The Design and Implementation of Concurrent Input/Output Facilities in ACT++ 2.0

Dennis Kafura and Manibrata Mukherji

TR 92-46

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

August 23, 1992

The Design and Implementation of Concurrent Input/Output Facilities in ACT++ 2.0

By

Dennis Kafura and Manibrata Mukherji

Virginia Tech

kafura@cs.vt.edu

ACT++ 2.0 is the most recent version of a class library for concurrent programming in C++. Programs in ACT++ consist of a collection of active objects called actors. Actors execute concurrently and cooperate by sending request and reply messages. An agent, termed the behavior of an actor, is responsible for processing a single request message and for specifying a replacement behavior which processes the next available request message. One of the salient features of ACT++ is its realization of I/O as an actor operation. A special type of actor, called an interface actor, provides a high level interface for a file. Interface actors are sent request messages whenever I/O is necessary and can also transparently perform asynchronous I/O. ACT++ has been implemented on the Sequent Symmetry multiprocessor using the PRESTO threads package.

1.0 Introduction

Structuring applications as a collection of objects that communicate and cooperate with each other to solve a single problem is argued to be a natural way of modeling the real world and of capturing the concurrency of operations that exist among the entities of the real world. The objects which model the real world entities can be thought of as autonomous agents which compute independently of each other and communicate with each other via messages. To exploit the concurrent behavior of the objects comprising an application it is imperative to use a programming language that enables one to specify and control concurrent objects.

While C++ [Stroustrup 86] is a widely used language, computation in a C++ program is carried out by a single thread of control and the sender of a message blocks until the requested operation is completed. Given the advances made in the realm of computers having multiple processors and the development of distributed programming paradigms, the latter limitation of C++ not only inhibits the user from exploiting the natural concurrency in the problem domain but also fails to exploit the physical parallelism of computations that can be achieved in order to enhance efficiency.

Our goal in designing ACT++ was to provide the ability to specify and control concurrent objects in C++. Instead of inventing a new model of concurrent computation we decided to implement some features of the Actor model of concurrent computation [Agha 86, Agha and Hewitt 87]. [Kafura and Lee 90] describes the design issues and the implementation details of an earlier version of ACT++. This paper describes ACT++ 2.0 (henceforth just ACT++), the latest version which enhances the earlier one in three respects. First, we have used the PRESTO [Bershad et al 88, Bershad 90] threads package as the underlying system to control concurrent objects. PRESTO was used because it is written in C++ and provides a high-level abstraction for concurrent programming using "lightweight" processes called *threads*. Second, we have implemented a concurrency control mechanism called *behavior sets* [Lavender and Kafura 90] which can be used to tackle the conflict between the inheritance mechanism of object-oriented languages and the specification of synchronization constraints. Third, we have introduced special features using which concurrent input/output (henceforth I/O for short) - simultaneous I/O requests from multiple, simultaneously executing threads of control - can be handled correctly and efficiently.

In this paper we will discuss in detail the design issues and the implementation details of the concurrent I/O facilities implemented in ACT++. In the remainder of the paper, first, we introduce the salient features of the model of computation implemented in ACT++. Then we discuss the problems posed by concurrency, UNIX, and the choice of PRESTO in the design of the I/O system. Then we discuss how those problems have been resolved in ACT++. We conclude the paper by stating the current status of our work and directions of future research.

2.0 The ACT++ Model of Computation

In the following we summarize the salient features of the model of computation implemented in ACT++.

- An ACT++ program consists of a collection of actors which execute concurrently using independent threads of control.

- Actors communicate asynchronously via messages called *request* messages. Each actor has the ability to buffer messages in a queue called its *mail queue*. Actors can also participate in synchronous communications with objects that are not actors.

- Request messages contain requests for the execution of program segments in an actor called *behavior scripts*. Behavior scripts are implemented as the methods of a special type of object called the behavior object (also referred to as the behavior). Each behavior is responsible for processing exactly one request message.

- The behavior object that will process an actor's next request message is called its *current behavior*. At any point in time only a single behavior object is designated as the current behavior of an actor. *The execution of a specific method in the behavior object of an actor is regarded as the processing of a message by that actor.*

- The processing of a request message by a behavior of an actor could lead to the occurrence of one or all of the following actions:

- send asynchronous request messages to itself or other actors,
- synchronously invoke methods in itself or in other non-actor objects,
- create more actor or non-actor objects, and
- specify the next behavior of the actor (called a *replacement behavior*) that will process the next request message.

- Actors may also exchange *reply* messages. As a result of a request message to it, an actor may send a reply message to the requestor with the result of the computation. A special type of object called a Cbox is used to buffer reply messages. Cboxes are defined with blocking semantics so that an actor blocks when it tries to extract a reply message from an empty Cbox.

- Replacement behaviors of an actor in ACT++ are not restricted to process request messages in the strict order of their arrival. Instead, a replacement behavior can consult the values of its instance variables to decide which request message to process next.

3.0 Some Problems for I/O in a Concurrent Environment

This section considers three general problems pertaining to I/O in a concurrent environment in which multiple threads of control execute simultaneously. These problems are:

- the interference between concurrent sequences of I/O operations directed at the same file,

- the blocking effects of low-level I/O system calls in UNIX, and

- the consistency of the interpretation of I/O commands executed in different process contexts.

The first of these problems is inherent to concurrent computation and simply another instance of the general problem of interference among concurrent activities over their access to shared resources. The second problem is due to our choice of implementing ACT++ on UNIX. The third problem arises due to the fact that the scheduling mechanism in PRESTO, in the preemptive mode, maintains a balanced load on the different processes by assigning a ready *thread* to run on any idle process. As a result, a *thread* can execute on different processes in its lifetime and since file descriptors do not retain their meaning across different process contexts, *thread* migration becomes an obstacle to concurrent I/O.

3.1 The Problem of Interference

Interference arises if multiple concurrently executing processes perform I/O to or from the same destination. UNIX ensures individual I/O calls are non-preemptive, that is, if I/O is initiated by a process on a file descriptor, the corresponding file table entry will remain locked until the I/O is complete. But in between system calls no locking is available. As a result, interleaved executions of the I/O calls might lead to different results in different executions of the same program. For example, in the case of writing to a terminal, multiple concurrent writes might cause overlapped and incomplete display of information depending on the relative speeds and the volume of I/O performed by the processes.

Two solutions to the interference problem is to do I/O from a *critical section* or use an *I/O server* to do all I/O. Neither solution is elegant and does not reduce the burden on the user. In the case of using critical sections, the user has to explicitly manage the locks used to implement each critical section. Moreover, in programming environments like ACT++ which do not provide locks at the user level, this solution cannot be implemented. In the case of a server, the user must both define the server and ensure that the server does not become a performance bottleneck. Given the choice between doing I/O using critical sections and using an I/O server, an I/O server is more attractive because a server encapsulates all low level operations and provides high level abstractions for the user.

A useful and efficient language feature for concurrent I/O would allow a user to create an I/O server for each file and then control the sharing of a server between the different threads of control running the application. The creation of one server for each file reduces the possibility of the server becoming a bottleneck and also reduces the complexity of an individual server since it has to manage I/O to a single file only.

3.2 The Problem of Blocking I/O Calls

The current version of ACT++ is implemented on the UNIX operating system. As a result, the basic mechanisms for doing I/O are those available in UNIX. Among the I/O features available in UNIX are a set of system calls including *open*, *close*, *read*, and *write*, which perform I/O using a unique identifier called a *file descriptor*. File descriptors are unique identifiers which act as an internal representation of the standard files or the special devices with which they are associated. Other system calls, like *fcntl* and *ioctl*, can be used to modify the status of file descriptors. Only unformatted raw byte I/O is possible through the read and write system calls - no low level facility for formatted I/O is available.

The problem with the *read* and *write* system calls is that they are blocking calls. This means that if the I/O is not possible immediately when a call is made, the process making the I/O call will block. When the I/O is possible, the operating system will unblock the process allowing it to complete the I/O. In an application running on multiple processes which does not use a central server for doing concurrent I/O, every process may block as a result of unsatisfied I/O calls thereby defeating the purpose of using multiple processes altogether. The latter situation might also cause a real-time application to miss important events in the external world. In an environment, like ACT++, which uses *threads* instead of processes, a blocking call executed by a *thread* blocks the process on which it was executing. This hurts other *threads* which are ready and could have executed on the blocked process. The execution of a blocking I/O call by a *thread* on each process might block all the processes and yet there could be many ready *threads* waiting to execute.

To avoid blocking, UNIX provides non-blocking, **asynchronous** I/O facilities for terminals and sockets only. To perform asynchronous I/O the user must

- write a signal handler for the SIGIO signal that the operating system delivers to a process when the I/O is ready,
- mark the file descriptor for asynchronous I/O by using a special option of the fcntl system call, and
- identify the process or process group to which the SIGIO signal must be delivered.

A language feature which hides all of the above details for doing asynchronous I/O will certainly be easier to use.

3.3 The Problem of Consistency of File Descriptors

Another problem for concurrent I/O in a multi-process application is the consistency of file descriptors across processes. Because it is an index to a process specific table called the *file descriptor table*, a file descriptor is meaningful only in the context of the process in which it was created. But PRESTO on the Symmetry assumes that all objects are created in the shared memory which makes them accessible from every process. Therefore, PRESTO implements the most efficient scheduling mechanism for load balancing in such an environment and allows a *thread* to execute in the context of different processes during its lifetime, if more than one process is being used to run the application. As a result, a file descriptor obtained by a *thread* while running on one process is rendered useless when the *thread* executes on another process. Therefore, the run time system must be enhanced to ensure that a *thread* that executes an *open* call on a particular process is scheduled to run on the same process throughout its lifetime.

4.0 Concurrent I/O Facilities in ACT++

The problems of concurrent I/O discussed above have been resolved in ACT++ as follows.

- Interference is avoided by creating an I/O server for each file which serializes all I/O to that file. The ACT++ class library contains the definition of the file server objects.
- Blocking is prevented by embellishing the I/O servers with the ability to do asynchronous I/O to terminals only. All work to set up a file for asynchronous I/O is done transparently by the I/O server.

- Consistency of file descriptors is maintained by enhancing PRESTO with the capability of *thread binding* - the ability to constrain a *thread* to run on a specific process throughout its lifetime.

In the following sections we will discuss the design issues and the implementation of each of the above solutions.

4.1 Interface Actors - the File Specific I/O Servers

The file-specific I/O servers which handle I/O requests in an ACT++ application are called interface actors (henceforth IA). As the name signifies, IAs are actors themselves. This implies that I/O operations are performed via messages to an IA instead of direct system calls. The `IActor` class in ACT++ implements IAs. It is a subclass of the `Actor` class which implements all user-defined actors in an application. A partial definition of the `IActor` constructor is as follows.

```
IActor::IActor(char* fname, File_Beh* init_beh, char* name) : ((Behavior*) init_beh, 1, name)
        { int fd = init_beh->Open(fname);}
```

The first argument to `IActor` is the name of the file for which the IA will act as a server. The second argument is a pointer to the initial behavior object. The third argument could be used to assign a name to the IA. The assignment is optional; a null name is assigned by default.

The initializer list associated with the `IActor` constructor is used to initialize the arguments of the `Actor` constructor. Significant among them is the second argument which specifies the number of *threads* that can be active simultaneously inside an IA. The default value of 1 limits the number of *threads* executing inside an IA at any point in time to one. This ensures that an IA serializes all I/O requests to a file because with a limit of one *thread* it will never process the next request message until the current request message has been serviced completely. This also ensures that multiple I/O requests are not activated simultaneously on the same file descriptor which would result in unnecessary blocking of *threads.*

The behaviors of an IA are instantiations of the `File_Beh` class which is defined as follows.

```
class File_Beh : public Behavior     {
        int fd;
        int oldflags;
public:
        File_Beh();
        int Open(char* fname);
        void Read(Rbox* r, int nbytes);
        void Write(Wbox* w, int nbytes);     }
```

The `File_Beh` constructor sets the `fd` data member to -1 and the `oldflags` data member to 0. The `fd` data member stores the file descriptor of the device for which the IA is the server. Since file descriptors start from 0, a -1 indicates that no valid file descriptor is stored in `fd`. The `oldflags` data member is used to store the status a file descriptor has when it is created. It is used to restore the status of the descriptor when the IA is deleted.

The initial behavior passed to the `IActor` constructor is an incomplete object which is not ready to handle I/O requests. It is the invocation of the `Open` method in it from the `IActor` constructor which assigns valid information to the private data members thereby making it ready for I/O processing. This way of constructing an "interface actor-behavior" pair is different from the construction process of an ordinary "actor-behavior" pair. In the latter case, all the data members are properly initialized and the behavior is ready to process a message before it is passed to the actor constructor.

The reason behind this deviation is related to the proper handling of concurrent I/O in an application running on multiple processes. Although I/O in the case of multiple processes is discussed in a later section, we will discuss the rationale behind the "interface actor-behavior" construction now. The construction of an initial `File_Beh` object and the construction of an IA are two independent activities that are executed by the same physical *thread*. In a single process application, the *thread* is guaranteed to execute on the same process. But, in a multi-process preemptive run-time environment, even though each construction phase can be made non-interruptible, the *thread* might be preempted between the two construction phases. Thus, if such an interruption occurred, the file descriptor could be obtained by the `File_Beh` constructor on process A and the IA could be constructed on process B. The execution of the constructors on two different processes is not of any concern except for the fact that *threads* created in the IA must execute on the process on which the file descriptor was obtained. Therefore some information pertaining to *thread* binding must be recorded when the file descriptor is created. This information can be maintained either in the behavior object or in the IA. If maintained in the behavior object the data has to be explicitly passed from a behavior to its replacement. Moreover, when a *thread* is created in an IA the *thread* binding information must be obtained by invoking methods in the replacement behavior. Hence, for the sake of efficiency, a design decision was made that *thread* binding information will be maintained in the IA. As a result, the construction of the `File_Beh` object is completed as a part of the IA construction process causing all *threads* which execute request messages to an IA to run on the process on which the IA was constructed.

The following are the most important actions taken in the `Open` method.

- The *open* system call is invoked to obtain the file descriptor corresponding to the file name argument `fname` supplied by the user.

- The signal handler for the SIGIO signal is specified. This is the signal that the operating system sends when I/O is ready, i.e. when the I/O operation on the file can be completed without blocking. The specification is done using the *sigvec* system call.

- The current status flags associated with the file descriptor `fd` is obtained by using the `F_GETFL` option of the *fcntl* system call and this status is recorded in the `oldflags` data member.

- The file descriptor `fd` is enabled for both asynchronous I/O and I/O without delay. It is done by setting the `FASYNC` flag which indicates that asynchronous I/O can be done on the file and by setting the `FNDELAY` flag which indicates that a *read* or a *write* system call will never wait for the file to become ready for I/O when the calls are made. With the `FNDELAY` flag set, an I/O call returning with a zero value indicates that the file is not ready for I/O and asynchronous I/O must be performed. To set the new status, the *fcntl* system call is invoked using the `F_SETFL` option and providing the flag settings as arguments.

- The current process is marked as the recipient of the SIGIO signal. This is an important step because it informs the operating system to which process to deliver the SIGIO signal for a file on which asynchronous I/O is being performed. The *fcntl* system call is used once again, this time with the `F_SETOWN` option. The process id of the current process is obtained using the *getpid* system call and it is passed as an argument to *fcntl*.

4.2 The Asynchronous I/O Manager

As mentioned before, the blocking on an I/O call is prevented in ACT++ by embellishing the I/O servers with the capability of doing asynchronous I/O. This feature has been realized by a special object called the asynchronous I/O manager (henceforth AIOM). Every IA communicates with the AIOM in order to perform an I/O operation. There is only one AIOM object per process in an ACT++ application. It manages the low-level aspects of doing I/O, keeps track of all the file descriptors that have been used by IAs to do asynchronous I/O in the context of that process, determines which file descriptor is ready for I/O when a SIGIO signal occurs, and acts as an intermediary between the application and the operating system.

AIOM objects are instantiations of the `AsyncIoMgr` class. A partial definition of the class follows.

```
class AsyncIoMgr  {
      int fdtablesize;
      fd_set io_mask;
      Io_queue* io_pend_list; ...    }
```

The `fdtablesize` data member records the size of the file descriptor table of the process with which the AIOM is associated. This member is used when the set of file descriptors is searched to find out which ones are ready for I/O. The `io_mask` data member records the file descriptors that have been marked for asynchronous I/O. It is set to zero initially. `Fd_set` is a system defined type that is used to declare bit masks corresponding to the bit masks used by the system to manipulate file descriptors. The `io_pend_list` data member is a pointer to an instantiation of the `Io_queue` class. The queue stores information by which the AIOM informs waiting IAs that file descriptors are ready.

Every IA invokes the `init_async_io` method in the local AIOM in order to perform an I/O request. From `init_async_io`, the request may be satisfied synchronously or asynchronously. If the I/O is not satisfied synchronously then the file descriptor is prepared to do asynchronous I/O. In the latter case the I/O completes with the invocation of the `propagate_signal` method in the local AIOM. In the following we will discuss the function of these two methods.

4.2.1 The init_async_io Method

The arguments passed to the `init_async_io` method are

- the file descriptor on which I/O is requested,
- the type of I/O request that is being made (read or write),
- the source or destination of the data (depending on the type of the request) that is involved in the operation,
- a Cbox pointer to which a reply message is sent to inform the IA that the descriptor is ready in the case of asynchronous I/O, and
- the number of bytes of data involved in the I/O operation.

The major actions in `init_async_io` are the following.

- Depending on the type of request, either the *read* or *write* system call is invoked.
- If the call returns more than zero bytes, then the file was ready and the request has been satisfied and control returns from the method.
- If the system call returns zero then the file is not ready for I/O and the following preparation for asynchronous I/O is made before returning from the method.
    - The bit corresponding to the current file descriptor in the the `io_mask` data member is set,
    - an object is created using the current file descriptor and the Cbox pointer, and
    - this object is enqueued in the `io_pend_list` queue.

The implementation of the `init_async_io` method is further complicated by its sensitivity to interruptions. There are three reasons for not interrupting `init_async_io`. The first and most important reason is the possibility of deadlock. If `init_async_io` is processing a file descriptor that is not ready for I/O and the SIGIO interruption occurs after the *read* or *write* system call is over but before the file descriptor is enqueued to `io_pend_list`, then deadlock is imminent. This is because, after the *read/write* call is executed for a file descriptor, I/O must be initiated by the SIGIO signal for that descriptor. Since the relevant information could not be enqueued when the *thread* was interrupted, the signal handler will not be able to send a reply message to the IA waiting on that file descriptor. Deadlock may be avoided because after the signal handler executes, the *thread* executing `init_async_io` will be resumed and the information will be enqueued. If the signal handler is invoked subsequently due to some other I/O event, then `io_pend_list` will be searched again and a reply message will be sent to the IA corresponding to the file descriptor that was skipped last time. But the possibility of deadlock remains. Second, if interrupted before the *read* or *write* calls then, although the file may be ready, the I/O will be unnecessarily delayed. Third, if the file is not ready then the file descriptor should be enqueued in the `io_pending_list` queue immediately to prevent unnecessary delay in sending a reply message when the descriptor does become ready.

One way in which the *thread* executing `init_async_io` can be interrupted is by the ACT++ scheduler. To prevent that we use the `thisthread->nonpreemptable()` call at the very beginning of the method. This ensures that the *thread* will not be preempted by the scheduler. `Thisthread` is a pointer to a *thread* that is maintained by each process in PRESTO to remember the current *thread* that is executing on the process. Before returning from `init_async_io` the `preemtable()` method is invoked in `thisthread` to restore the *thread* as preemptable.

To address the deadlock issue a call is made to the *sigblock* function at the beginning of `init_async_io` to block the SIGIO signal. Blocking a signal means that if the signal occurs during the period in which it is blocked, then it is held by the system and is delivered to the process when the signal is enabled once again. Before returning from the method the SIGIO signal is unblocked.

4.2.2 The propagate_signal Method

The `propagate_signal` method is responsible for the proper notification of the IAs that are waiting for an I/O request to complete. This method is invoked by the SIGIO signal handler in the AIOM corresponding to the process which is supposed to receive the SIGIO signal.

The first major event in `propagate_signal` is to make the *select* system call. The *select* system call is used to determine which file descriptors are ready for I/O. Two copies of `io_mask` are used as arguments to *select. Select* tests for readiness each file descriptor for which the corresponding bit is set in the masks. After testing all such file descriptors in each mask, *select* returns, in place, a mask of those descriptors which are ready for I/O. The mask corresponding to the second argument is set for all the file descriptors that are ready for reading and the mask corresponding to the third argument is set for all the file descriptors that are ready for writing. Note that since an IA serializes I/O requests, no file descriptor can be ready for both reading and writing and so the same bit will not be set in both the returned masks.

Select returns the number of file descriptors that are ready for I/O. If no descriptors are ready then control is returned immediately from `propagate_signal`. Otherwise the `io_pend_list` queue is scanned to determine which file descriptors in the queue are ready for I/O. For each file descriptor extracted from the queue, it is determined using the `FD_ISSET` macro whether the bit corresponding to this file descriptor is set in either of the masks returned by *select*. If set, the file is ready for I/O. Therefore, the corresponding IA is sent a reply message. Then the bit corresponding to the descriptor is cleared in the `io_mask` indicating that the request has been serviced.

4.3 Reading and Writing Using an Interface Actor

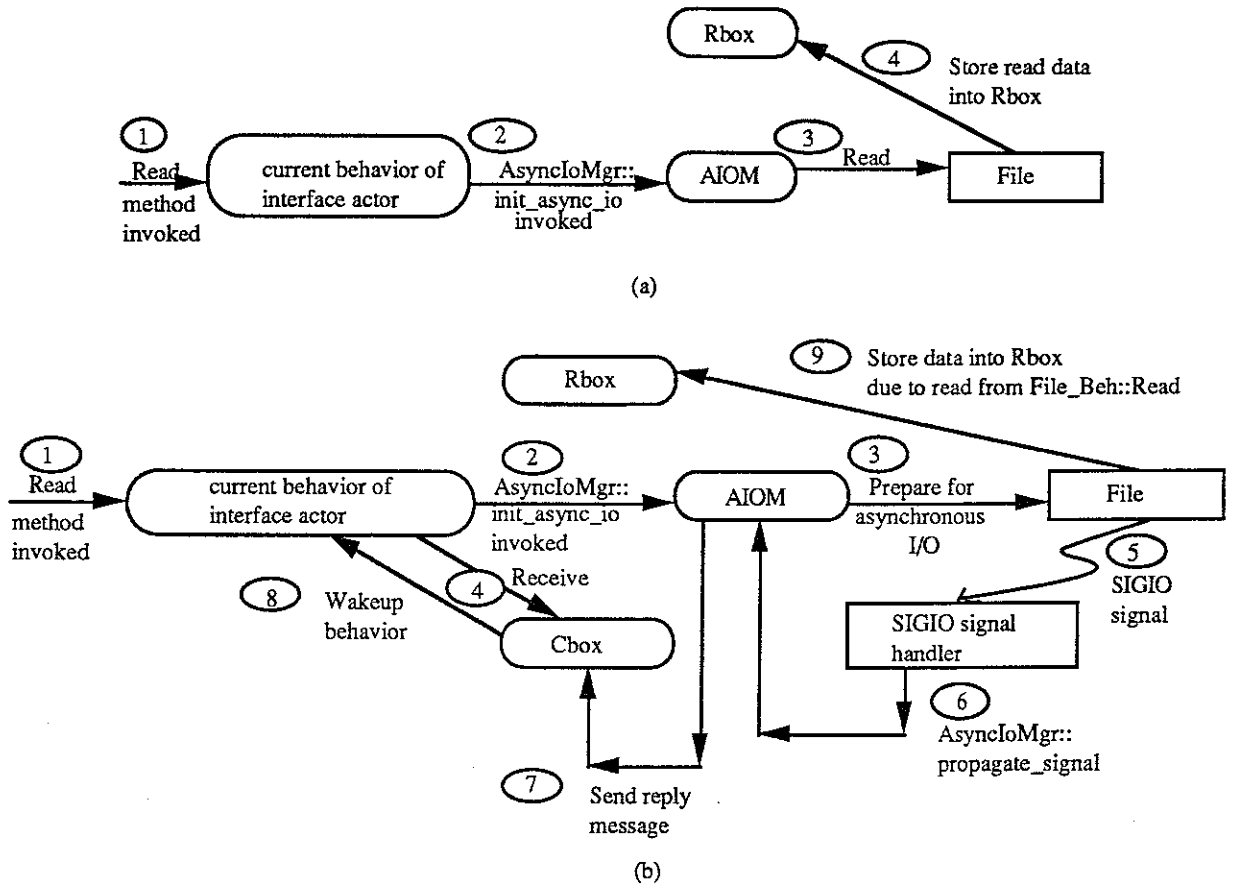
To read from or write to a file, read or write request messages, respectively, are sent to the corresponding IA. The read request message invokes the *Read* method and the write request message invokes the *Write* method in the current behavior of the IA.

An object called a Rbox is used to store the data read by the *Read* method. Another object called a Wbox is used to store the information that is to be written by *Write*. Pointers to a Rbox and a Wbox are sent to *Read* and *Write* respectively. Both of these objects are character buffers extended with blocking semantics, that is, *threads* can block on these objects if no data has been read into a Rbox or the data in a Wbox has not been written.

The sequence of operations involved in a synchronous and an asynchronous read operation is shown in Figure 1(a) and 1(b) respectively. The operations in the case of a write are similar and are not considered separately.

The read request message invokes the `Read` method in the current behavior of the IA (step 1 in Fig. 1(a)). A Rbox pointer and the number of bytes to be read are passed as arguments to `Read`. In the `Read` method, first, a Cbox is created. This is the Cbox on which the current *thread* will block if the read request cannot be satisfied. After that, the `init_async_io` method is invoked in the AIOM (step 2) for the process on which the *thread* is executing. Among many arguments, the Rbox is passed to `init_async_io`. The *read* system call is invoked from `init_async_io` (step 3) with the Rbox as an argument and since this is the synchronous read case, the call is satisfied and the data is read into the Rbox (step 4).

If the I/O cannot be completed synchronously in `init_async_io` then, the file descriptor is prepared for asynchronous I/O (step 3 in Fig. 1(b)). Then control is returned to `Read` with an indication that asynchronous I/O has to be performed. This causes the *thread* executing `Read` to block on the Cbox by invoking the `receive` operation (step 4) in it.



Rbox
4
Store read data into Rbox
1
Read method invoked
current behavior of interface actor
2
AsyncIoMgr:: init_async_io invoked
AIOM
3
Read
File
(a)
9
Store data into Rbox due to read from File_Beh::Read
Rbox
1
Read method invoked
current behavior of interface actor
2
AsyncIoMgr:: init_async_io invoked
AIOM
3
Prepare for asynchronous I/O
File
5
SIGIO signal
SIGIO signal handler
4
Receive
8
Wakeup behavior
Cbox
6
AsyncIoMgr:: propagate_signal
7
Send reply message
(b)

Figure 1: (a) The sequence of operations involved in a synchronous read operation. (b) The sequence of operations involved in an asynchronous read operation.

When the file is ready for I/O, a SIGIO signal is sent (step 5) by the system which in turn invokes the SIGIO signal handler. The signal handler invokes the `propagate_signal` method (step 6) in the AIOM. After doing some file descriptor related operations, `propagate_signal` sends a reply message to the Cbox (step 7) on which the *thread* executing the `Read` method blocked. This awakens the *thread* (step 8) which then executes the *read* system call to read the data from the device (step 9) into the Rbox.

5.0 Handling Concurrent I/O in a Multi-Process Application

The most important issue of concurrent I/O in a multi-process application is that of file descriptor consistency across processes. The solution to this problem in ACT++ is to selectively bind *threads* to processes. The specific *threads* that are bound to processes are those which execute the `Read` and the `Write` methods in the behaviors of an IA. This is because these methods use the file descriptor for the file for which the IA is the server and as such must be executed on the process on which the IA constructor was executed.

Since PRESTO does not support selective binding of *threads* to processes we have implemented that in PRESTO. To implement selective *thread* binding, the following features had to be incorporated in PRESTO.

- The ability to mark a *thread* which executes the Read or Write method in an IA in order to keep a record of the process on which it must be executed.
- The addition of a process specific queue in the scheduler along with the existing queue (the common queue) for each process that runs the application.
- A modification of the *thread* queueing and dequeueing mechanisms in the scheduler to take into account the presence of the process specific queues.

The introduction of the *thread* binding feature caused the following enhancements to be made.

- Modification of the VTALRM signal handler in PRESTO to take into account the process specific queues in the scheduler.
- Modification of the Actor and IActor classes in ACT++ to use the *thread* binding features.

In the following sections we will discuss the implementation of each of the above extensions.

5.1 Implementing Thread Marking

To implement *thread* marking, the `Thread` class in PRESTO has been subclassed. Named `New_Thread`, the class represents the *threads* which execute an ACT++ application and which can be bound to specific processes, if necessary. The subclass is defined below.

```
class New_Thread : public Thread      {
protected:
      int t_iactor;
      Process* t_runonlyon;
      int t_processq;
public:
      New_Thread(?);
      int get_proc()                  {return (int)t_runonlyon;}
      int set_proc(Process* p)        {t_runonlyon = p;}
      void set_iactor()               {t_iactor = 1;}
      void set_processq(int i)        {t_processq = i;}
      int get_processq()              {return t_processq;}
      int test_iactor();      }

int New_Thread::test_iactor()
{
      if (t_iactor)       return 1;
      else                return 0;    }
```

The `t_iactor` data member is used to record that a *thread* has been created inside the behavior of an IA (henceforth such a *thread* will be referred to as an IA *thread*). The `t_runonlyon` data member is used to record the process object on which the *thread* must always execute. The `t_processq` data member is used to store the position of the process specific queue in which this *thread* will be enqueued when ready to execute. The methods in the `New_Thread` class operate on these three data members in order to set, reset, or return them and are self explanatory.

5.2 Implementing Process Specific Queues

To implement the process specific queues and the associated operations we replaced the scheduler object in PRESTO by a new scheduler object which gives priority to IA *threads*. The new scheduler object is an instantiation of the `Prty_Scheduler` class in ACT++ which is defined as follows.

```
class Prty_Scheduler : public Scheduler    {
protected:
      ThreadPool* psc_t_ready[NUMPROCS];
      int thread_pres[NUMPROCS];
      virtual int get_prty_qptr(int i);
      virtual void set_thread_pres(int);
      virtual void reset_thread_pres(int);
public:
      Prty_Scheduler(int numschedulers, int quantum = DEFQUANTUM);
      virtual ~Prty_Scheduler();
      virtual Thread* getreadythread();
      virtual void resume(Thread* t);
      virtual int invoke();
      virtual int readyqlen();
      virtual int Scheduler_qlen();
      virtual int process_qlen(int);      }
```

Among the data members of the `Prty_Scheduler` class, `psc_t_ready` is an array of pointers to queues. There can be as many queues as there are processors in the system (recorded by the NUMPOCS macro in PRESTO). Each element of the array stores a pointer to a queue holding ready IA *threads* which will execute only on a specific process - `psc_t_ready[i]` is the pointer to the ready queue for the process object `i`.

The `thread_pres` data member in `Prty_Scheduler` is an array of integers that stores the status of the IA *thread* queues. If `thread_pres[i]` is set to 1 then the queue pointed to by `psc_t_ready[i]` has at least one ready *thread* in it; if set to 0 the `ith` queue is empty.

The `get_prty_qptr` method is used to return the pointer to the queue in the scheduler for a particular process. The `set_thread_pres` and `reset_thread_pres` methods are used to set and reset, respectively, the element in the `thread_pres` array corresponding to a particular process.

5.3 The New Thread Queueing and Dequeueing Mechanisms

Introduction of the process specific queues caused the *thread* queueing and dequeueing mechanisms to be altered. The `getreadythread` and the `resume` methods implement the new schemes.

The `Prty_scheduler::getreadythread` is the specialization of the `getreadythread` method in the `Scheduler` class which is used to dequeue a ready *thread* from the scheduler object. It is defined as follows.

```
Thread* Prty_Scheduler::getreadythread()
{
      int i = (int) thisproc;
      int k = get_prty_qptr(i);
      Thread* t;
      if (k == -1)       {
            cerr << "\n -1 RETURNED BY GET_PRTY_QPTR IN GETREADYTHREAD \n";
            this->abort(SIGKILL);
            kill(getpid(), SIGILL);
            //NOT REACHED
      }
      if (psc_t_ready[k]->size() != 0)
      {
            t = psc_t_ready[k]->get();
            decr_thread_pres(k);
            sc_lock->lock();
            (void) busybits(1);
            sc_lock->unlock();
      } else
            t=Scheduler::getreadythread();
      return t;    }
```

UNIX processes running a PRESTO application are represented as objects which are instantiations of the `Process` class in PRESTO. PRESTO creates as many UNIX processes as there are free processors in the system and for each such process creates a process object. Each UNIX process created in PRESTO has a variable in private memory called `thisproc` which stores a pointer to the process object that represents it. The above method is invoked by a special *thread* called the *scheduler thread* of which there is one per process object. Whenever a *thread* finishes execution, control is returned to the *scheduler thread* which then extracts the next available ready *thread* from the scheduler. The functionality of the above method is the same as `Scheduler::getreadythread` except that code has been added to search the process specific queues before searching the common queue. That is achieved by first converting the value of the `thisproc` pointer of the free process into an integer and then invoking `get_prty_qptr` using that integer value as an argument. `Get_prty_qptr` returns the position of the IA *thread* queue pointer in `psc_t_ready` corresponding to the free process. If the returned value is a -1 then it is a fatal error and the PRESTO run time system is aborted. Otherwise, if the size of the IA *thread* queue of the free process is not zero, then, a ready IA *thread* is extracted. Then the `busybits` method is invoked inside a critical section to set the bit corresponding to the current process in a mask that represents the busy processes in the application and a pointer to the extracted *thread* is returned.
If the IA *thread* queue has no ready *threads* the common queue is searched by invoking `Scheduler::getreadythread`. Then, whatever is returned by `Scheduler::getreadythread` is returned. It is not necessary to access the `psc_t_ready` queue in a critical section because there is a unique queue for each process, only one *thread* executes on a process object at any time, and a scheduler *thread* is non-preemptable. Yet, many *threads* might try to invoke `busybits` simultaneously from different processes. Hence it is invoked in a critical section.

Searching the process specific queues before searching the general queue implicitly assigns a higher priority to IA *threads* over ordinary *threads*. This prioritizing is reasonable because it ensures that I/O activities, which are not processor intensive, will always be attended to first in an ACT++ application.

The `resume` method in `Prty_Scheduler` is used to enqueue a *thread* that has just been created or was blocked and has become ready for execution once again. The method is defined as follows.

```
void Prty_Scheduler::resume(Thread* t)
{
       int index;
       if (t->flags()&TF_SCHEDULER)
              t->error("Can't resume a scheduler thread\n");
       t->isready();
       if (t->test_iactor())
       {
        if ((t->get_processq()) == -1)
        {
         index=get_prty_qptr(t->get_proc());
         t->set_processq(index);
        } else
             index = t->get_processq();
        psc_t_ready[index]->insert(t);
        incr_thread_pres(index);
       } else
             sc_t_ready->insert(t);   }
```

The above method is a specialization of the `resume` method in the `Scheduler` class. Since a *scheduler thread* is never preempted, it must never be enqueued in the scheduler. That is tested by the first if-statement. If the *thread* is not a *scheduler thread* then it is marked ready for execution. Then it is determined whether the *thread* is an IA *thread* by invoking the `test_iactor` method in `t`. `Test_iactor` returns a 1 if the `t_iactor` data member is set; a 0 if not. If a 1 is returned then `t` must be enqueued in a process specific queue instead of the common queue. If a 0 is returned then the *thread* is inserted in the common queue by invoking the `insert` method in `sc_t_ready`.

To determine on which process specific queue to enqueue `t`, the `get_processq` method is invoked in `t`. If `t` has just been created then the queue pointer is not yet recorded in the *thread*. `Get_processq` indicates this by returning a -1. If a -1 is not returned, `index` records the location of the IA *thread* queue pointer for the *thread*. Otherwise, the `get_proc` method is invoked in `t` to obtain the process on which `t` must execute. Then, `get_prty_qptr` is invoked with the integer value of the process pointer as the argument. This invocation returns the location of the IA *thread* queue pointer in `psc_t_ready` for `t` which is recorded in it once and for all by invoking the `set_processq` method.

After the correct queue for `t` has been determined and stored in `index`, `t` is placed on the queue by invoking the `insert` method in `psc_t_ready[index]`.Then the `incr_thread_pres` method is invoked with `index` as the argument to record the presence of an IA *thread* in the queue.

5.4 The New VTALRM Signal Handler

The VTALRM signal handler in PRESTO is called `sigpreempt_alrm`. This signal handler is executed on the expiration of the allowed time quantum when PRESTO runs in a preemptive fashion. This signal handler is responsible for determining which processes are running *threads* that are preemptable, and in such a case, signalling those processes to execute the next ready *thread* in the queue of the scheduler. Note that only the process at the root of the process hierarchy that executes a PRESTO application is configured to receive the VTALRM signal. The addition of the process specific queues in the scheduler required modifications to the signal handler. The modified signal handler is shown below. Each addition is identified by a comment.

```
int sigpreempt_alrm(int sig, int code, struct sigcontext* scp)
{
   register *sp = (int*) scp->sc_sp;
   Process* p;
   Thread* t;
   int numtopreempt;
   int i;
   int numinscheduler;                            //Addition
   numtopreempt = sched->readyqlen();
   numinscheduler=sched>Scheduler_qlen();         //Addition
   numalarms++;
   double q=((double) sched->quantum())/1000.0;
   for (i = 0; numtopreempt && i <
      sched->sc_p_activeschedulers; i++)
   {
      p = sched->sc_p_procs[i];
      int r = sched->process_qlen(i);             //Addition
      if ((!r) && (!numinscheduler))
            continue;                             //Addition
      t = p->runningthread();
      if (t && t->canpreempt())
      {
            if (p==thisproc)  (void)sigpreempt_notify(sig,code,scp);
            else              kill(p->pid(), SIGPREEMPT_NOTIFY);
            numtopreempt--;
            if ((!r) && (numinscheduler > 0))
                  numinscheduler--;               //Addition
      }
   }
   return 0;      }
```

The general scheme of operation of the original handler was to determine the number of ready *threads* in the only queue of the old scheduler by invoking the `readyqlen` method in `sched`, the shared pointer to the scheduler object maintained by PRESTO. This number gave it some idea about how many process to examine for possible *thread* preemption. If there were more ready *threads* than processes, then, all the processes were examined. If there were fewer *threads* than processes then, all the processes were not examined. This was the basic idea behind the `for` loop condition in the handler. Note that the signal handler never extracts any *thread* from the queue of the scheduler object. The process objects that are signalled are responsible for extracting the *threads*. The signal handler just ensures that enough *threads* are preempted and the corresponding processes relieved so that the waiting *threads* in the scheduler are allowed to execute.

To interrogate the status of the *threads* running on the processes, the corresponding process object pointers stored in the `sc_p_procs` data member of the scheduler were used. Starting from 0, each process object pointer in `sc_p_procs` was obtained and a pointer to the *thread* running on that process was retrieved. Then it was determined whether the *thread* could be preempted by invoking the `canpreempt` method in the *thread*. If preemptable, a signal was sent to the process which would eventually result in the execution of another signal handler, `sigpreempt_notify`, which would prepare the current *thread* for preemption and enable the switchover to the *scheduler thread*. If the process being interrogated happened to be the one on which the `sigpreempt_alrm` function itself was executing, the `sigpreempt_notify` function was invoked directly instead of sending a signal.

To account for the process specific queues in the new scheduler object the handler has been modified as shown above. The modifications have resulted in the following changes to the scheduling algorithm.

- The process specific queues are searched for a ready *thread* before searching the common queue. Therefore, IA *threads* are given higher priority than non-IA *threads*.
- If a particular process specific queue is found empty only then is the common queue searched for a ready *thread*. If the common queue is empty then the *thread* on the current process is not signalled for preemption.
- An empty common queue does not mark the end of the scheduling algorithm. The algorithm terminates only after all the process specific queues have been searched.

In the modified `sigpreempt_alrm` function, using `Prty_Scheduler::readyqlen`, the `numtopreempt` variable receives the total number of ready *threads* available in the scheduler which includes both IA *threads* and *threads* in the common queue. Then the number of *threads* available in the common queue alone is determined using `Prty_Scheduler::Scheduler_qlen`. The `numinscheduler` variable is used to keep track of the number of ready *threads* in the common queue that are available for execution on processes that do not have any waiting IA *threads*. As each *thread* is scheduled for execution from the common queue, this variable is decremented. When the value of this variable reaches zero, only IA *threads* can be scheduled for execution; if a process does not have an IA *thread* in its process specific queue then it must not be signalled. `Numinscheduler` helps to determine this situation.

Inside the `for` loop, after retrieving the `i`th process object pointer from `sc_p_procs`, the `Prty_Scheduler::process_qlen` method is invoked in `sched` to determine if there are any ready IA *threads* in the queue for this process. If there is no ready IA *thread* and there are no ready *threads* in the common queue then, the process need not be signalled and hence the next iteration of the loop is executed. Note that although the common queue is found to be empty, we still have to execute the next iteration of the `for` loop and examine the next process object in `sc_p_procs` because there might be a ready IA *thread* in the queue of that process.

Finally, it is determined whether the current *thread* on the process can be preempted. If so, the process is signalled along with the following additional action. If the queue for the current process is empty and there is at least one ready *thread* in the common queue then, the `numinscheduler` variable is decremented. This is done to ensure that if the ready *thread* in the common queue is the last *thread* to be scheduled for execution on the process just signalled then, in the next iteration of the for loop, no process would be unnecessarily signalled if its IA *thread* queue is empty.

5.5 Modifying the Actor and IActor Classes

The following additions have been made to the Actor class.

```
class Actor : public Object    {
      int iactor;
      Process* my_process;
public:
      void set_iactor();
      void set_proc(Process*);        }
```

Two new data members have been added to the Actor class in connection with the correct handling of operations of IAs. The iactor data member is set when an IA is created and it is used to take special scheduling actions on the *threads* created inside the IA. In the case of IAs, the my_process data member stores a pointer to the process object which represents the process on which the *threads* created in the IA must execute. Note that these two pieces of information have to be stored inside the IA object because at the time an IA is created no *threads* are created inside it. Later, when the IA services request messages, the iactor data member is used to mark the *threads* created to execute the messages and the my_process data member is used to set the t_runonlyon data member in those *threads*.

The next addition to the Actor class involves the creation of *threads*. After creating a new *thread* in the actor if it is found that the iactor data member in the actor is set then, the t_iactor data member is set in the new *thread* by invoking the set_iactor method on the *thread*. Also, the t_runonlyon data member is set in the new *thread* by invoking the set_proc method and passing the my_process data member in the actor as the argument. The above steps mark a *thread* as an IA *thread* and ensures that the *thread* binding mechanism will work properly.

The addition to the IActor class involves the constructor as shown below.

```
IActor::IActor(char* fname, File_Beh* init_beh, char* name) :
                  ((Behavior*)init_beh, 1, name)
{
      thisthread->nonpreemptable();
      Actor::set_proc(thisproc);
      Actor::set_iactor();
      int fd = init_beh->Open(fname);
      thisthread->preemptable();     }
```

Two of the additions in the `IActor` constructor are invocations to the `set_iactor` and `set_proc` methods in the `Actor` class. The `thisproc` pointer of the process executing the `IActor` constructor is passed as an argument to `set_proc`. This signifies that all *threads* created in the IA must execute on this process.

The invocations of the `nonpreemptable` and `preemptable` methods in `thisthread` play a very significant role. In an application running a preemptive scheduler, a *thread* executing the `IActor` constructor can be preempted at any point during its execution. After being resumed, the *thread* might execute on a different process. If such a thing happens between the invocation of the `set_proc` method and the invocation of the `Open` method then, the first process would be recorded as the site of execution of all subsequent *threads* in the actor but the file descriptor obtained in `Open` will be on the second process. To avoid this file descriptor consistency problem we mark the *thread* as nonpreemptable at the very beginning and mark it as preemptable at the very end of the constructor.

6.0 Conclusion

We have implemented the ACT++ class library for the Sequent Symmetry multiprocessor using PRESTO 0.4. This library is meant to be used with AT&T C++ version 1.2 and any other compatible compilers. We are currently porting PRESTO onto SUN 3/80 and DECstation 5000/240. Our next target is to use PRESTO 1.0 to implement ACT++. This new version of PRESTO uses AT&T C++ 2.1.

The design of the I/O system for ACT++ involved a substantial modification to the code for PRESTO. The design of the PRESTO system allowed us to make all modifications to the C++ parts of PRESTO by subclassing existing classes, for example, the `Threads` and the `Scheduler` classes. The remaining modifications involved code written in C, for example, the VTALRM signal handler. In such cases we had to insert our changes into the existing PRESTO code to achieve the desired functionality. The ability to use subclassing to extend and alter PRESTO is, we believe, both a reflection of the quality of the PRESTO design as well as an affirmation of the utility of object-oriented techniques for thread-based, concurrent programming.

An important feature of the I/O system in ACT++ is that it provides a high level abstraction for doing I/O, namely, interface actors. In many languages, for example, C and ADA, I/O is not considered to be a part of the conceptual framework of the language and is achieved through calls made to predefined library routines. On the contrary, in ACT++, I/O is achieved using the same conceptual entities - actors, behaviors, request messages, reply messages, and different types of Cboxes - which are used to perform the major computation in any ACT++ application. As a result, the ACT++ environment provides a uniform framework for computation and I/O using actors.

In order to enhance the capabilities of the I/O system, we will extend interface actors to handle typed I/O. We have introduced the capability of handling only null-terminated string I/O so that we could concentrate on the details of implementing the asynchronous I/O capabilities. In future, we will use the `ifstream` and `ofstream` classes available in the `iostream` library of C++ to handle typed I/O.

4.1 Acknowledgements

We thank Vikul Khosla for an initial design of the I/O system and Keung Hae Lee for his implementation of an earlier version of ACT++ [Lee 90].

References

[Agha 86] Agha, Gul, Actors: A Model of Concurrent Computation in Distributed Systems, MIT Press, Cambridge, MA, 1986.

[Agha and Hewitt 87] Agha, Gul, and Hewitt, Carl, "Concurrent Programming Using Actors," in Object-Oriented Concurrent Programming, A. Yonezawa and M. Tokoro (eds.), MIT Press, Cambridge, MA, 1987, pp. 37-53.

[Bershad et al 88] Bershad, B.N., Lazowska, E.D., and Levy, H.M., "PRESTO: A System for Object-Oriented Parallel Programming," Software Practice and Experience, 1988.

[Bershad 90] Bershad, B.N., "The PRESTO User's Manual," Report, Department of Computer Science, University of Washington, Seattle, Washington, 1990.

[Kafura and Lee 90] Kafura, D., and Lee, K.H., "ACT++: Building a Concurrent C++ with Actors," Journal of Object-Oriented Programming, Vol. 3, No. 1, May/June, 1990, pp. 25-37.

[Lavender and Kafura 90] Lavender, G., and Kafura, D., "Specifying and Inheriting Concurrent Behavior in an Actor-Based Object-Oriented Language," Technical Report TR 90-56, Department of Computer Science, Virginia Tech, Blacksburg, VA, 1990.

[Lee 90] Lee, K.H., Designing A Statically Typed Actor-Based Concurrent Object-Oriented Programming Language, Ph.D. Dissertation, Department of Computer Science, Virginia Tech, June 1990.

[Mukherji 92] The implementation of ACT++ on a shared memory multiprocessor, M.S. Project Report, February, 1992, Department of Computer Science, Virginia Tech.

[Stroustrup 86] Stroustrup, Bjarne, The C++ Programming Language, Addison-Wesley, Menlo Park, CA, 1986.