Pipes: Linguistic Support for Ordered Asynchronous Invocations

by

Adrian Colbrook
Eric A. Brewer
Wilson C. Hsieh
Paul Wang
William E. Weihl

April 1992

Abstract

We describe pipes, a new linguistic mechanism for sequences of ordered asynchronous procedure calls in multiprocessor systems. Pipes allow a sequence of remote invocations to be performed in order, but asynchronously with respect to the calling thread. Using pipes results in programs that are easier to understand and debug than those with explicit synchronization between asynchronous invocations.

The semantics of pipes make no assumptions about the underlying architecture, which enhances code portability. However, the implementation of pipes by the language compiler can be optimized so as to take advantage of any underlying message ordering a particular architecture may provide. Pipes also provide application-transparent flow control for asynchronous invocations and are able to throttle invocations from multiple calling threads.

We present four implementations of pipes and show that the performance and space overheads associated with pipes are low.

Keywords: linguistic mechanism, ordered asynchronous invocations, serialization, message-layer implementation, flow control.

This work was supported by the National Science Foundation under grant CCR-8716884, by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-89-J-1988 and by an equipment grant from Digital Equipment Corporation. Individual authors were supported by a Science and Engineering Research Council Postdoctoral Fellowship, an Office of Naval Research Graduate Fellowship, and National Science Foundation Graduate Fellowships.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

1 Introduction

In this paper we describe pipes, a new linguistic mechanism for sequences of ordered asynchronous procedure calls in MIMD multiprocessor systems. Pipes allow a sequence of remote invocations to be performed in order, but asynchronously with respect to the calling thread. Using pipes results in programs that are easier to understand and debug than those with explicit synchronization between asynchronous invocations.

Remote procedure calls (RPCs) have become the standard method of communication in distributed systems [8, 21, 34, 52] and in several parallel programming languages [7, 11, 37]. They use a widely understood abstraction, the procedure call, that precisely specifies the remote interface. This allows programs to be statically type checked and permits communication code to be generated automatically. However, RPCs usually prevent the caller from running in parallel with the call, since the caller must wait for a reply. This often leads to worse performance than explicit message passing. Consequently, some languages and systems have introduced asynchronous procedure calls [26, 37, 43], which allow a caller to continue to execute once its call has been sent. Mechanisms such as futures [22] and promises [35] permit callers to run in parallel with the call and later use the results of the call.

Providing an asynchronous call mechanism alone is sometimes insufficient. A number of applications require ordered delivery and execution. In particular, if call n has a side effect that ought to affect call $n+1$, then the execution of the calls should be synchronized to provide the required sequencing semantics.

For example, consider executing operations on a dictionary data structure that is stored remotely. A dictionary is a partial mapping from keys to data that supports three operations: `insert`, `delete` and `search`. A number of useful computations can be implemented in terms of dictionary abstract data types, including symbol tables and pattern-matching systems. To execute operations on the dictionary, we need to send messages to the processor storing the dictionary. Since the dictionary is remote it may be useful to be able to send operations asynchronously and access the result later. Intuitively, we want the actions associated with the messages to occur in the order that the messages are sent. Thus, if a thread sends an `insert(key k, data d)` request followed by a `search(key k)` request, we expect the insert to take place first and the search to succeed and return the data value `d` associated with the key value `k` (assuming that no other thread has changed or deleted the entry `(k,d)` in the meantime).

A further example where ordering semantics may be required is the updating of cached copies of a replicated object. In some applications, such as the B-tree described by Wang and Weihl [49, 51], updates to replicated objects can propagate to cached copies in a lazy fashion. Updates can be propagated by sending asynchronous messages, but updates to copies must be made in the correct order.

The correct ordering of operations can be achieved by invoking the operations synchronously. However, this prevents the caller from running in parallel with the call. During an invocation the caller will be delayed until it receives an acknowledgment: The acknowledgment could be sent as soon as the invocation is received and enqueued at the callee, but the caller is still delayed for at least the round-trip message delay of the network.

The round-trip message delay can be removed by performing the remote invocations asynchronously. In this case the calling thread and target object must provide the synchronization required to preserve order. Synchronization to achieve this effect can be coded explicitly in the application program; this leads to complex and less efficient programs. Our mechanism for ordered asynchronous calls allows programs to be simpler and more efficient than programs in which ordering is enforced by application-level synchronization.

Previous work on distributed system by Gifford and Glasser [20] and in the Mercury system [30] has resulted in the design of remote invocation mechanisms in which a sequence of calls between a single sender and a single receiver are run in order, but asynchronously with respect to the caller. We have adapted these ideas for use in the multiprocessor language PRELUDE [50]; our design provides integrated language support for ordered asynchronous invocations, and also generalizes the previous work by allowing calls from multiple sending threads to be ordered.

The use of a linguistic mechanism to provide ordered asynchronous invocations makes no assumptions about the underlying architecture. Pipes guarantee the correct execution order regardless of the ordering properties of the underlying architecture. This enhances portability as no changes need be made to application code when moving to a different architecture. The implementation of pipes by the compiler and runtime system can still take advantage of properties of the underlying architecture, so there is no loss in performance. In fact, implementing the synchronization at the system level as opposed to the application level results in a more efficient implementation. We show that the performance and space overheads associated with pipes are low.

Pipes can also be used to provide flow control for asynchronous invocations. Carrying out flow control in software is difficult in general, since a receiver cannot always determine who the next sender will be. Implementing flow control in application programs also increases the complexity of the code. Pipes can provide application-transparent flow control for asynchronous invocations and can throttle invocations from multiple calling threads.

In Section 2, we define precisely what it means to "order" asynchronous invocations. In Section 3, we outline the impact the underlying architecture has upon message ordering. Section 4 describes the semantics of pipes at the language level and illustrates the mechanism in an object-oriented language. We show that a mechanism for ordered asynchronous calls leads to programs that are both simpler to understand and more efficient than ones in which the ordering is enforced by application-level synchronization. In Section 5, we describe different implementations of pipes and demonstrate how an implementation can be optimized to take advantage of order-preserving networks. Performance results for each implementation are given in Section 6. Finally, in Section 7 we describe using pipes to provide application-transparent flow control.

2 Order Semantics

Simple asynchronous calls provide concurrency, but can result in programs that are hard to understand. In many situations, a process can run concurrently with a sequence of calls that it makes, but the calls themselves should run sequentially in invocation order. Synchronization to achieve this effect can be coded explicitly in the application program, but this leads to complex programs. A mechanism for ordered asynchronous calls leads to programs that are both simpler to understand and more efficient. In this section we define precisely what it means to "order" asynchronous calls.

To serve as an illustration, we introduce a simple banking system that provides `deposit` and `withdraw` operations on account objects. Each operation takes an integer argument; the number of dollars to be deposited or withdrawn in each case. Assume that a withdrawal may only be made if there are sufficient funds in the account; a negative account balance is not permitted. If we wish to perform operations on a remote account, we need to send messages to the processor storing the account. We want the actions associated with each message to occur in the correct order. Thus, if we send a "deposit" request followed by a "withdraw" request, we expect the deposit to take place first.

If a sequence of calls from a single process are to run sequentially in invocation order then the actions associated with sequence of messages must be *serializable* in the order of invocation. Two actions are serializable if all of the observable effects are consistent with one of the actions occurring entirely before the other. For example, if we perform the actions `deposit(50)` and `withdraw(25)` on a bank account, then there are three possible results:

Serializable in order: `deposit` appears to run entirely before `withdraw`, which results in a net increase of $25.

Serializable out of order: `withdraw` appears to run entirely before `deposit`, which could cause the withdrawal to fail if there are insufficient funds in the account to make the withdrawal. The net change will be +$25 normally and +$50 if the withdrawal fails.

Not Serializable: the deposit and withdrawal run in parallel, with both reading the initial balance. The net change could be -$25 or +$50 (or, in general, some more bizarre interaction), and the withdrawal may fail.

The first case has the desired semantics: the actions are serializable in the order specified by the sequence of invocations.

To ensure these semantics, a system must remember the order of invocation of the asynchronous calls and ensure that the observable effects of the calls are consistent with that order. Essentially, order must be preserved "end to end": intermediate orders, such as the order of reception at the target, need not match the order of invocation.

Pipe semantics ensure that the invocations on the pipe execute in order, with one invocation running to completion before the next invocation begins. Therefore, pipe calls are serializable in the order specified by the sequence of invocations.

In some cases it is possible to preserve the correct semantics without each invocation running to completion. If an object uses fine-grain locking to provide mutual exclusion between concurrent invocations, then an invocation can begin executing as soon as the previous invocation holds the locks it requires. The pipe and target object cooperate to determine when the next invocation can start. The protocol and interfaces required to facilitate this cooperation are an area of future consideration.

3 Architectures and Message Order

The cost of preserving order for a sequence of asynchronous calls depends greatly on the underlying architecture. Bus-based machines, for example, ensure that messages arrive in order, since the bus serializes all messages.

In general, an interconnection medium preserves message order if and only if the path between two nodes is fixed. If there are multiple paths between two nodes, then messages sent on different paths can arrive out of order. For networks, the single-path requirement is equivalent to *deterministic* routing [15, 45]. Most contemporary multiprocessors use deterministic routing [1, 5, 16, 44].

The network performance of deterministic techniques has reached a limit, under random traffic, of a stable maximum throughput of 45 to 50% of the limit set by the network bandwidth [39]. *Adaptive* routing [14, 39, 53] improves network performance under high contention by allowing messages to travel along alternate paths if the primary path is congested. Under adaptive routing, messages may arrive out of order under high congestion, since later messages may take paths that are significantly faster.

In addition to the gains in performance, adaptive routing schemes can also enhance reliability by performing fault-tolerant routing [39]. Since existing multiprocessor systems are already richly connected, an adaptive routing scheme is able to take advantage of this path redundancy. Adaptive routing is used in the CM-5 [47] and is likely to become more popular as machines scale.

If the interconnection medium preserves message order, the overhead for ensuring execution order is essentially zero. If the architecture makes no such guarantees, then it must be possible to detect and reorder messages that arrive out of order. However, even if the architecture ensures message order, some overhead is still required if the path between two high-level objects can change. For example, if an object migrates to a new node, the path from it to a particular target object will change. Conceivably, a message sent before the migration could arrive *after* a message sent after the migration, since they travel along different paths.

The obvious solution is to exploit the order properties of the network except across migration. When an object migrates, the last message sent on the old path and the first message sent on the new path must be ordered correctly. If these two messages are executed in the correct order by the target object, the rest of the messages will be also, since the network maintains their order.

A simple way to preserve order across migration is to require an acknowledgment of the last message sent on the old path. The first message along the new path is delayed until this acknowledgment is received, so the two messages clearly execute in order. Thus, if the network preserves order, the overhead is nonzero only across migrations (which should be rare compared to messages sends) and even then it is small. However, this solution requires invocations to be buffered at the caller following a migration until the acknowledgment is received. We present an implementation in Section 5 that avoids buffering at the caller.

Finally, some architectures do not ensure delivery, in which case the system must support retransmission. In this case, the cost of preserving order is likely to be low, since the system handles retransmission.

The system as a whole thus has three possible levels of overhead for preserving order:

1. If the network preserves order and the path between any two objects never changes, the cost is zero. The cost is essentially zero if the software must also support retransmission.

2. If the network preserves order and the system supports migration, then the cost is zero, except across migrations.

3. If the architecture does not preserve order, then there must be some overhead for all ordered messages.

In all cases, the overhead will be small, but the relative cost of preserving order depends on the overhead of message passing as a whole. On fine-grain multiprocessors the message send can cost only thirty cycles, [1] which makes the cost of preserving order very significant.

[1] This is a typical time to send an invocation on the J-machine [16]. Comparable times can be achieved on other fine-grain multiprocessor architectures [40].

4 Pipes: A Linguistic Mechanism

Ordered asynchronous invocations incur the overhead required to maintain order so they are more expensive than unordered invocations. Also, unordered asynchronous invocations can run in parallel with each other; this may be important in some applications. Therefore, it is desirable to provide both ordered and unordered asynchronous invocations in the language; the user can choose to use ordered invocations, with their additional overhead and constraints on concurrency, only in those situations where it is necessary. In this section we begin by describing the functionality of our pipe mechanism and then argue that a construct that provides ordering semantics for asynchronous invocations should be included in programming languages designed for use in multiprocessor systems.

4.1 The semantics of pipes

We describe a pipe construct that supports ordered asynchronous invocations in the context of an object-oriented language. We have implemented pipes as part of the PRELUDE portable parallel language [50]. PRELUDE is a statically typed object-oriented language, and the examples presented here use the PRELUDE syntax. However, a pipe construct can be included in any procedural language intended for programming multiprocessors.

We make use of parameterized type definitions (sometimes termed generic types in the literature) in the style of CLU [32]. Let the parameterized class **pipe**`[T]` denote the class of pipes to an object of type `T`. A pipe is created by the class method **pipe**`[T].new`. For example, if `x` is an object of type `account`, then invoking **pipe**`[account].new(x)` creates and returns a pipe object of type **pipe**`[account]` for ordering asynchronous method invocations to object `x`. Objects of type **pipe**`[account]` provide all methods provided by type `account`. However, the return types for these methods are promises: if `account` provides a method `withdraw(int)` **returns**`(int)`, then **pipe**`[account]` provides a method `withdraw(int)` **returns**`(`**promise**`[int])`.

The parameterized class **promise**`[T]` refers to a promise for an object of type `T`. Promises [35] are similar to futures [22], except that the value of a future can be extracted implicitly as it has the same type as the object. The value of a promise must be explicitly extracted. For an object of type **promise**`[T]`, the method `claim()` **returns**`(T)` returns the value of the promise, an object of type `T`; it waits if the promise has not been filled in.

A promise is created by an asynchronous call. For example, an asynchronous call to a procedure that normally returns a type `T` returns the type **promise**`[T]`. In PRELUDE, invocations on pipe objects return promises. However, pipe invocations could just as easily return futures. It is the notion of a place holder for the return of the asynchronous invocation that is important rather than the particular flavor

```
x: account
p: pipe[account]
y: promise[int]
z: promise[bool]

p := pipe[account].new(x)
y := p.deposit(50)
z := p.withdraw(25)
```

Figure 1: Pipe invocations on an `account` object

of the place holder used.

To perform a sequence of ordered asynchronous calls to an object, we merely perform the same sequence of calls in a synchronous manner to one of its pipes; we refer to such calls as "pipe calls". The pipe ensures that pipe calls are processed by the target object in the same order that they are sent. Abstractly, we can view a pipe of type `pipe[account]` as a forwarder that queues up all calls sent to it and returns promises of the appropriate types. Semantically, it sends the queued calls sequentially to an `account`; a call is sent to `account` only after `account` has finished processing the previous queued call. The actual implementations, described in more detail later, use several queues so that delays in the interconnection network have minimal effect on the computation.

If a calling thread is to perform a sequence of ordered asynchronous calls on an object, it must first obtain a pipe assigned to the object. This can be accomplished either by accessing an existing pipe object assigned to the object, or by creating a new pipe. The calling thread then invokes the sequence of pipe calls synchronously on the pipe object. For example, suppose two asynchronous method invocations, `deposit` and `withdraw`, are to be sent to `account` object `x`, and the invocation for `deposit` must occur before the invocation for `withdraw`. The code shown in Figure 1 accomplishes this behavior; it assumes that `deposit` returns a value of type `int` (the current balance in the `account`) and `withdraw` returns a value of type `bool` (indicating whether there were sufficient funds in the account to make the withdrawal).

The pipe `p` in Figure 1 ensures that the call to `withdraw` does not start running on `x` until the call to `deposit` has completed. The results of the calls can be obtained by the calling thread (or some other thread that obtains the promises) by claiming the promises returned by the pipe calls.

Pipe objects of type **pipe**`[T]` may need some operations other than the ones provided by objects of type `T`. For example, one useful method is `sync`, which ensures that all calls in the pipe's queue issued by the calling thread have been executed before returning. However, providing additional methods such as `sync` for **pipe**`[T]` may introduce name clashes; for example, `T` might have a method named `sync`. If the number of such useful methods is large, the name clash problem can be significant. To avoid this problem in PRELUDE, objects of type **pipe**`[T]` provide an additional method besides those provided by type `T`. This method, called `get_basepipe`, is used to extract the "underlying pipe object"; `get_basepipe` returns an object of type `basepipe`. Operations such as `sync` are performed on this basepipe object. The name clash problem still exists in this scheme, however it has been reduced so that the instantiation **pipe**`[T]` is legal as long as `T` has no method named `get_basepipe`.

There is no special syntax for pipe calls, except that the object on which the method is invoked has type **pipe**`[T]`. Note that a pipe, like any other object, can be passed on to other objects as an argument in a procedure or method invocation. Multiple objects and threads can send calls through the same pipe object. Also, there can be multiple pipe objects associated with any single object.

Pipe semantics make no guarantees about the order of execution of calls made from different threads. This depends upon the scheduling of the calling threads. However, if order is required across multiple threads, only their access to the pipe object needs to be synchronized. For example, consider the case where two concurrent threads t_0 and t_1 wish to make calls to an object `x:account`. If the semantics of the application require that a call made by t_0 (for example a `deposit`) be executed before a call made by t_1 (for example a `withdraw`) then the two threads must be synchronized. In a language with a pipe construct, t_0 and t_1 simply make synchronous calls to the same pipe object connected to `x` in the required order (t_0 followed by t_1). The two threads can execute concurrently with the invocations on `x`. To achieve the same semantics without pipe calls would require t_0 to invoke `deposit` synchronously on `x` and, when this invocation completes, t_1 invokes `withdraw`. Providing a linguistic mechanism for ordering asynchronous invocations from different objects and threads can therefore lead to increased concurrent execution.

4.2 The need for a linguistic construct

The sequencing semantics that we have described can be implemented without a language construct such as the pipe. The ordering semantics should be included within the language rather than at the application level for the following three reasons:

1. The resulting application programs can have greater concurrency.
2. The resulting application programs are simpler to write and more likely to be correct.
3. The language compiler can improve performance by taking advantage of any message ordering the underlying architecture provides without affecting the portability of applications.

The applications programmer must use explicit synchronization to order asynchronous invocations in a language that does not provide pipes. This can be achieved by synchronizing in the calling thread or the target object. For example, if a thread wishes to make two ordered asynchronous invocations on an `account` object, the code in Figure 2 performs the required synchronization in the calling thread.

```
x: account
y: promise[int]
z: promise[bool]
balance: int

y := fork x.deposit(50)
balance := y.claim()
z := fork x.withdraw(25)
```

Figure 2: Asynchronous invocations on an `account` object

In PRELUDE the reserved word **fork** proceeding an invocation expression or statement causes the invocation to be made asynchronously. The invocation `y.claim()` returns only when the `deposit` invocation has completed. If the promise `y` has not been filled when the claim is made, then the thread waits. This has exactly the same affect as executing the `deposit` invocation synchronously, so we have lost the concurrency present in Figure 1. However, it may be possible to continue thread execution between the `deposit` invocation and promise claim if the calling thread has useful work. The only restriction is that the promise `y` be claimed before the next invocation on `x`. This appears to be a satisfactory solution provided that the thread can continue execution between successive asynchronous invocations on the same object. However, if the `deposit` finishes quickly, the `withdraw` will not begin until the other work is complete.

We would like to be able to begin `withdraw` as soon as `deposit` completes so as to improve concurrency. This can be achieved by forking a new thread immediately after the `deposit` invocation. The new thread executes the promise `claim` followed by `withdraw`. Alternatively, we could simply fork a new thread that performs the `withdraw` and `deposit` invocations synchronously and allow the existing thread to continue execution. However, these solutions have drawbacks. First, new routines must be written that correspond to the threads that are forked. A separate routine is required for every ordering of invocations that is used, leading to code expansion and reduced clarity. Second, the existing thread and new thread may need to synchronize eventually, which adds to the complexity of the code. Third, if the sequence of invocations is determined dynamically by the calling thread, it may be impossible to write a separate procedure that encapsulates the sequence.

In all the solutions described so far, before the `withdraw` invocation can be made, the return value for the `deposit` invocation must be received. The execution of successive invocations on `x` will be delayed at least for the time it takes for the return value of the first invocation to reach the calling thread followed by the time it takes the next invocation to reach the target object: in total, at least a round-trip message delay if `x` is remote. This is semantically equivalent to pipe calls; a pipe call is sent to an object only after the object has finished processing the previous queued call. However, in Section 5 we show that our implementations for pipes can queue invocations at the location of the target object, which increases concurrency by overlapping the transmission of a call with the execution of previous calls.

To avoid the losses in concurrency that result from synchronization at the calling thread, we could instead decide to synchronize at the target object. To implement this approach we must have a means of preserving the invocation order. An obvious approach is to assign a sequence number to each invocation. This technique is actually used in one of our pipe implementations, which is described in detail in the next section. We also need a mechanism for ordering and queueing invocations at the target object. To implement this in Prelude we would have to introduce new methods that execute their invocations in the order given by the sequence numbers. The result is a large amount of additional complex code. Application code that makes use of a language construct for preserving order is more likely to be correct, since the resulting code is much simpler.

The implementation of pipes by the compiler and runtime system can also take advantage of order-preserving networks. However, this is not visible to the programmer, so that moving user code to an architecture that does not provide message order will not result in any changes at the source level. Introducing the pipe construct at the language level therefore enhances the portability of user code. Even if it is possible for application code to take advantage of properties of a given architecture to further improve performance, such code lacks portability.

Although we have illustrated our pipe mechanism with a very simple banking system example, our arguments still hold for more complex examples such as dictionary operations or updating copies of a replicated object. For example, operations on a remote dictionary can be performed by connecting a pipe object between every client of the dictionary and the dictionary itself. To guarantee that updates to cached copies of a replicated object are made in the correct order, we connect pipes from the base copy to each cached copy.

In this section we have shown that a language mechanism for ordered asynchronous calls leads to programs that are both simpler to understand and more efficient than ones in which the ordering is enforced by application-level synchronization. In the following sections we show how pipes may be implemented so as to maximize performance on a given architecture. We also show that the performance and space overheads associated with pipes are comparable to those for unordered asynchronous invocations.

5 Implementation

In this section we describe four implementations of pipes that we built as part of the Prelude runtime system. The implementations run on the Proteus parallel-architecture simulator [10].

In each case, a pipe is implemented as two objects, a head and a tail. Typically the head of a pipe is located on the same processor as the threads making calls on the pipe, and the tail is located on the same processor as the target object of the pipe. However, any combination of the relative locations of calling threads, head, tail and target object is permissible; all objects may also migrate.

The head and tail objects both contain queues of pending invocations. Buffering invocations at both ends of the pipe allows most of the communication delay of the interconnection network to be hidden from the computations using the pipe. This is possible since a pipe call returns as soon as the invocation is placed on the queue of the head. In the normal case, when the calling thread and head object are co-located, this does not involve any interactions with other processors, nor any network latency.

The semantics of pipe calls guarantee that the order of execution of calls from a single thread matches the order of invocation by the thread. Each implementation ensures that:

1. Pipe calls from a single thread to the head object of the pipe are ordered regardless of whether the thread and head object are co-located.

2. Pipe calls are queued in the tail object in the same order as they are queued in the head object.

3. Invocations on the target object are made in the same order as the corresponding pipe calls, regardless of whether the target object and tail object are co-located.

For every pipe call, the compiler generates code that synchronously calls the runtime system routine `pipe_send`. If the calling thread and head object are co-located, then `pipe_send` adds the pipe call to the queue in the head object before returning; if the thread and head object are on different processors then `pipe_send` sends the pipe call to the location of the head object where it is placed in the queue. In the second case, the call to `pipe_send` does not return until an acknowledgment that the invocation has been queued in the head object is received by the calling thread. This ensures that pipe calls from a single thread to the head of the pipe are ordered, regardless of whether the thread and head object are co-located.

We present two implementation of pipes for networks that do not preserve message order. Both are equally applicable to networks that preserve order. The first uses sequence numbers and the second uses synchronous messages. We then present two additional implementations of pipes for networks that preserve message order. The first is an simplified version of the sequence numbers implementation; the second is based on a new technique that uses multiple queues. In all cases we consider the normal case pipe performance, where the head and tail objects are not located on the same processor. The interesting differences between implementations involves the techniques used to ensure that the invocations are queued at the tail in the same order as they were queued at the head.

The tail object has associated with it a queue of pending invocations together with a thread that executes these invocations. The queued invocations are executed synchronously and in order by the thread associated with the tail object. Therefore, the invocations on the target object are executed in the same order as the corresponding pipe calls regardless of whether the target object and tail object are co-located.

The head and tail can migrate independently. Object migration for the head and tail is implemented by creating a new object at the destination of the migration. The original object is removed some time later. We refer to the new object as the *new head* or *new tail* of the pipe, and we refer to the original object as the *old head* or *old tail* of the pipe.

5.1 An implementation using sequence numbers

In this implementation pipe calls are sent asynchronously from the head to the tail. Each is assigned a sequence number by the head, and the tail keeps a record of the sequence number for the next call it expects to receive. When the tail receives a pipe call whose sequence number corresponds to the expected value, it adds the pipe call to the tail queue and increments the expected value for the next call. If a pipe call is received out of order then it is buffered. After a pipe call has been added to the tail queue any buffered calls are checked to see if they should now also be added to the queue.

If there are no bounds on the possible number of outstanding messages then an infinite number of distinct sequence number values is required. In practice, we place an upper bound, b, upon the number of outstanding messages. This means that we require $b + 1$ unique sequence number values. In our implementation b is a 32 bit integer. Using a 8 bit sequence number is probably sufficient for an implementation in which objects do not migrate. However, when a tail migrates, a large number of invocations may be buffered at the old tail. The head then begins sending to the new tail. The sequence number bound must be sufficiently large so as to prevent overlapping of sequence numbers for invocations from the old tail and head. An 8 bit sequence number (256 distinct sequence number values) may not be sufficient in this case. A 16 bit sequence number (64K distinct sequence number values) should be enough, and a 32 bit sequence number (4G distinct sequence number values) will clearly be adequate. Buffered messages can be stored in a variety of ways, one of the most efficient being a table as used by Clark [12]. In our implementation the queue at the head of the pipe is in fact implicit, as we simply place the pipe call on the outgoing message queue of the sending processor.

The migration facilities provided by this implementation are the most general but are also the most costly of all the implementations. The head and the tail of the pipe can be migrated independently at any time. Migration is complete when a new head (or tail) object has been created on the target processor and the tail (or head) object has been notified. When a tail migrates, the old tail may have invocations in its queue and may still be receiving invocations that were sent before the head object was notified of the migration. These are forwarded to the new tail object where they are inserted into the queue in their correct positions according to their sequence numbers. If a tail has migrated several times in a short space of time, there may be several queues forwarding pipe calls to the tail.

It is not clear that supporting a general migration scheme such as this is useful. We anticipate that object migration will be rare, since it is likely to be relatively expensive. It therefore seems unlikely that a case would arise where it was advantageous to have simultaneous migrations of both the head and tail. For this reason, we adopt a more restrictive but simpler migration scheme in our other implementations.

5.2 An implementation using synchronous messages

In order to preserve message order between the head and tail objects in this implementation, we simply send each invocation synchronously from the head to the tail. As soon as the invocation arrives at the tail an acknowledgment is sent back to the head; note that the acknowledgment is sent before the invocation is executed. Once the acknowledgment is received, the head object is free to send the next invocation. Therefore, there is at most one invocation in transit from a pipe head to its tail at any time. This approach is only useful if the round-trip message time is not the limiting factor in performance. The assumption is that the acknowledgment will normally be received by the head before a further pipe call is made. We have found this to be the case.

We send the first pipe call from the head to the tail and set a *send in progress* flag at the head. The `pipe_send` call returns in the normal fashion without waiting for an acknowledgment from the tail object. As soon as the invocation arrives at the location of the tail, an acknowledgment is sent back to the head. Any invocations on the head that occur while the *send in progress* flag is set are simply queued at the head. The acknowledgment interrupts the processor storing the head. If, when the interrupt occurs, the queue of invocations at the head is not empty, then the invocations in the queue are sent to the tail and the *send in progress* flag remains set. If the queue is empty when the interrupt occurs then the handler clears the send in progress flag.

During head migration, the new head does not begin sending invocations until all the outstanding invocations from the queue of the old head have been sent to the tail. During tail migration, the head object queues all invocations locally until an acknowledgment is received from the new tail. An acknowledgment is not sent by the new tail until all outstanding invocations have been forwarded by the old tail object. This is the simplest of all the migration schemes used in our implementations.

One drawback of this approach is the increase in message traffic; two messages are now required for each pipe call, whereas the other schemes require only one. A more serious drawback is that part of the overhead for pipe calls includes the time taken to execute the interrupt handler for the acknowledgment. In Section 6 we show that this leads to poor performance.

In this implementation, the order that invocations are received by the processor storing the tail object matches the order in which the invocations were made, even across migration. There is no need for reordering before execution.

5.3 A simplified sequence number implementation

For networks that preserve message order, we need be concerned only with the maintenance of execution order across migrations of the head or tail of the pipe. If we use sequence numbers for each call from the head to the tail then we can receive an out-of-order message only during or immediately following migration. In fact, we make the restriction that only one pipe migration (either the head or the tail but not both) can be in progress at any point in time, which greatly simplifies the problem of message reordering at the tail.

In this implementation, the order of reception at the processor storing the tail object matches the order in which the invocations were sent by the head object except across migration. The following invariant holds: out-of-order messages occur only when the path changes, and the set of out-of-order messages are in the correct order relative to each other. Thus when the last message on the old path arrives, we know that the set of (previously) out-of-order messages are the next to execute and are in the correct order. This trivial form of reordering gives us the invariant that pipe calls are queued in the tail object in the same order that they were queued in the head object.

The implementation is similar to that described in Section 5.1; the differences are the simplifications that result from the restriction placed upon migration. We associate a migration lock with the pipe; this lock is part of the head. Before a pipe head or tail can migrate, the thread performing the migration must acquire the migration lock for the pipe.

To illustrate the effect of this restriction, consider the case where the head object migrates from processor P_0 to P_1 while the tail object of the pipe is stored on processor P_2. Let each message be represented by $[x]$, where x is the sequence number of the message. During head migration the old head sends an *end of stream (EOS)* message to the tail object. The EOS message contains a sequence number and is the last message sent from the old head. Assume that the messages up to and including to $[i]$ have arrived at the tail, there are n messages in transit from the old head to the tail, and the old head sends the message $EOS[i+n+1]$. An order-preserving network guarantees that messages $[i+1]$ to $[i+n+1]$ will arrive in order. The new head on processor P_1 may now begin sending messages with sequence number $[i+n+2]$. It is possible that messages sent from processor P_1 arrive out of sequence at P_2 (i.e., before message $EOS[i+n+1]$). However, messages arriving from processor P_1 arrive in order *with respect to each other.* Out-of-order messages are buffered as before, but they are stored as a linked list. When the $EOS[i+n+1]$ message arrives the tail object performs two operations. It adds the list of messages that arrived out of order to its queue (by concatenation) and it sends an acknowledgment back to the head. The acknowledgment notifies the head that the migration is complete so that a further migration is now free to take place (the acknowledgment clears the migration lock associated with the pipe).

The migration of the tail object proceeds in a similar style. A request to migrate a pipe tail is sent first to the head object, where the migration lock for the pipe is set. The tail is then migrated. The new tail notifies the head that migration has taken place and the head sends an EOS message to the old tail. The head then begins sending invocations to the new tail. When the EOS message has been received by the old tail and all the invocations sent to it have been forwarded to the new tail, the new tail sends an acknowledgment back to the head that clears the migration lock.

5.4 An implementation using multiple tail queues

The use of sequence numbers in the previous implementation turns out to be unnecessary. Since the network preserves order, the sequence numbers are only necessary to order messages following migration. With the same restriction placed on migration as in the previous implementation, we are able to remove the sequence number overhead for migration control from normal invocations.

In this implementation a tail contains two queues for invocations. Only one of theses queues is *active* at any time; the thread executing pipe calls takes invocations from the active queue and executes them in order. The key invariant is that the order of reception *at each queue* matches the order of transmission. Thus, the messages in a queue are always in the same order as the corresponding invocations.

A migration of the head or tail causes the status of the queues at the tail to be swapped; the active queue becomes inactive, and vice versa. The head begins sending to the new active queue. This is depicted for head migration in Figure 3. Let the head be stored on processor P_0 and the tail on processor P_2. In Stage 1, no migration has taken place and the head sends invocations to the active queue of the tail. In Stage 2, the head migrates to processor P_1. The migration lock in the new head is set and the old head sends the end of stream message to the tail. The new head can begin to send invocations to the inactive queue of the tail. When these invocations arrive at the tail they are enqueued but not processed since the execution thread is processing the active queue. When the tail receives the EOS message, it takes the contents of the active queue and adds them to the front of the inactive queue, swaps the status of the queues, and the sends an acknowledgment to the new head, whose location is given in the EOS message. When the new head receives the acknowledgment, it clears the migration lock associated with the pipe, as shown in Stage 3. The EOS message ensures that the switch between queues is made at the correct time; the switch is analogous to concatenating the out-of-order messages onto the tail queue in the simplified sequence-number implementation.

The migration of the tail object proceeds in a similar style, as shown in Figure 4. Again, let the

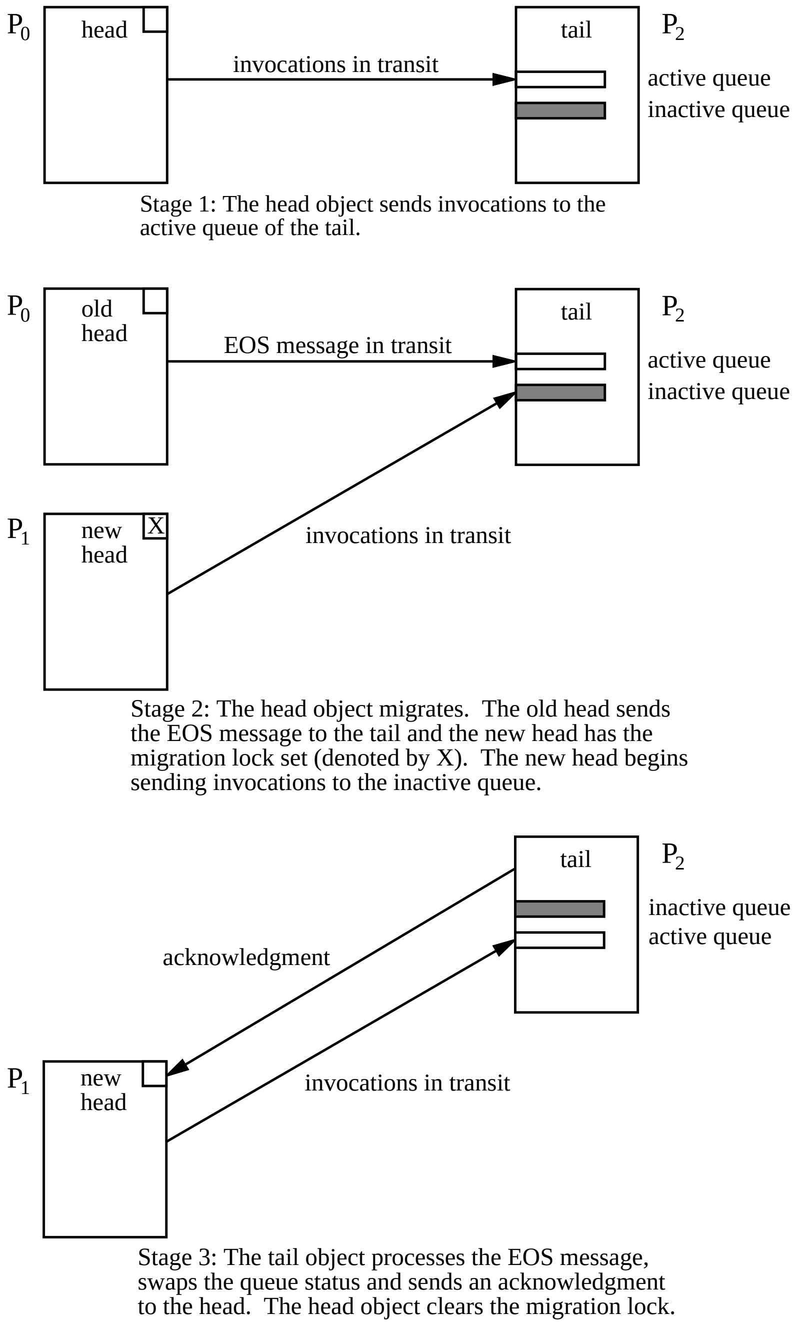

Figure 3: Head migration using multiple queues

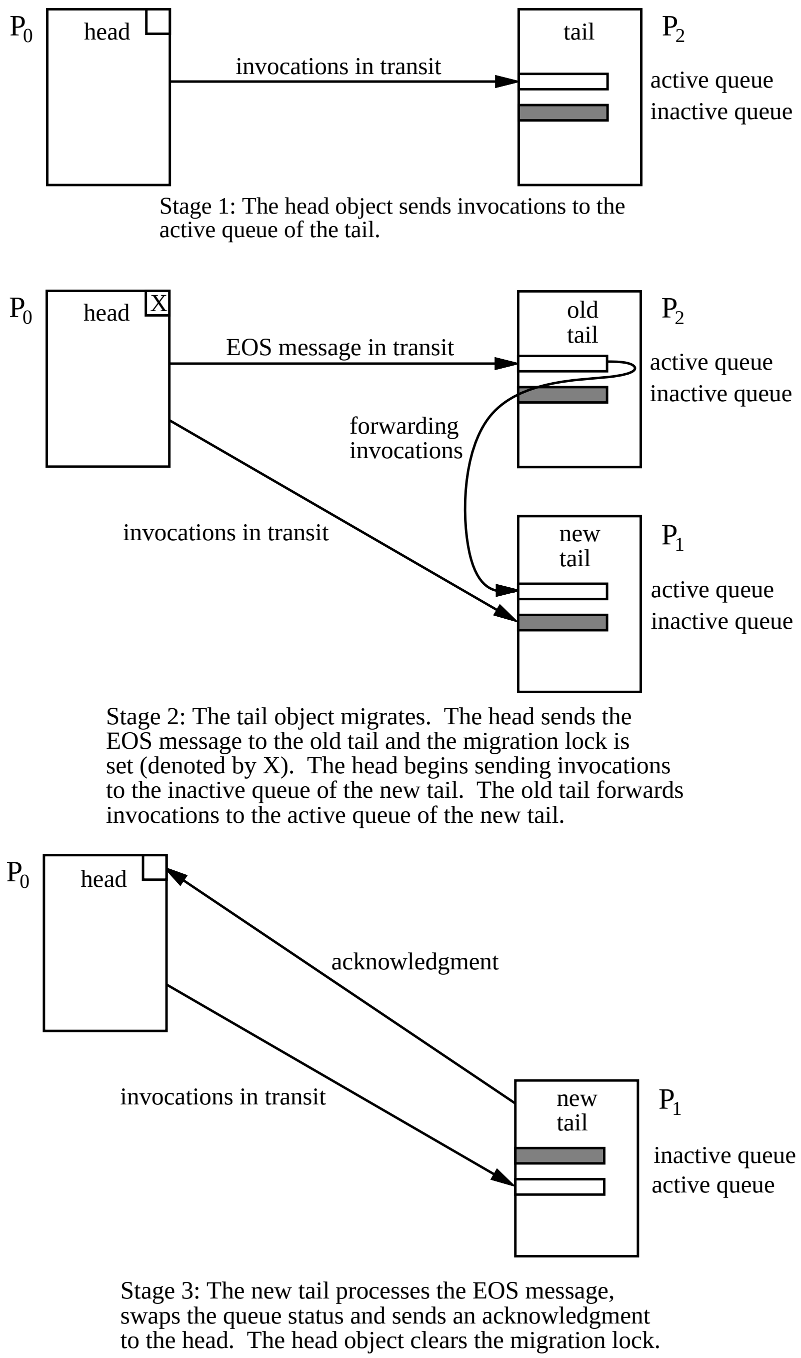

Figure 4: Tail migration using multiple queues

head be stored on processor P_0 and the tail on processor P_2. In Stage 1, no migration has taken place and the head sends invocations to the active queue of the tail. A request to migrate a pipe tail is sent first to the head, where the migration lock is set. In Stage 2 tail migrates to P_1 and a message is sent to the head object notifying it of the new location of the tail. The head sends an EOS message to the old tail and then begins sending invocations to the inactive queue of the new tail. The old tail forwards all queued invocations to the active queue of the new tail, including the EOS message. When the new tail receives the EOS message it swaps the queues and sends an acknowledgment to the head to clear the migration lock, as shown in Stage 3.

6 Performance Comparisons

In this section we compare the performance of the four implementations of pipes. We contrast the time taken to process an invocation at the head, the time spent queuing the invocation at the tail, and the storage overhead. Finally, we compare the overheads of pipes with the cost of ordering all asynchronous invocations in the system.

6.1 Performance at the caller

We measure the "normal case" pipe operation, where the calling thread is located on the same processor as the head, and the tail and target object are remote. We contrast the time taken to call `pipe_send` for each implementation. This time is the latency for pipe calls seen by the calling thread. Furthermore, we found that the performance bottleneck for pipe calls is due to this latency. Thus, the latency for pipe calls at the caller determines the throughput of the pipe.

In order to appreciate the relative overhead of pipe implementations compared to those for issuing an unordered asynchronous invocation, we express the time for the pipe call in terms of the number of additional RISC machine cycles above the cost of an unordered asynchronous invocation.

The code executed during `pipe_send` is almost identical in all the pipe implementations. Furthermore, it is very similar to the code that is executed to perform an unordered asynchronous call. This makes it possible to identify the operations that are not present in all implementations and express the time to call `pipe_send` in terms of the time to execute these operations. There are only seven of these operations:

Access tail reference: read the tail object reference in the head object data structure.

Check locality of the tail: determine whether the tail is co-located with the head.

Increment send count: increment the count of the number of sends in progress (used in migration control).

Operation, number of RISC cycles, read (R) and write (W) operations		Sequence Number	Synchronous Send	Simplified Sequence #	Multiple Queues
access tail reference	(1 cycle, 1R)	×	×	×	×
check tail locality	(3 cycles, 1R)	×	×	×	×
increment send count	(3 cycles, 1R, 1W)	×	×	×	×
decrement send count	(3 cycles, 1R, 1W)	×	×	×	×
increment sequence #	(3 cycles, 1R, 1W)	×		×	
marshal sequence #	(4 cycles, 1R, 2W)	×		×	
reply interrupt	(32 cycles)		×		
Total Additional Cycles		17	42	17	10

Table 1: Cost breakdowns for the time for pipe calls at the head expressed in terms of the number of RISC cycles required over and above the time for an unordered asynchronous invocation. The number of memory reads and writes for each case is also given.

Decrement send count: decrement the count of the number of sends in progress (used in migration control).

Increment sequence number: read and increment the sequence number associated with the head object.

Marshal sequence number: marshal the sequence number into the message buffer that is sent to the tail object.

Synchronous reply interrupt: execution of the interrupt handler used for the acknowledgment in the pipe implementation that uses synchronous messages.

A counter of the number of pipe calls in progress at the head (the *send count*) is required to synchronize between pipe invocations and head migration. When a head migration request occurs, it waits until the send count is zero before migrating the head. This ensures that pipe invocations and head migrations occur atomically relative to each other.

Table 1 gives a breakdown of the time for the call to `pipe_send` for each implementation in terms of the seven operations listed above. We measure the time in terms of RISC cycles; we assume that a memory read and write each take only one cycle. In some systems this may not be the case, so Table 1 also gives a breakdown of the number of read and write operations for each case.

The results in Table 1 show that, for networks that do not preserve message order, the sequence number implementation has a lower time and hence a higher throughput than using the synchronous message implementation. The comparatively poor performance of the synchronous message implementation is caused by the time taken by the processor storing the head object to execute the interrupt handler. If the network preserves order, then the multiple queue implementation has the best throughput of all; this is because the operations associated with the sequence numbers have been removed.

6.2 Performance at the target

We also measured the number of additional RISC cycles required at the tail over and above the time for an unordered asynchronous invocation. We measured for "normal case" operation, where invocations are received in order (so no buffering is required) and the tail and target object are co-located. We are again able to break down the time in terms of a number of operations:

Unmarshal sequence number: unmarshal the sequence number from the message buffer received at the tail object.

Check sequence number: compare the sequence number value in the message to that for the next in-order message.

Send acknowledgment: send the acknowledgment back to the head object in the pipe implementation using synchronous messages.

Table 2 gives a breakdown of the additional time required by each implementation at the tail. These results also show that, for networks that do not preserve message order, the sequence number implementation has lower overhead than the synchronous send implementation. For networks that preserve message order, the multiple queue implementation incurs no additional overhead over unordered asynchronous invocations.

6.3 Storage overhead

Finally, we compared the storage overheads of each implementation in terms of the space that each requires over and above the structures common to all. All the implementations of head objects require 1 word to store a reference to the tail object, 1 byte to store the send count and 1 byte to store status flags and migration information. All the implementations of tail objects require 2 words to store references to the head object and target object, 2 words for pointers to the front and back of the ordered invocation

Operation, number of RISC cycles, read (R) and write (W) operations	Sequence Number	Synchronous Send	Simplified Sequence #	Multiple Queues
unmarshal sequence # (4 cycles, 2R, 1W)	×		×	
check sequence number # (3 cycles, 1R)	×		×	
send acknowledgment (10 cycles)		×		
Total Additional Cycles	7	10	7	0

Table 2: Cost breakdowns for the time for pipe calls at the tail expressed in terms of the number of RISC cycles required over and above the time for an unordered asynchronous invocation. The number of memory reads and writes for each case is also given.

Pipe Object	Sequence Number	Synchronous Send	Simplified Sequence #	Multiple Queues
head	4 bytes	1 bit	4 bytes	1 word
tail	4 bytes + 9 words	–	4 bytes + 2 words	2 words

Table 3: Additional storage overheads for the pipe implementations.

queue and 1 byte for status flags. Table 3 gives the additional space overheads for each implementation for head and tail objects.

We use a table to store out-of-order messages in the sequence number implementation for networks that do not preserve message order. We have chosen a table size of eight entries (i.e., if the tail expects message $[i]$, then messages $[i+1]$ through $[i+8]$ can be stored in the table). Any messages that are not stored in this table are held in an overflow buffer, which is implemented as a linked list. This results in 9 additional words of storage for the table plus an additional 4 bytes at the head and tail for the current sequence number value. In the synchronous send implementation, only an additional bit (the *send in progress* flag) is required at the head. For simplified sequence numbers, the table used for storing out-of-order messages is replaced with pointers to the front and back of the out-of-order queue. Finally, for the multiple queue implementation, an additional word is required at the head for the reference to the additional queue, and two pointers are required at the tail for the front and back of this queue.

6.4 Cost of ordering all asynchronous invocations

Rather than providing a linguistic mechanism for ordering asynchronous invocations, we could order all asynchronous invocations in the system. However, this raises semantic issues concerning the meaning of "ordered invocations". For example, the sender of an invocation may be viewed as a thread, an object or a processor. Similarly, the receiver may be viewed as an object or a processor. Let us assume for example, that we choose to order invocations between a sending thread and receiving object.

On networks that do not preserve message order we could implement ordered asynchronous messages by using sequence numbers for all asynchronous invocations between the processors in the system. In the normal case, this would increase the time to send an asynchronous invocation by 7 cycles (increment a sequence number and marshal it), and would increase the time to receive an asynchronous message by 7 cycles (unmarshal the sequence number and check the value). For networks that preserve message order, we would only need to be concerned with preserving execution order across migration, which can be achieved by using an acknowledgment for the last message sent on the old path. However, this is a naive approach for multiprocessor systems for the following four reasons:

1. A message send can cost as little as 30 cycles (the time for a message send on the J-machine [16]). In this case the additional 7 cycles at the sender would be significant.

2. In a multiprocessor system with N processors the amount of storage space required to hold the sequence numbers and table for reordering would be $O(N)$ on every processor.

3. Serializing all invocations reduces concurrency.

4. Migrations increases the complexity of the implementation.

The storage overhead associated with ordering all asynchronous invocations for a fine-grain multiprocessor is best illustrated by an example. Consider a 4K-node machine that uses adaptive routing, thus allowing out-of-order delivery. We will assume that the system correctly handles object migration and implements message ordering by sequence numbers between each processor pair. Each processor requires 10 words of storage for each of the other processors in the system. The total storage overhead would then be $(4096 - 1) \times 10 \approx 40$K words per processor. If each processor has 2M words of local memory, then the storage overhead would represent 2% of the memory in the system. However, if a more intelligent scheme is adopted then most of this overhead can be avoided. For example, the synchronized-clock message protocol (SCMP) [36] for distributed systems only keeps sequence number information for pairs that have communicated "recently".

Serializing all invocations between a thread and an object restricts concurrency between threads executing on the same object, and can lead to severe performance reductions. In addition, in those cases where order is not important, the execution order that yields the highest throughput should be chosen; this would not be possible if order is imposed in every case. Furthermore, we only wish to serialize invocations between thread/object pairs and not between processor pairs. The target object is therefore required to do additional work to determine which invocations can execute concurrently. A linguistic construct that explicitly specifies where order is required removes these restrictions.

If threads and objects migrate then sequence numbers between processors is not sufficient. Additional code is required to forward messages following a migration and to handle reordering. This reduces performance.

6.5 Discussion

In this section we have shown that the performance overheads associated with pipes are low. For pipe calls in systems in which the network does not preserve message order, an additional 17 RISC cycles are required at the head and an additional 7 cycles are required at the tail. In systems in which message order is preserved, pipe calls require only an additional 10 cycles at the head. Also, the space overhead in both cases is small. Finally, the alternative of preserving order for all asynchronous invocations in a system is too restrictive in terms of space overhead and performance.

7 Flow Control

Flow control in a multiprocessor is used to restrict message traffic so as to prevent buffer overflow. A flow control policy provides a mechanism for throttling traffic on the network and for determining which circuits must be throttled. In this section we describe how the pipe construct can be used to provide flow control for asynchronous invocations.

Flow control can be supported by either the sender or receiver. A send/acknowledge protocol for each message is a form of receiver-controlled flow control. A processor sends a message and waits for an acknowledgment before sending the next message on that link. The acknowledgments can be "piggy-backed" onto messages going in the opposite direction, which leads to looser coupling between the sender and receiver. Alternatively, flow control can be sender-controlled. Messages are only sent when the sender knows that there is sufficient storage for the message at the receiver.

In the absence of system support for flow control, it may be necessary for the applications program to limit the number of outstanding asynchronous invocations. For example, in the classic "producer-consumer" problem, it is possible for the producer to get too far ahead of the consumer if asynchronous invocations are used. The stored messages may require a substantial amount of memory; this may slow the system down due to paging or swapping. In the extreme, messages may be lost due to insufficient storage and run-time errors can occur. One solution for this example is to have the consumer occasionally send an acknowledgment to the producer [18]. However, in more general cases throttling may be impossible as it may not be possible to determine who the next sender will be. Implementing flow control in application programs also increases the complexity of the code. Gehani [18] implemented four versions of the producer-consumer example and noted that: "This version [asynchronous with flow control] of the producer-consumer example is considerably harder to understand (and to debug) than the other versions."

Pipes can provide system-level flow control for single and multiple senders by using a clear-to-send mechanism. When a pipe tail finds that its pending invocation queue has become too long, it can throttle all the senders by throttling the head of the pipe. The tail sends a message to the head that causes the head to stop sending invocations to the tail. The head could then simply queue invocations locally until it is free to begin forwarding them to the tail. However, this approach does not throttle the calling threads, which are still free to add invocations to the queue in the head. The problem has simply been moved from the tail to the head. To avoid this problem, the head suspends all local threads that attempt invocations on the pipe. Invocations are made synchronously on the head and remote invocations generate local threads. Therefore, suspending all local threads throttles all the senders as they attempt to make invocations on the pipe. Once the tail has processed enough of the pending invocations, it sends a message to the head that causes all the suspended threads to be restarted; normal pipe operation resumes.

In this case, the applications programmer need not be concerned with flow control in asynchronous invocations that use pipes. This results in code that is simpler to understand and debug. Pipes also solve the problems associated with throttling multiple senders since all invocations must be made synchronously on the pipe head. However, pipes also provide sequencing semantics that are not always required. To avoid paying for these semantics in applications that require only the flow control properties of the pipe, we could remove the sequencing semantics. We term this construct "an unordered pipe"; invocations on an unordered pipe execute in parallel with no guarantees made about the order of execution.

8 Related Work

The issue of synchronous versus asynchronous message passing has been discussed in the context of programming languages [2, 18, 33] and distributed operating systems [19, 46]. Languages have been designed that contain only synchronous message passing constructs [7, 11, 23, 24, 25, 28, 48], or only asynchronous message passing constructs [17] or a combination of both [2, 6, 13, 18, 41].

Liskov, Herlihy and Gilbert [33] show that the combination of synchronous communication (such as rendezvous or remote procedure calls) with a static process structure (such as Ada tasks) leads to complex and indirect solutions to common problems in distributed and concurrent systems. They conclude that a language designed for programming these systems should provide either asynchronous communication or a dynamic process structure (but it need not provide both). With a dynamic process structure, fork and join constructs can be used to program in an asynchronous style with remote procedure calls [8]. However this approach has a high overhead for each asynchronous call [20].

The language SR [2] originally provided synchronous and asynchronous communication with only static process structure. Conversely, Concurrent C [18] originally had only synchronous message passing with a dynamic process structure. With experience, the designers of both SR and Concurrent C felt that these mechanisms were not sufficiently expressive. SR was extended to include a dynamic process structure and asynchronous communication was added to Concurrent C.

Interprocess communication in languages such as CSP [24], occam [28], SR [2], Concurrent C [18] and Hermes [6] is performed via explicit send and receive primitives. CSP and occam support only synchronous messages. Concurrent C, SR and Hermes support both synchronous and asynchronous messages. However, asynchronous messages can not have return values.

In the standard Actor model (the model used in the language Acore [38]) there are no guarantees that messages are sent in order. However, the order of reception and execution are equivalent. In ABCL/1 [54] messages are transmitted, received and executed in order.

Programming languages such as Sloop [37], Emerald [9, 25, 27] and Amber [11] abstract away from explicit send and receive constructs by providing a remote procedure call interface. Emerald and Amber support only synchronous invocations although their dynamic process structure allows an asynchronous style to be used. Sloop attempts to combine the advantages of the procedural interface with the parallel execution capabilities of asynchronous invocation; when one object invokes an operation on another object, it only waits for the operation to complete if the operation returns a value. Operations that do not return a value execute asynchronously.

Previous work by Gifford and Glasser [20] and in the Mercury system [29] resulted in the design of remote invocation mechanisms for distributed systems in which a sequence of calls between a single client and a single server are run in order, but asynchronously with respect to the caller. This permits the sequencing semantics required by certain calls to be preserved, while allowing other calls to run in parallel. Gifford presents a communication model for distributed systems that combines the advantages of bulk data transport and remote procedure calls in a single framework. However, the definition of a server specifies whether its operations must be invoked synchronously or asynchronously by the clients.

In Mercury [29], call-streams allow a sender to make a sequence of calls to a receiver without waiting for replies. The stream guarantees that the calls will be executed at the receiver in the order they were made and that the replies from the receiver will be delivered to the sender in call order. To make Mercury usable within a particular programming language, some extensions to the language (termed veneers) are needed. Veneers are provided for Argus [31], C, and Lisp. However, a single Mercury stream cannot be shared between a number of calling threads. Stream calls made by different threads are sent on different streams.

In some distributed operating systems [4, 26, 42], processes transfer messages among themselves in an asynchronous, network-transparent, and process-name-transparent manner. In these systems the message order between a sending process and a receiving process is preserved. The language Matchmaker [26] is used in the Mach environment [26] to hide the underlying message-passing mechanisms via a procedural interfaces for sets of operations upon objects; a similar interface is provided by the language Lynx [43] for the Charlotte [3, 4] operating system. In Mach, a client process can initiate an asynchronous request on a server. A server can choose any execution style appropriate to the semantics of the operations being performed, from serial execution to unconstrained parallel execution. Each reply is typically returned to a unique port (a send-once right) that is generated at request time and is destroyed when the reply is received. This must be coded explicitly within the clients and servers.

9 Conclusion

We have described a new linguistic mechanism, the pipe, for sequencing asynchronous procedure calls in multiprocessor systems. Our design provides integrated language support for ordered asynchronous invocations, and also allows invocations from multiple sending threads to be ordered.

In many applications, such as transaction systems and dictionaries, a sequence of invocations must be performed in order but can run in parallel with the calling thread. Implementing ordered invocations on top of unordered invocations adds complexity and is difficult to debug.

Ordering semantics should be provided by the language rather than coded at the application level because application programs will be simpler and more likely to be correct, and the compiler can improve performance by taking advantage of message ordering provided by the underlying architecture without affecting the portability of applications.

We have presented four implementations of pipes. Of the implementations we considered, the general sequence number implementation is the best for systems in which the network does not preserve message order. For systems that preserve message order, our multiple queue implementation is the most favorable. In general, the performance and space overheads associated with pipes are low.

Pipes can also provide application-transparent flow control for asynchronous invocations; they can throttle invocations from multiple calling threads. To avoid paying for the sequencing semantics of pipes in applications that require only the flow control properties, unordered pipes provide an unordered channel connecting multiple senders to a receiver.

Given that the ordering semantics should be provided by the programming language, we can choose to provide an explicit construct such as the pipe, or guarantee order for all asynchronous invocations in the system. We have shown that preserving the order for all asynchronous invocations in a system is too restrictive in terms of performance and space overhead. We conclude that procedural languages designed for programming multiprocessor systems should provide a pipe mechanism for ordered asynchronous invocations, which allows the programmer to choose to incur the additional cost of ordering only when the semantics of the application require it.

10 Acknowledgments

Several others contributed to our work on pipes, including Crysanthos Dellarocas, Anthony Joseph, Carl Waldspurger, Barbara Liskov, Sanjay Ghemawat and Robert Gruber.

References

[1] A. Agarwal et al. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. In *Scalable Shared-Memory Multiprocessors*. Kluwer Academic Publishers, 1992.

[2] G.R. Andrews and R.A. Olsson. The evolution of the SR language. *Distributed Computing*, 11:133–149, 1986.

[3] Y. Artsy, H. Chang, and R. Finkel. Interprocess communication in Charlotte. *IEEE Software*, 4(1):22–28, 1987.

[4] Y. Artsy and R. Finkel. Designing a process migration facility: the Charlotte experience. *IEEE Computer*, 22(9):47–56, 1989.

[5] W. Athas and C.L. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, pages 9–24, August 1988.

[6] D.F. Bacon and A. Lowry. A portable run-time system for the Hermes distributed programming language. Technical Report 15686, IBM Research Division, T.J. Watson Research Center, February 1990.

[7] J.K. Bennett. The design and implementation of Distributed Smalltalk. In *Proceedings of the Object-Oriented Programming Systems Languages and Applications Conference*, pages 318–330, 1987.

[8] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[9] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. Technical Report 86-02-04, Department of Computer Science, University of Washington, February 1986.

[10] E.A. Brewer, C.N. Dellarocas, A. Colbrook, and W.E. Weihl. PROTEUS: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, MIT Laboratory for Computer Science, 1991.

[11] J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, and R.J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. Technical Report 89-04-01, Department of Computer Science, University of Washington, April 1989.

[12] D.D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–29, 1989.

[13] R.P. Cook. *Mod: A language for distributed programming. *IEEE Transactions on Software Engineering*, SE-6(6):563–571, 1980.

[14] W.J. Dally and A. Hiromichi. Adaptive routing in multicomputer networks using virtual channels. *to appear in IEEE Transactions on Parallel and Distributed Systems*, 1992.

[15] W.J. Dally and C.L. Seitz. The Torus routing chip. *Distributing Computing*, 1:187–196, 1986.

[16] W.J. Dally et al. The J-machine: A fine-grain concurrent computer. In G.X. Ritter, editor, *Proceedings of the IFIP Congress*, pages 1147–1153. North-Holland, August 1989.

[17] J.A Feldman. High level programming for distributed computing. *Communications of the ACM*, 22(6):353–368, 1979.

[18] N.H. Gehani. Message passing in Concurrent C: synchronous versus asynchronous. *Software Practice and Experience*, 20(6):571–592, 1990.

[19] W.M. Gentleman. Message passing between sequential processes: the reply primitive and the administrator concept. *Software Practice and Experience*, 11:435–466, 1981.

[20] D.K. Gifford and N. Glasser. Remote pipes and procedures for efficient distributed communication. *ACM Transactions on Computer Systems*, 6(3):258–283, August 1988.

[21] D.K. Gifford et al. Information storage in centralized computer systems. Technical Report CSL-81-8, Xerox Palo Alto Research Center, 1982.

[22] R. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[23] P. Brinch Hansen. Joyce - a programming language for distributed systems. *Software Practice and Experience*, 17:29–50, 1987.

[24] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, Englewood Cliffs, NJ, 1985.

[25] N.C. Hutchinson. Emerald: An object-based language for distributed programming. Technical Report 87-01-01, Department of Computer Science, University of Washington, January 1987.

[26] M.B. Jones and R.F. Rashid. Mach and Matchmaker: Kernel and language support for object-oriented distributed systems. In *Proceedings of the Object-Oriented Programming Systems Languages and Applications Conference*, pages 67–77, 1986.

[27] E. Jul, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, 1988.

[28] INMOS Limited. *Occam Programming Manual.* Prentice Hall, Englewood Cliffs, New Jersey, 1984.

[29] B. Liskov, T. Bloom, D. Gifford, R. Scheifler, and W. Weihl. Communication in the mercury system. Technical Report PMG Memo 59-1, MIT Laboratory for Computer Science, 1987.

[30] B. Liskov, T. Bloom, D. Gifford, R. Scheifler, and W. Weihl. Communication in the Mercury system. In *Proceedings of the 21st Annual Hawaii Conference on System Sciences*, January 1988. Available as MIT LCS Programming Methodology Group Memo 59.

[31] B. Liskov, M. Day, M. Herlihy, P. Johnson, G. Leavens (editor), R. Scheifler, and W. Weihl. Argus reference manual. Technical Report MIT/LCS/TR-400, MIT Laboratory for Computer Science, November 1987.

[32] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* MIT Press, 1986.

[33] B. Liskov, M. Herlihy, and L. Gilbert. Limitations of synchronous communication with static process structure in languages for distributed computing. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 150–159, 1986.

[34] B. Liskov and R. Scheiffer. Guardians and actions: Linguistic support for robust, distributed programs. In *Proceedings of the Ninth ACM Symposium on the Principles of Programming Languages*, 1982.

[35] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 260–267, 1988.

[36] B. Liskov, L. Shrira, and J. Wroclawski. Efficient at-most-once messages based on synchronized clocks. *ACM Transactions on Computer Systems*, 9(2), May 1991.

[37] S.E. Lucco. Parallel programming in a virtual object space. In *Proceedings of the Object-Oriented Programming Systems Languages and Applications Conference*, pages 26–33, 1987.

[38] C.R. Manning. Acore: The design of a core actor language and its compiler. Master's thesis, MIT Laboratory for Computer Science, 1987.

[39] J. Ngai. A framework for adaptive routing in multicomputer networks. Technical Report Caltech-CS-TR-89-09, Computer Science Department, California Institute of Technology, 1989.

[40] R.S. Nikhil, G.M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. Computational Structures Group Memo 325-1, MIT Laboratory for Computer Science, 1991.

[41] F.N. Parr and R. Strom. NIL: a high-level language for distributed systems programming. *IBM Systems Journal*, 22(1/2):111–128, 1983.

[42] M.L. Powell and B.P. Miller. Process migration in DEMOS/MP. In *Proceedings of the Ninth ACM Symposium on Operating System Principles*, pages 110–119, 1983.

[43] M.L. Scott. Language support for loosely coupled distributed programs. *IEEE Transactions on Software Engineering*, SE-13(1):88–103, 1987.

[44] C.L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, 1985.

[45] H. Sullivan and T. Bashkow. A large-scale homogenous machine. In *Proceedings of the 4th International Symposium on Computer Architecture*, pages 105–124, 1977.

[46] A.S. Tanenbaum and R.V. Renesse. Distributed operating systems. *ACM Computer Surveys*, 17(4):419–470, 1985.

[47] Thinking Machines Corporation, 245 First Street, Cambridge, MA 02154-1264. *The Connection Machine CM-5 Technical Summary*, October 1991.

[48] United States Department of Defense, ANSI/MIL-STD-1815A. *Reference manual for the Ada programming language*, January 1983.

[49] P. Wang. An in-depth analysis of concurrent B-tree algorithms. Technical Report MIT/LCS/TR-496, MIT Laboratory for Computer Science, January 1991.

[50] W. Weihl, E. Brewer, A. Colbrook, C. Dellarocas, W. Hsieh, A. Joseph, C. Waldspurger, and P. Wang. PRELUDE: A system for portable parallel software. Technical Report MIT/LCS/TR-519, MIT Laboratory for Computer Science, 1991.

[51] W.E. Weihl and P. Wang. Multi-version memory: Software cache management for concurrent B-trees. In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pages 650–655, 1990.

[52] J. White. A high-level framework for network-based resource sharing. In *Proceedings of the National Computer Conference*, pages 561–570, 1976.

[53] J. Yantchev and C.R. Jesshope. Adaptive, low latency, deadlock-free packet routing for networks of processors. *IEE Proceedings Part E*, 136(3):178–186, 1989.

[54] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Modelling and programming in an object-oriented concurrent language ABCL/1. In Akinori Yonezawa and Mario Tokoro, eds., *Object-Oriented Concurrent Programming*, 1987.