Stampede can garbage collect it immediately.

The copy-in/copy-out semantics allows Stampede to reclaim *all* the space used internally in ports. However, since an item passed through STM may contain references to other application data structures that are unknown to Stampede, Stampede invokes a user-supplied cleanup handler before finally disposing of the item. This "upcall" is always done in the context of the thread that originally `put` that item into the port (it is piggy-backed on to other Stampede calls performed by that thread), because that thread is best suited to interpret the contents of the item.

5.2 Communicating Complex Data Structures through STM

The `put` and `get` mechanisms described above are adequate for communicating contiguously allocated objects through ports, but what about linked data structures? In the Smart Kiosk, for example, an image data structure consists of one object containing the pixel data, and a chain of dynamically computed "image attribute objects" attached to the main object using C pointers; however, an image and its attributes are, conceptually, a single unit that we wish to communicate through an STM port. The C pointers are of course meaningless in a different address space.

To solve this, Stampede extends the basic STM system with a notion of "object types". The following call:

```
spd_dcl_type (type, flatten_method, unflatten_method, ...)
```

declares a new object type (represented by an integer), and associates with it a set of methods, or procedures. Two of these are for flattening and unflattening objects of this type into a contiguous sequence of bytes for transmission between address spaces.

A variant of the port `put` call takes a pointer to the data structure, as before, but it now takes the type as a parameter instead of the object size (which is not particularly meaningful for a linked data structure). Similarly, a variant of the `get` call now returns a pointer to the linked data structure, and its type. Figure 4 shows an overview of how these facilities are used. Stampede takes care of the flattening, communication and unflattening necessary to reconstitute the linked data structure for the consumer. These actions are done lazily, *i.e.,* only when a consumer actually attempts to `get` an item, and the flattened bits are cached and communicated at most once between any two address spaces. The normal garbage collection process, described in the previous section, also recycles buffers containing flattened bits.

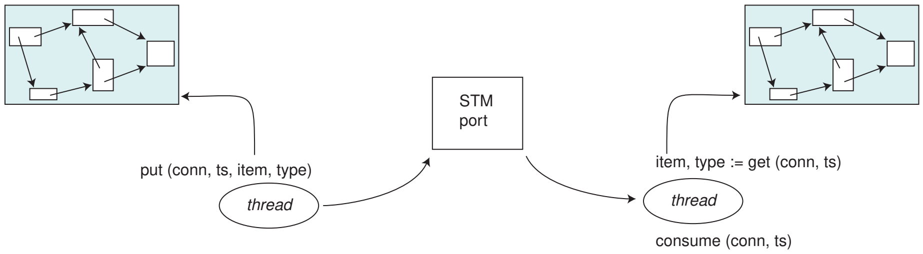

Figure 4: Communicating complex objects through ports, based on "types"

If we implemented Stampede in a language with a richer type system, the application programmer could be relieved of the burden of specifying flatten and unflatten methods (similar to the "serializer" mechanisms in Java). However, even in this case, it would be useful to have the ability to override these default methods. For example, image data structures in the Smart Kiosk application include a linked list of attributes which can, in fact, be recomputed from the object during unflattening, and so do not need to be transmitted at all. Further, the image data itself can be compressed during flattening and decompressed during unflattening. Such application- and type-specific generalizations of "flattening" and "unflattening" cannot be provided automatically in the default methods.

5.3 Synchronizing 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 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 [13]. 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, a thread in the Smart Kiosk at the bottom of the analysis hierarchy can pace itself to grab images from a camera and put them into an output port at 30 frames per second, using absolute frame numbers as timestamps.

6 Cluster-wide Distributed Shared Objects (DSO)

Space-Time Memory is well suited for managing temporally indexed collections of data that are processed in a pipeline manner. But what about ordinary, shared, updatable data? Stampede provides a lower-level, "shared memory-like" mechanism called Distributed Shared Objects (DSO). This mechanism is borrowed from our earlier work on Cid [8], and is also closely related to the Midway shared memory system [2].

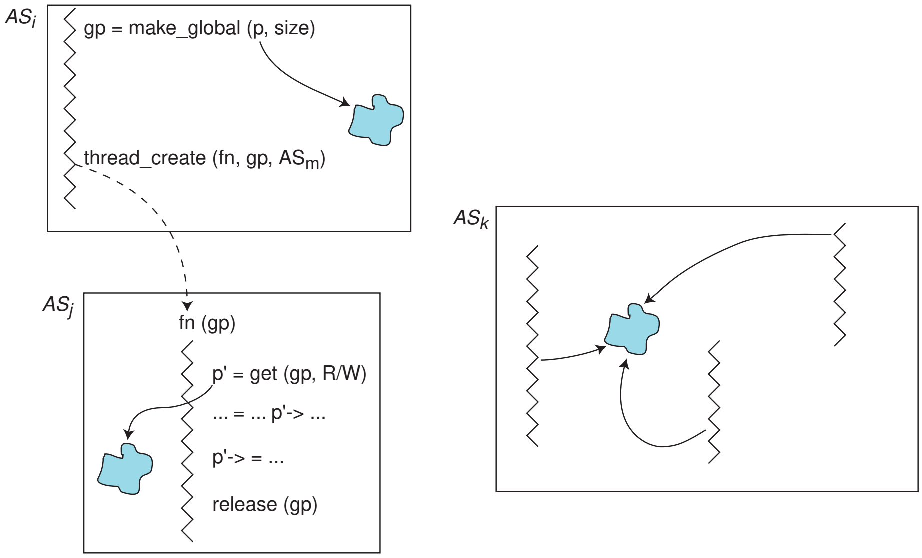

Figure 5: Overview of Stampede's Distributed Shared Objects (DSO)

Figure 5 shows an overview of DSO usage. First, a thread dynamically declares an object as a global object using the call:

```
spd_dso_gptr  gp;    void  *p;    int  size;

gp = spd_dso_make_global (p, size);
```

The returned value `gp` is an application-wide unique identifier for the object. Once declared global, all threads (including the thread that declared it global) must only access the object between `get` and `release` calls:

```
spd_dso_get (gp, mode, & p', & size, ...);

... arbitrary code to manipulate the object using p'-> ...

spd_dso_release (gp, ...);
```

In the `get` call, the thread specifies the desired object using `gp`, and the desired access `mode` in which to obtain the object, such as `READ` (shared) or `WRITE` (exclusive). The `get` call returns an ordinary C pointer to the object (`p'`) and the object's size. The Stampede runtime system implements, in software, a roving-owner consistency protocol to implement the access mode semantics. Each address space contains at most one copy of the object (shared by all threads in that address space).

How does a thread "know" about a global object that may have been created by another thread? The base mechanism is that a `gp` may be passed as an argument during thread creation. Then, inductively, an object may contain other `gptr`'s as fields.

This is a different programming interface from the Midway system, with which it shares the idea that synchronization is associated with specific shared data. Midway has the traditional notions of locks and data, and the application program makes explicit calls to associate a lock with the data that it guards. This association is exploited in the consistency protocol to decide exactly what data needs to be moved to a processor that acquires a lock to enter a critical section (Midway calls this "entry consistency"). In Stampede's DSO, there is no separate notion of locks. Instead, the programmer directly thinks in terms of shared objects, to which a thread at various times has exclusive, shared, or no access. Unlike flat transparent shared memory systems, DSO does not perform a "check-for-miss" or global-to-local address translation on every memory reference; essentially, this is done once, during the `get` call, which transforms the global name `gp` to a local name `p'`. Subsequent accesses to the object, prior to the `release` call, are just ordinary pointer dereferences, at full speed. The actual addresses `p'` at which an object is replicated by the protocol may vary across different `get`'s. This also makes it easy for the object manager on an address space to evict objects that are not currently in use, and to reuse the freed storage for other objects. When the application no longer needs a DSO object `gp`, it can call `spd_dso_free(gp)` on any address space; the protocol consistently frees all replicas and calls a user-supplied free() routine on the address space where it was originally made global.

In addition to the usual `READ` and `WRITE` modes, Stampede's DSO design includes other modes such as `RECENT_COPY`, `PRODUCER` and `CONSUMER`. The former is useful when the application is resilient to accessing a perhaps stale (but consistent) copy of the object, and the latter modes are useful when two threads access an object in the producer-consumer idiom.

Stampede also has an asynchronous variant of the `get` call. This can be used to "prefetch" an object and also to initiate concurrent `get`'s for multiple objects, instead of obtaining them serially. The constructs for these split-phase transactions originated in dataflow languages [1], and were subsequently used in languages like Split-C [4] and Cid [8].

Finally, DSO also supports the distributed sharing of linked data structures, just like the system described for STM in Section 5.2, using type-specific flatten and unflatten methods. These methods are called automatically, and lazily, by the consistent replication protocol. The cacheing and management of the buffers for the flattened bits for an object are a little more complicated in DSO than in STM because of their different semantics: STM has copy-in/ copy-out semantics, whereas DSO objects are truly shared and updatable.

The Stampede application programmer has a spectrum of choices in making a linked data structure available cluster-wide. At one extreme, he can have the the entire data structure moved *en masse* by hooking flatten and unflatten methods into the consistent object replication protocol. At the other extreme he can replace every C pointer by a `gptr`, and access individual elements of the data structure across the cluster at a fine grain. Or, in between, he can define *regions* of the data structure that are to be treated as single units, using `gptr`'s to link between regions, and providing flatten/unflatten methods to have regions moved as units. The choice, on this spectrum, is clearly going to depend on the application.

7 Status and Plans

Essentially all the features of Stampede described above have been implemented for clusters as of April 1998. The only pieces still missing are dynamic creation of address spaces, and the non-standard sharing modes in DSO: RECENT_COPY, PRODUCER, CONSUMER, *etc.* We are currently able to run, on a cluster, an early prototype of the compute-intensive vision component of the Smart Kiosk, using color models to track multiple targets in front of a single camera.

Earlier, this color-based tracking application and an image-based rendering application exhibited good performance and speedups on a single SMP version of Stampede. Experimental results and pseudo-code can be found in [12].

Stampede is implemented as a C library under Digital Unix. Our main back-end compute server is a cluster of four AlphaServer 4100's, each being an SMP with four 400 MHz Alpha processors and 1.5 GB main memory. The SMPs are interconnected with Digital's Memory Channel, Myricom's Myrinet, and an 100 Mb/s FDDI ring. Memory Channel is an extremely low-latency "protected remote write" cluster interconnect [5]. Stampede runs on each of these, and indeed runs on any mix of Alpha Digital Unix workstations and SMPs, resorting to UDP sockets when no better interconnect is available. The Stampede system uses CRL's CLF substrate [9] which provides basic cluster services such as process startup and standard I/O, debugging, and high-speed communication. We cannot yet run on different processor architectures and operating systems, but we do have near-term plans to port it to Windows NT on Alpha and x86 machines.

On an experimental basis, Stampede also incorporates the Cashmere Distributed Shared Memory (DSM) system [7] as an alternative to DSO for ordinary shared data. While the rest of Stampede is very portable (it can even work on workstations over UDP sockets), Cashmere is quite closely tied to Digital's Memory Channel. Thus, we view this as an experimental feature to allow us to compare the costs of data-sharing over DSM and DSO. If DSM is found to be a valuable component of Stampede, we can consider either porting Cashmere to be independent of Memory Channel, or replacing it with some other portable DSM system.

We also have three separate implementations of Space-Time Memory (STM): on top of Cashmere, on top of DSO, and a direct implementation using CLF messaging. Again, this is an experimental setup to allow us to compare the costs of communication and sharing in these three implementations.

In the coming months, we expect to make the system more robust, and then to conduct performance studies to understand the behavior of the system under various choices: the relative performance of Space-Time Memory over its three implementations; the relative performance of ordinary data sharing over DSO and DSM; the effects of thread placement, *etc.* We will of course be tuning and optimizing the implementation continuously.

A related project already underway is to study the integration of dynamic task and data parallelism in Stampede [10]. Many opportunities for data parallelism exist in the Smart Kiosk. For example, images can be partitioned into regions and processed by parallel threads, with each thread looking for all color models in a region. Alternatively, the color models can be partitioned, with each thread looking at entire images for a single color model. Stampede currently has task parallelism only (thread creation), but it is sufficiently flexible to enable manual construction of data parallel structures. However, the book-keeping necessary to split datasets into data parallel chunks and then to recombine the results, can be quite onerous. We have many ideas for higher-level support for data parallelism, but first we intend to conduct some experiments using manually constructed data parallelism to understand where it is most effective.

Further out, we will also be expanding the application on Stampede from the current one-camera vision algorithm towards a full Smart Kiosk system, including stereo vision, more sophisticated vision algorithms, speech recognition and other sensor technologies. As this evolution happens, we expect Stampede's focus to shift towards issues of dynamic thread creation, load balancing, *etc.*

8 Conclusion

There is an emerging class of "smart" applications that monitor a variety of sensors; perform sophisticated, computationally demanding "recognition" algorithms involving individual sensors and combined information from multiple sensors; and, have real-time constraints in that they must react to events in the real-world. The platforms for these applications may combine low power front-end machines together with powerful back-end servers. We have described one such application, CRL's Smart Kiosk, but the description could equally well fit robots, autonomously navigating vehicles, interactive animation for entertainment and training, *etc.*

We have described Stampede, a portable programming system for such applications and platforms, that we are building at CRL. Stampede has dynamic threads that can share data uniformly across multiple distributed address spaces. A key novel feature of Stampede is Space-Time Memory, which permits these applications easily to manage time-sensitive data in the presence of real-time constraints and dynamic thread structure.

Acknowledgements: We would like to thank Kath Knobe for detailed comments that improved this paper substantially. Kath, Jamey Hicks, David Panariti and Mark Tuttle have also been excellent sounding boards for ideas during our design discussions.

References

[1] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[2] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the IEEE CompCon Conference*, 1993. Also CMU Technical Report CMU-CS-93-119.

[3] 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.

[4] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. v. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proc. Supercomputing 93, Portland, Oregon*, November 1993.

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

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

[7] 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.

[8] 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.

[9] R. S. Nikhil and D. Panariti. *CLF*: A common Cluster Language Framework for Parallel Cluster-based Programming Languages. Technical Report (forthcoming), Digital Equipment Corporation, Cambridge Research Laboratory, 1998.

[10] J. M. Rehg, K. Knobe, U. Ramachandran, and R. S. Nikhil. Integrated Task and Data Parallelism for Dynamic Applications. In *LCR98: Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers, Carnegie Mellon University, Pittsburgh, PA, USA*, May 28–30 1998.

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

[12] 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, 1997.

[13] A. Singla, U. Ramachandran, and J. Hodgins. Temporal Notions of Syncrhonization and Consistency in Beehive. In *Proc. 9th An. ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, June 1997.

[14] 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*. Cambridge University Press, 1998. In press.

digital™

Stampede
A programming system for emerging scalable interactive multimedia applications

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

CRL 98/1
May 1998