International Journal of Parallel Programming, Vol. 19, No. 1, 1990

Cache Coherence Requirements for Interprocess Rendezvous

Russell M. Clapp,[1] Trevor N. Mudge, and Donald C. Winsor

Received May 1989; Revised July 1990

Multiprocessors in which a shared bus is used by the processors to communicate with common memory are an emerging class of machines where there is a need to support parallel programming languages. A language construct that is found in a number of parallel programming languages to support synchronization and communication in the interprocess rendezvous. Shared-bus multiprocessors require a protocol to keep the date in their caches coherent. There are two major categories of these protocols: invalidation and write-broadcast. This paper examines the requirements for cache coherence protocols to support efficient interprocessor rendezvous. The approach taken is to examine the memory referencing patterns to the run-time data structures during rendezvous execution. The appropriate coherence protocol is shown to be a function of the processor scheduling strategy used by the run-time system at synchronization points during the rendezvous. When processes migrate freely as a result of the scheduling strategy, invalidation protocols are found to be more efficient. When migration is restricted by the scheduler, write-broadcast protocols are more efficient.

KEY WORDS: Cache coherence; rendezvous; run-time systems; process migration; concurrent programming languages.

1. INTRODUCTION

In the past few years a wide variety of commercial multiprocessors have emerged. The most successful by far have been the shared-memory style architectures that connect no more than a few dozen processors and their

[1] Advanced Computer Architecture Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, Michigan 48109–2122.

0885-7458/90/0200-0031$06.00/0

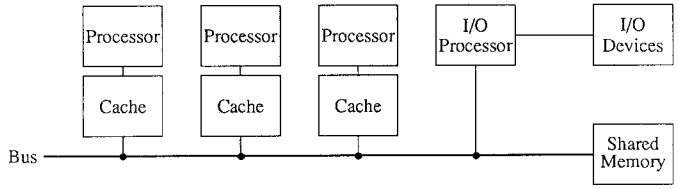

Fig. 1. Cached shared-memory multiprocessor.

caches to a common memory using a shared bus.[1,2] Figure 1 illustrates this shared-bus architecture. Examples of such systems are the Sequent and Encore series of computers.[3,4] In these architectures copies of the data can be present in several caches. Therefore, a mechnism is present to keep the data consistent or, as usually referred to, *coherent*. The most common approaches to maintaining coherence rely on a bus watching mechanism in each cache that monitors bus activity to keep its data consistent. These bus watching mechanisms are frequently referred to as snooping cache protocols.[5,6]

The commercial success of the shared-bus multiprocessors is due, in large part, to the fact that they provide a direct and cost-effective replacement for supermini class systems. In this role, they usually provide a timeshared computing facility that serves a job stream of logically independent heavyweight processes, typically separate user programs. These processes are assembled into a single job queue and distributed among the available processors. This application of shared-bus multiprocessors has been referred to as *multistream* operation.[7] More recently, shared-bus multiprocessor architectures are also being used to speedup the execution of single programs through parallel processing.[8–13] The advent of single user parallel computers such as multiprocessor workstations and graphics supercomputers, including Digital's experimental Firefly,[14] Silicon Graphics' IRIS 4D series,[15] Apollo's DN10000,[16] and Ardent's Titan (now the Stardent 3000),[17] are furthering this trend.

To take advantage of the shared-bus multiprocessor's potential for parallel processing requires programming languages that support parallelism. By incorporating the parallelism into the language, the runtime support can be made more efficient by taking advantage of lightweight processes that share address space and permit fast context switching between program tasks without operating system intervention. Key components in any parallel procedural language are mechanisms for synchronization and communication. The interprocess rendezvous is a language construct that supports both of these mechanisms. This paper examines the efficient implementation of interprocess rendezvous on

shared-bus multiprocessors, focusing on the cache coherence requirements. An efficient implementation of the rendezvous is of general interest because, in various forms, it is the basic synchronous communication primitive for several concurrent languages, including Ada,[18] Concurrent C,[19] and Occam.[20] The rendezvous is also one of several forms of communication in the distributed programming languages Synchronizing Resources,[21] LYNX,[22] and NIL.[23] Each of these language has its own particular variations of the rendezvous.

There have been several pioneering studies of the impact of cache coherence on parallel programs.[6,24–26] In these studies, multiprocessor traces were used as input to a simulator. These traces came from synthetic multiprogramming workloads and parallel applications. The applications used were written in sequential languages, and parallelized at the language level using *ad hoc* system dependent library routines and Single Program Multiple Data[1] (SPMD) techniques similar to those described in Refs. 9, 27–29. SPMD parallel programs consist of one sequential program that is replicated on several processors, each operating on its own set of data. The resulting traces are not representative of the behavior of parallel language programs that communicate primarily by rendezvous. Multiprogramming traces do not exhibit any application level sharing, and SPMD style programs are normally found on multicomputer systems such as hypercubes. Because SPMD style programs are more suited to machines that do not share memory, this style of programming is dramatically different from what one would expect in the tightly coupled shared-memory class of machines considered in this discussion. In contrast, when parallel algorithms are coded in a concurrent language using processes and rendezvous, the parallel programs are more tightly coupled and data sharing between processes is more common. It is this latter style of parallel programming that leads to higher levels of data sharing than that currently observed in the aforementioned parallel traces.

An alternative approach to coherence protocol evaluation would simulate performance based on memory reference patterns that are derived from statement frequency statistics gathered from a large base of parallel programs. While this approach was used to construct a representative mix of statements for the Dhrystone benchmark program,[30] this gathering of statistics has not been done for programs written in concurrent languages. Furthermore, any such study would be highly dependent on the applications chosen, since the state of parallel programming in such languages is still immature, and there are too few large programs to use as a basis. With this immaturity has also come some controversy and confusion over "good programming practice" for concurrent languages. In the case of Ada, the Language Reference Manual (LRM)[18] states explicitly that rendezvous

are the preferred mechanism for sharing data between program tasks. In fact, if variables are shared in Ada without using rendezvous to at least provide synchronization, the value of the variables is not guaranteed to be consistent, and the program is said to be *erroneous*. Clearly, the rendezvous construct has been included in many concurrent languages as the primary means of data sharing between threads of execution. And finally, even though it is unclear as to the percentage of memory references that are due to rendezvous activity, it is becoming apparent that even infrequent synchronization activity and references to shared data can dominate a program's running time if they are not implemented efficiently.[31]

Our approach to establishing the cache coherence requirements for interprocess rendezvous is quite different from previous approaches. We examine what happens to the run-time support data structures during interprocess rendezvous and, in particular, how cache coherence protocols affect them. The approach is based on our experience in designing run-time support mechanisms for interprocess rendezvous.[32–35] We propose a typical implementation and then examine the reference patterns that occur to the run-time system data structures during rendezvous execution. From this referencing behavior the effect of different cache coherence protocols is assessed. The advantage of this approach is that it overcomes the current lack of a large base of parallel programs with which to perform experiments, particularly ones coded using one of the several concurrent languages listed earlier. On the other hand, the results of our approach of examining data structure reference patterns are subject to the implementation choices for the run-time system. While we describe the variations in rendezvous semantics for several languages, the following sections will primarily concentrate on the rendezvous semantics for Ada.

The remainder of this paper is organized as follows. The next section provides background on the rendezvous construct and describes a typical implementation of its run-time support. In Section 3, a brief overview of cache coherence protocols is given. In Section 4, we examine the interaction of the rendezvous run-time support with processor scheduling strategies and assess the impact on cache coherence requirements. Finally, some including remarks are made.

2. INTERPROCESS RENDEZVOUS

2.1. Background

The term *rendezvous* was first used by the designers of Ada[18] to describe the construct for communication between processes, which are

referred to as *tasks* in Ada (*process* and *task* will be used interchangeably). This high-level primitive can be used to send a signal, lock a shared variable, or send a message, while avoiding the problems associated with low-level primitives such as semaphores.[36]

The rendezvous construct can be traced to the process communication primitives described in Hoare's Communicating Sequential Processes (CSP) and Brinch-Hansen's Distributed Processes (DP).[36] The two processes involved in a rendezvous communicate through a procedure call style interface and suspend execution while data is transferred. If one process requests a rendezvous while the other is busy, the requester will block and wait for the other process to become available. In some versions it is possible to withdraw requests for rendezvous or suspend while requesting one of several possible rendezvous. The *extended rendezvous* allows computation (a critical section) in one process and bidirectional data flow. The main difference between the rendezvous and a procedure call is that the called process in a rendezvous is an active object that may be capable of handling several different call interfaces, while the procedure is a passive object with a single interface that is only activated when called. Moreover, procedures are normally reentrant and multiple calls may execute in parallel.

2.2. RENDEZVOUS SEMANTICS

Basic Model. The basic rendezvous is summarized in Figs. 2 and 3. Both figures show execution time with solid lines and blocked time with dashed lines. The ovals show points where the run-time system is invoked,

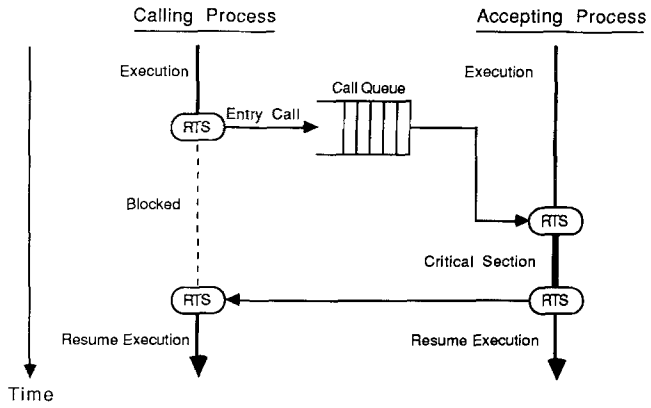

Fig. 2. Rendezvous execution: caller blocks first.

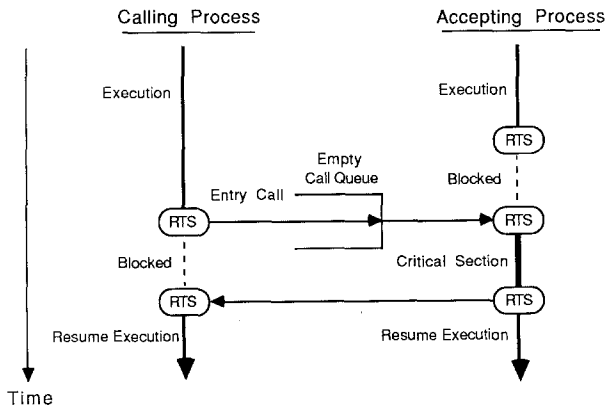

Fig 3. Rendezvous execution: acceptor blocks first.

and context switches occur. We will adopt the terminology of Ada and refer to these control transfer as *synchronization points.*[18]

Figure 2 shows the case where the calling process issues an *entry call* before the accepting process is ready for communication. The calling process is blocked by the rendezvous request. The run-time system enqueues the *call record* (see Fig. 4 for queue structure) and may schedule another process to execute on that processor. When the accepting process is ready to receive the call, it enters the run-time system so that the call may be dequeued and the accepting process be prepared to service it. At this point, the run-time system may schedule another process to run on this processor, but it may also reschedule the accepting process since it is now ready to proceed with the rendezvous. If there is a critical section to be performed (an extended rendezvous), the accepting process executes it and enters the run-time system when it is completed. Next, any return parameters are made available to the calling process and both processes are made runnable and are eligible to be scheduled. If there is no critical section, the two calls to the run-time system made by the accepting process may be combined into one. Variations may limit the queue to a length of one (e.g., Occam) or may keep the queue in priority order (e.g., Concurrent C). (A detailed comparison of rendezvous semantics between Ada and Concurrent C can be found in Ref. 37.)

Figure 3 shows the case where the acceptor is the first process to reach the synchronization point. Since no calls are pending, the acceptor blocks until a call is made. When a call is available to be serviced, a call record is made available immediately to the accepting process and the rendezvous begins. The rendezvous completes in the same manner described for Fig. 2.

Variations. Several general variations of the simple rendezvous are possible. These include placing conditional requirements and time bounds on both the calling and receiving sides of the call. A conditional requirement would prevent a call from being attempted or accepted if it cannot proceed immediately. In the case of time bounds, a call record is removed from the queue if a timeout expires before the call can be accepted, or an accepting process stops waiting for a call to arrive after some delay and resumes execution of some other useful work. An *alternate* construct allows the accepting process to choose from one of several different entry points. This construct can be combined with *guards*, which are boolean conditions on entry points that must be satisfied in order for a particular call to be accepted.

2.3. RENDEZVOUS IMPLEMENTATION

Run-Time Environment. Here, consider a basic run-time environment that is similar to the one described in Ref. 32. Although this reference describes an implementation intended for a distributed memory multiprocessor, the characteristics of a multithreaded run-time system and its data structures for the rendezvous are nearly identical for any process model parallel language on a shared-bus multiprocessor.

It is assumed that there is a gobal time-of-day clock available for reading by all processors, and that an interval timer is available for time slicing.[38] It is also assumed that the hardware provides some support for mutual exclusion such as the locks described in Ref. 39 or some type of noninterruptible memory read-modify-write instruction. The run-time system is a reentrant kernel resident in memory. Each processor executes in the run-time system in response to calls to perform rendezvous. The run-time system maintains several types of data structures to help perform its operations. These include task control blocks (TCBs) for each process, a run queue of TCBs corresponding to processes that are ready to be scheduled, entry queues of call records for each entry, pointers to the head and tail of each entry queue, and a linked list of timed events. The TCBs contain state information and local stack space. The run queue indicates the order in which ready but non-executing processes are to be scheduled when processors become available. This queue will be kept in priority order if priorities are supported. Entry queues are managed as linked lists of call records, with pointers to their heads and tails stored in the TCB containing the entry point (see Fig. 4). The list of timed events is a doubly linked list of records that indicate an action to be performed at a certain time (e.g., cancel a timed entry call). This list is sorted on order of increasing time values.

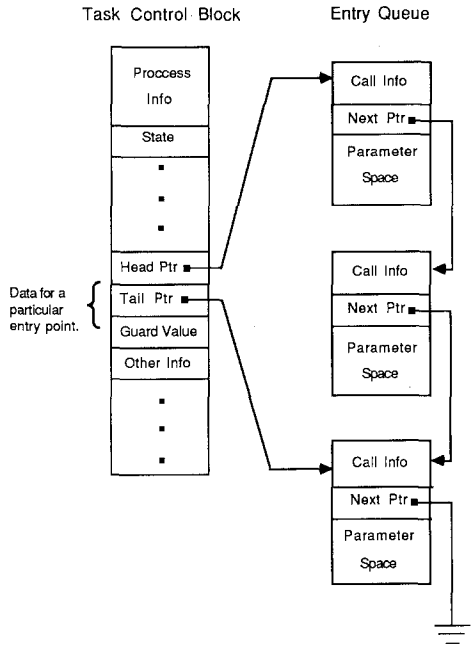

Fig. 4. Task control block and entry queue for an accepting task.

Implementation. The case of the simple call and accept is straightforward and contains the basic operations for all forms of rendezvous. Below we describe a typical implementation of the rendezvous that accounts for some of the variations listed earlier.

The compiler generated code for the calling process begins by assembling a call record and making a call to the run-time system that passes a pointer to this record. The call record is a contiguous block of memory allocated by code that is generated by the compiler. This memory could come from a pool of available storage or be a frame allocated on the calling process' local stack space. The call record's entries include the identity of the caller, the name of the entry called, and either values of parameters or pointers to parameters. If the parameters are passed by reference, pointers are present in the record. If the parameters are passed by value or copy, values are placed in the record. If they are passed by copy, the same parameters space is used by the accepting process to return results.

After the run-time system is invoked and receives the pointer to the call record, the state of the calling process is changed to indicate that it is waiting for rendezvous completion. A check is then made to determine if

the called process is awaiting a call on that entry. If it is, a pointer to the call record in memory is passed to that process and the rendezvous is started by changing the state of the called process. If the process is not able to immediately accept the rendezvous, the call record is inserted into the queue for that entry. The entry queue head and tail pointers are then updated as necessary. When the call is accepted, the pointers for the queue are updated to remove the call record from the linked list. A pointer to this record is given to the accepting process so that it may have access to the parameters and modify them as necessary. At the conclusion of the rendezvous, the run-time system is entered by the accepting process so that the calling process may be made runnable. The code generated for the calling process reads the value of the parameters from the call record (if necessary) and then deallocates this space.

In order to prevent race conditions with multiple processors accessing data, locks are normally used to provide mutual exclusion. They must be acquired by the run-time system before it modifies any shared data structure. A lock is implemented to provide mutual exclusion for each entry queue in the system. In addition to entry queues, the run queue, timed events list, and in some cases, TCBs, also need locks.

Support for Variations. In order to support variations of the rendezvous, the run-time system may need to remove call records from the middle of the queue, or check the parameters of a queued call to see if it is eligible for execution. In the case of Occam, there will be queues for each channel or possible sender-receiver pair which will be either empty or of length one. In other cases, the queues may be of variable length.

In the case of the timed entry call, the call itself is issued in the same manner as the simple call. If it is accepted immediately, no more work is required. If it is not, an additional record is created and put in the timed events list. This record indicates the time when the call is to be canceled, a pointer to the call record that has been inserted into the entry queue, and a unique call identifier. The timed events list is a queue that is sorted in order of increasing times. Whenever the run-time system begins executing, the time-of-day clock is read and the value compared to the first event on the list. If the event's time has passed, it and any others that may be ready are processed.[38] In the case of a timed entry call, a check is made to see if the calling task is still waiting on the same entry call. If it is, the call record is removed from the queue and the task's state is changed to make it runnable. If the call has already been accepted, no action is taken after the timeout.

When an acceptor nondeterministically has to choose between several possible entry calls, the run-time system implements a check of the specified

queues to determine if a call can be accepted. If one is ready, the rendezvous proceeds as usual; otherwise, (depending on the semantics of the language), the accepting process may wait for an eligible call. In this case, it is up to the run-time system to make this process runnable when an eligible call occurs. If a bound is placed on the time a process should wait for an eligible entry call, a record is added to the timed events list. If the timeout occurs and the acceptor is still waiting, its state is changed to runnable so that it may proceed with its execution after the call point.

3. PROTOCOLS FOR MAINTAINING CACHE COHERENCE

A simple software solution to the cache coherence problem is to place all shared writable data in non-cachable storage and, if cache entries are not tagged with a process identifier, to flush a processor's cache each time the processor performs a context switch. Although this scheme does provide coherence, it does this at a very high cost in performance. The classical hardware solution to the cache coherence problem is to broadcast all writes: each cache sends the address of the modified line to all other caches. The other caches invalidate the modified line if they have it. Although this scheme is simple to implement, it is not practical unless the number of processors is very small. As the number of processors is increased, the cache traffic resulting from the broadcasts rapidly becomes prohibitive.

The most practical solutions to the cache coherence problem in a system with a large number of processors use a directory scheme in which the directory information is distributed among the caches. These schemes make it possible to construct systems in which the only limit on the maximum number of processors is that imposed by the total bus and memory bandwidth. These schemes are the snooping cache protocols mentioned earlier.[40] Each cache monitors addresses on the system bus, checking each reference for a possible cache hit.

There are two major classes of protocols for enforcing cache coherence with snooping caches. Both use the snooping hardware to dynamically identify shared writable lines, but they differ in the way in which write operations to shared lines are handled.

In the first class of protocols, when a processor writes to a shared line, the address of the line is broadcast on the bus to all other caches, which then invalidate the line. Two examples are the Illinois protocol and the Berkely Ownership Protocol.[40,41] We refer to this class of protocols as *invalidation* protocols.

In the second class of protocols, when a processor writes to a shared line, the written data is broadcast on the bus to all other caches, which

then update their copies of the line. Cache invalidations are never performed by the cache coherence protocol. Two examples are the protocol in DEC's Firefly multiprocessor workstation and that in the Xerox Dragon multiprocessor.[6,14,42] We refer to this class of protocols as *write-broadcast* protocols.

Each of these two classes of protocol has certain advantages and disadvantages, depending on the pattern of references to the shared data. For a shared data line that tends to be read and written several times in succession by a single processor before a different processor references the same line, the invalidation protocols perform better than the write-broadcast protocols. The invalidation protocols use the bus to invalidate the other copies each time a new processor makes its first reference to the shared line, and then no further bus accesses are necessary until a different processor accesses the line. Invalidation can be performed in a single bus cycle, since only the address of the modified line must be transmitted. The write-broadcast protocols, on the other hand, must use the bus for every write operation to the shared data, even when a single processor writes to the data several times consecutively. Furthermore, multiple bus cycles may be needed for the write, since both an address and data must be transmitted. For example, the DEC Firefly uses multiple cycles for both reads and writes in order to simplify the design of the bus as well as support the write-broadcast protocol.[14] On the other hand, write-invalidation on the Sequent Symmetry can be performed in one bus cycle.[7]

For a shared data line that tends to be read much more than it is written, with writes occurring from random processors, the write-broadcast protocols tend to perform better than the invalidation protocols. The write-broadcast protocols use a single bus operation (which may involve multiple bus cycles) to update all cached copies of the line, and all read operations can be handled directly from the caches with no bus traffic. The invalidation protocols, on the other hand, will invalidate all copies of the line each time it is written, so subsequent cache reads from other processors will miss until they have reloaded the line.

An adaptive protocol that attempts to incorporate some of the best features of each of the two schemes is proposed in Ref. 43. This protocol, called EDWP (Efficient Distributed-Write Protocol), is essentially a write-broadcast protocol with the following modification: if some processor issues three writes to a shared line with no intervening references by any other processors, then all the other cached copies of that line are invalidated and the processor that issued the writes is given exclusive access to the line. The particular number of successive writes before invalidating the line, three, was selected based on a simulated workload model.

4. CACHE COHERENCE PROTOCOL REQUIREMENTS FOR RUN-TIME SUPPORT

4.1. Process Scheduling

A major factor in the protocol requirements for run-time support is the strategy used by the run-time system for processor scheduling when synchronization points are reached. For the purposes of brevity, we limit ourselves to considering two contrasting scheduling strategies that bound the range of possibilities. The first, *run-time scheduling*, assumes all runnable processes are kept in a run queue and are assigned processors as they become available. There are no restrictions as to which processes may run on any processor. Processes may be assigned processors in some priority order, but there is nothing to prevent a process from migrating between several processors during its lifetime. The second, *compile-time scheduling*, permits processes to be bound to processors so that process migration may be eliminated or considerably reduced.

The appropriate scheduling strategy for a particular situation should be influenced by the characteristics of the hardware, which include the bus bandwidth, the number of processors, the homogeneity of the processors, and the size of the processor caches. In cases where bus bandwidth is limited, there are a large number of processors, or special functional units are connected to only certain processors, compile-time scheduling offers advantages if the application allows it to be used. Limited bus bandwidth discourages process migration, as does a situation where there are more processors than processes. When all processor configurations are not homogeneous, process migration may be inhibited by functional requirements. On the other hand, cases where there are fewer processors, more bus bandwidth, or large processor caches, run-time scheduling offers advantages. Additional context-switching with process migration for load balancing is necessary when there are more processes than processors. Additional bus bandwidth supports the additional migration while large caches reduce the burden on the bus by retaining large portions of the working sets of multiple processes.

In addition to hardware, the characteristics of the software application must also be considered. Examples of applications that are best served by run-time scheduling that does not restrict process migration are branch-and-bound algorithms, parallel make, and parallel compile codes. These applications create processes dynamically in a way that cannot be predicted before run-time. The trade-off is between load balancing and bus bandwidth. In contrast, embedded applications and many CAD and scientific codes can be scheduled off-line, and are best served by compile-time

scheduling. Here the load balancing is done *a priori* and bus bandwidth requirements can be minimized. In the case of distributed program development and multi-process simulations, the specific structure of the application suggests which scheduling strategy should be used. Thus, the application domain and hardware configuration influences the process scheduling strategy, which in turn dictates the access patterns to the run-time system data structures.

4.2. Rendezvous Data Access Patterns

We can now examine the access patterns to the run-time system data structures described in Section 2 and assess their relationship to the two basic types of cache coherence protocols described in Section 3, invalidation and write-broadcast. Specifically, we consider the access patterns to shared data structures of multiple processes wishing to perform a rendezvous. We consider these patterns from the point where the rendezvous may actually begin, i.e., when the later of the two tasks involved requests the rendezvous. It is only at this point that processors may begin to share data. In addition to the two cases suggested by considering whether the caller or the acceptor is the later task to request the rendezvous, the case where multiple callers are accessing the entry queue must also be considered. This situation may occur because of the possible asymmetry of the rendezvous: multiple callers may request the same entry point serviced by a single acceptor. We consider this latter case when the acceptor begins accepting a call, but not when the acceptor has not yet reached the point of the rendezvous. When the acceptor is not accessing the entry queue, multiple callers may add entries to the queue without actually sharing data. Therefore, it is sufficient to consider the following cases:

1) one or more callers are blocked with their calls pending on an entry queue, and the accepting process begins to accept the first call (see Fig. 2),
2) a caller issues a call and the accepting process is blocked (Fig. 3 with the possibility of a non-empty queue), and
3) the accepting process is accepting a call while additional callers are requesting a rendezvous (Fig. 2 with additional callers).

Other cases reduce to one of these three here; for example, in the case where the accepting process attempts to accept a call but none are pending, it determines that its entry queue(s) are empty, then blocks and waits for a call to arrive. When a call does arrive, the situation is described by case 2. We consider the cases in more detail here.

Blocked Callers. One or more entry calls are queued for a particular entry and the accepting task begins to accept them. Before accessing the call records in the entry queue, the run-time system, executing on behalf of the accepting task, must first acquire a lock to ensure mutual exclusion. After obtaining the lock for the entry queue, the first call record is removed from the queue by changing the value of the queue head pointer to point to the next call record. This value is obtained from the "next-pointer" field of the first call record (see Fig. 4). After changing this pointer value, the lock is released and the accepting task is free to execute the rendezvous with the data available in the call record. The acceptor may access data in the call record without locking it, since the calling task is blocked until the rendezvous is completed. The calling task is made runnable by the run-time system at the completion of the call by inserting the caller into the run queue. It may then access any return data placed in the call record by the acceptor.

After the run-time system and accepting task complete the rendezvous, several data items associated with the call will reside in the cache of the processor that the accepting task executed on. Some of this data will have been modified, namely the entry queue head pointer, the state value in the TCB of the caller, the tail pointer of the run queue, and possibly locations used for return parameters.

We describe the accessing of this data by the run-time system as *serialized* sharing, because the shared data is accessed at synchronization points in a mutually exclusive fashion by the calling and accepting processes. Of the shared data structures modified in accepting a call, only the entry queue head pointer and run queue tail pointer are likely to be accessed again by the accepting process. The call record will not be accessed again by the acceptor, because a new call record containing the parameter space is accessed only in response to each particular instance of a rendezvous. The caller returns this record to the pool of available storage, or otherwise deallocates it by moving its stack pointer. It is possible, however, that a compiler optimization may allow the caller to reuse the parameter space in conjunction with a later call. An access by the acceptor to this same space would most likely occur much later in time, only as part of a different rendezvous. The same can also be said about the state value stored in the caller's TCB. Situations where this may be a useful optimization are where the acceptor is part of a server task that periodically polls its rendezvous entry points.

In examining the effects of cache protocols on the cache hits to shared data, we must again consider the policy used for processor scheduling. The key issue to be considered in determining an appropriate coherence protocol is whether processes are switched out when they become blocked.

In the case where processes are switched out, the use of an invalidation protocol provides the desired effect. When the calling process blocks after issuing the call, its processor is reassigned to another task, which presumably is executing with a different working set. When a write is made to the call record or caller's TCB by the acceptor, any copies of this data in other caches are invalidated, and thus useless updates are avoided to cache memory that is not being actively shared.

In the case where processes are suspended and not switched out at synchronization points (or at least not allowed to migrate), additional cache hits on both processors will occur if a write-broadcast protocol is used. This approach woud make the caller's task state and return parameters available in the caller's cache after the rendezvous, since the data is accessed by the caller's processor and cached before the rendezvous. A greater benefit occurs in this situation if the caller and acceptor are frequently communicating using the same entry point, and the caller repeatedly reuses the same memory space for the call record (e.g., the compiler optimization mentioned previously). In this case, the acceptor has possession of the call record on its cache the entire time, and it is always up to date. This is because the initialization by the caller is broadcast to the acceptor's processor cache. Further, any data items local to the caller that are modified by the acceptor indirectly through the call record (e.g., parameters passed by reference) will also be resident in both caches.

Blocked Receiver. One or more callers are attempting to access an entry queue in order to place a call record in it while the accepting process is blocked. The callers simply acquire the lock protecting the entry data, read the desired queue tail pointer, modify the last call record's next-pointer to point to the new call record, and modify the tail pointer to also point to the new call record (see Fig. 4). At this point, the lock is released, and the calling process blocks awaiting completion or cancellation of the call. Another calling process may now acquire the lock and perform the same queue operation.

After queueing a call record, the processor executing on behalf of the caller has several modified data items in its cache. These items include the call record, the entry queue's tail pointer, the state value in the TCB of the caller, and the next-pointer of the next to last call record in the queue (if there is one). The next accesses to the call record, state value, and the preceding call record's next-pointer will be the accepting task and run-time system as described in the blocked callers case. The next access to the entry queue tail pointer and the call record's next-pointer will be the next calling task.

In examining the data structure access patterns from the caller's point

of view, we again see that its behavior can be described as serialized sharing. After constructing a call record, the caller obtains a lock, modifies an entry queue, releases the lock, and then blocks. The accepting task locks and modifies data in the same manner. Because the access to shared data is serialized, the requirements for the cache coherence protocol are based on the scheduling strategy. As noted above in the blocked caller's case, the invalidation protocol works best when processes are switched out and processors reassigned at synchronization points. Since the access patterns to the shared data in this case are such that no process performs repeated modifications without blocking, the invalidation protocol is preferred.

In the case of compile-time scheduling, when process migration is restricted, a processor executing on behalf of the caller will reaccess data that has been shared and modified, namely the call record, queue tail pointer, and state value in the TCB. Use of a write-broadcast scheme in this case will increase cache hits as stated above. If the call records are reused as described in the blocked callers case, both the parameter space and the next-pointer will be modified several times over the course of several calls.

Caller and Receiver Executing. An accepting process attempts to dequeue a call record while a different calling process is attempting to enqueue a call record. If the caller gets a lock on the entry queue first, the situation is similar to the second case earlier. When the acceptor does obtain the lock, it dequeues the first call record as described in the blocked callers case mentioned previously. After the acceptor releases the lock on the queue, it may execute the critical section or be suspended by the scheduler if it wishes to reassign that processor. At this point, any calling tasks can access and modify the tail end of the queue as described in the blocked receiver case.

In considering this case where multiple processes are actively attempting to modify a single entry queue, the points made above about scheduling and coherence protocols are still valid. They are valid because the access to the entry queue is serialized through the use of mutual exclusion, and the patterns of access in relation to synchronization points are unchanged. A key point to highlight here, though, is that while calling processes are modifying the tail pointer for the entry queue, the accepting process is modifying the head pointer. The data that is truly shared between caller and acceptor (e.g., call record and state value in the TCB) is done so on a single caller to single acceptor basis. As stated earlier, there are synchronization points before and after each modification to this shared data. Therefore, the invalidation protocol appears best for the case

of heavy context switching with migration, while the write-broadcast shows advantages for the case of non-migrating processes.

Timed Call Considerations. In the case of the timed entry call, a record must also be inserted into the timed events list. When the timeout value expires, it is possible that an entry record will have to be removed from a queue, and that record could be anywhere in the queue. To remove it, the forward pointer of the previous record in the queue must be changed to point to the following call record. It may also be necessary to change the head and tail pointers.

The accesses to the timed events queue and entry queues for call cancellation is highly random. In the case where any processor may be processing a timeout, it appears that the use of an invalidation protocol is best, since it avoids broadcast updates to the timed events list to processors that are no longer using the data structure. If, however, the timing processing is done in a dedicated processor, it is desirable to keep a current copy of the timed events list in its cache, and allow it to broadcast entry queue changes to the other processors, since there is some chance the queue records in question will be accessed again in those processors.

4.3. The Use of Locks

There are many cases where the run-time system must obtain a lock in its shared data structures before accessing them. In the case of hardware locks, this operation is straightforward and does not involve the cache. Such a hardware lock mechanism is described in Ref. 39. However, in most cases, synchronization primitives involve access to a word in memory. The non-interruptible instruction *test & set* is a common example of a mutual exclusion primitive that involves a memory access. If many processors are contending for a lock using test & set, a ping-pong effect can set in where the lock variable is being written into many caches one after another, with other copies being unnecessarily invalidated or updated. Bus traffic in this case can be significantly reduced by using *test & test & set* to implement locks in both invalidation and write-broadcast cases.[44,45]

5. CONCLUSION

The data access patterns for rendezvous and other run-time system data structures, and consequently the cache coherence requirements, are strongly influenced by the process scheduling strategy used by the run-time system at rendezvous synchronization points. Based on our examination of these patterns, we have reached the following conclusions:

- Use invalidation with unrestricted migration (run-time scheduling), especially as it is usually less expensive in bus cycles than write-broadcast.
- Use write-broadcast with restricted migration (compile-time scheduling) when little context switching is performed.

In some case, however, it may be possible to gain the benefits of write-broadcast when using run-time scheduling. If the scheduler can be modified to incorporate processor *affinity*,[46] a restriction in migration is possible. When affinity is used, the scheduler attempts to reschedule processes on the same processor they used previously. Affinity can be used to varying degrees, including an elimination of migration altogether. In this case, a write-broadcast protocol would be best suited for rendezvous execution.

It may also be desirable to use multiple coherence protocols. For example, when multiple heavy weight processes are executing, each of which is made up of lightweight processes communicating by rendezvous, and if the number of heavyweight processes is allowed to vary at run-time, a write-broadcast protocol could be used for data shared between lightweight processes that generally do not migrate, while an invalidation protocol would be used for sharing between heavyweight processes. The design of the IEEE Futurebus[47] is intended to support multiple coherence protocols simultaneously, including both write-invalidate and write-broadcast protocols. It is possible for different processors to observe different protocols, and for any processor to switch to a different protocol at run-time. Coherence protocol actions can also be controlled in software, where the need for coherence actions causes a fault and transfer of control to an interrupt routine. Such an approach is used in the VMP multiprocessor prototype.[48] An adaptive protocol such as the EDWP scheme mentioned in Section 3 may also be appropriate.

Finally, these conclusions are drawn primarily from our experience with Ada. It may be necessary to employ different approaches when using other parallel languages. Again, the lack of suitable traces or program statement frequencies has prevented us from examining the effects of different semantics for other parallel languages. It is our desire to see that this situation is rectified in the near future.

REFERENCES

1. F. Basket and J. L. Hennessy, Small shared-memory multiprocessors, *Science* **231**:963–967 (February 1986).
2. C. G. Bell, Multis: A new class of multiprocessor computers, *Science*, **228**(4698):462–467 (April 1985).

3. G. Fielland and D. Rodgers, 32-bit computer system shares load equally among up to 12 processors, *Electronics Design*, pp. 153–168 (September 1984).
4. E. C. Corporation, *Multimax Technical Summary*, 257 Cedar Hill Street, Marlboro, Massachusetts 01752, REV A edition (May 1985).
5. P. Bitar and A. M. Despain, Multiprocessor cache synchronization, issues, innovations, evolution, in *Proc. of the 13th Int'l. Symp. on Computer Archit.*, pp. 424–433 (June 1986).
6. J. Archibald and J. L. Baer, Cache coherence protocols: Evaluation using a multiprocessor simulation model, *ACM Transactions on Computer Systems* **4**(4):273–298 (November 1986).
7. T. Lovett and S. Thakkar, The Symmetry multiprocessor system, in *Proc. of the 1988 Int.'l. Conf. on Parallel Processing*, pp. 303–310 (August 1988).
8. M. Annaratone and R. Rühl, Performance measurements on a commercial multiprocessor running parallel code, in *Proc. of the 16th Int'l. Symp. on Computer Archit.*, pp. 307–314 (June 1989).
9. A. H. Karp and R. G. Babb, A comparison of 12 parallel FORTRAN dialects, *IEEE Software*, pp. 52–67 (September 1988).
10. A. Osterhaug, *Guide to Parallel Programming*, Sequent Computer Systems, Inc. (1985).
11. R. H. Perrott, *Parallel Programming*, Addison-Wesley (1987).
12. R. G. Babb II, ed., *Programming Parallel Processors*, Addison-Wesley, Reading, Massachusetts (1988).
13. M. Kallstrom and S. S. Thakkar, Programming three parallel computers, *IEEE Software*, pp. 11–22 (January 1988).
14. C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite, Jr., Firefly: a multiprocessor workstation, *IEEE Transactions on Computers* **37**(8):909–920 (August 1988).
15. *IRIS-4D Series Owner's Guide*, Silicon Graphics, Inc., Version 4.0, Document Number 007-320-040 edition (1989).
16. *The Series 10000 Personal Supercomputer*, Apollo Computer Inc. (1988).
17. T. Diede, C. F. Hagenmaier, G. S. Miranker, J. J. Rubinstein, and W. S. Worley, Jr., The Titan graphics supercomputer architecture, *IEEE Computer*, pp. 13–30 (September 1988).
18. *Ada Programming Language* (ANSI-MIL-STD-1815A), Department of Defense, OUSD (R & D) (January 1983).
19. N. H. Gehani and W. D. Roome, Concurrent C, *Software Practice and Experience* **16**(9):821–844 (September 1986).
20. D. May, Occam, *ACM SIGPLAN Notices* **18**(4):69–79 (April 1983).
21. G. R. Andrews, R. A. Olsson, M. Coffin, I. Elshoff, K. Nilson, T. Purdin, and G. Townsend, An overview of the SR language and implementation, *ACM Transactions on Programming Languages and Systems* **10**(1):51–86 (January 1988).
22. M. L. Scott, Language support for loosely coupled distributed programs, *IEEE Transactions on Software Engineering* **SE-13**(1):88–103 (January 1987).
23. F. N. Parr and R. E. Strom, NIL: A high-level language for distributed systems programming, *IBM Systems Journal* **22**(1, 2) (April 1983).
24. S. J. Eggers and R. H. Katz, A characterization of sharing in parallel programs and its application to coherency protocol evaluation, in *Proc. of the 15th Int'l. Symp. on Computer Archit.*, pp. 373–382 (June 1988).
25. S. J. Eggers and R. H. Katz, The effect of sharing on the cache and bus performance of parallel programs, in *Proc. of the Third Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pp. 257–270 (April 1989).
26. R. L. Sites and A. Agarwal, Multiprocessor cache analysis using ATUM, in *Proc. of the 15th Int'l. Symp. on Computer Archit.*, pp. 186–195 (June 1988).

27. B. Beck and D. Olien, A parallel-programming process model, *IEEE Software*, pp. 63–72 (May 1989).
28. R. H. Thomas and W. Crowther, The Uniform System: An approach to runtime support for large scale shared memory parallel processors, in *Proc. of the 1988 Int'l. Conf. on Parallel Processing*, pp. 245–254 (August 1988).
29. M. L. Scott, T. J. LeBlanc, and B. D. Marsh, Design rationale for Psyche, a general-purpose multiprocessor operating system, in *Proc. of the 1988 Int'l. Conf. on Parallel Processing*, pp. 255–262 (August 1988).
30. R. P. Weicker, Dhrystone: A synthetic systems programming benchmark, *Communications of the ACM* **27**(10):1013–1030 (October 1984).
31. G. Graunke and S. Thakkar, Synchronization algorithms for shared-memory multiprocessors, *IEEE Computer*, pp. 60–69 (June 1990).
32. R. M. Clapp and T. N. Mudge, Ada on a hypercube, in *Proc. of The Third Conf. on Hypercube Concurrent Computers and Applications*, pp. 399–408, Pasadena, California (January 1988).
33. R. A. Volz and T. N. Mudge, Timing issues in the distributed execution of Ada programs, *IEEE Transactions on Computers* **C-36**(4):449–459 (April 1987).
34. R. A. Volz, T. N. Mudge, G. D. Buzzard, and P. Krishnan, Translation and execution of distributed Ada programs: Is it still Ada?, *IEEE Transactions on Software Engineering* **SE-15**(3):281–292 (March 1989).
35. R. M. Clapp and T. N. Mudge, A parallel language for a distributed-memory multiprocessor, in *Proc. of The Fourth Conf. on Hypercube Concurrent Computers and Applications*, pp. 515–522, Monterey, California (March 1989).
36. J. D. Ichbiah, J. G. P. Barnes, J. C. Heliard, B. Krieg-Breckner, O. Roubine, and B. A. Wichmann, Rationale for the design of the Ada programming language, *ACM SIGPLAN Notices*, Vol 14, No. 6 (June 1979).
37. N. H. Gehani and W. D. Roome, Rendezvous facilities: Concurrent C and the Ada language, *IEEE Transactions on Software Engineering* **SE-14**(11):1546–1553 (November 1988).
38. R. A. Volz and T. N. Mudge, Instruction level mechanisms for accurate real-time task scheduling, *IEEE Transactions on Computers* **C-36**(8):988–993 (August 1987).
39. B. Beck, B. Kasten, and S. Thakkar, VLSI assist for a multiprocessor, in *Proc. of the Second Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pp. 10–20 (October 1987).
40. R. H. Katz, S. J. Eggers, D. Wood, C. L. Perkins, and R. Sheldon, Implementing a cache consistency protocol, in *Proc. of the 12th Int'l. Symp. on Computer Archit.*, pp. 276–283 (June 1985).
41. M. S. Papamarcos and J. H. Patel, A low-overhead coherence solution for multiprocessors with private cache memories, in *Proc. of the 11th Int'l. Symp. on Computer Archit.*, pp. 348–354 (June 1984).
42. R. R. Atkinson and E. M. McCreight, The Dragon processor, in *Proc. of the Second Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pp. 65–69 (October 1987).
43. J. K. Archibald, A cache coherence approach for large multiprocessor systems, in *Proc. of the 1988 Int'l. Conf. on Supercomputing*, pp. 337–345. ACM Press (July 1988).
44. M. Dubois, C. Scheurich, and F. A. Briggs, Synchronization, coherence, and event ordering in multiprocessors, *IEEE Computer*, pp. 9–21 (February 1988).
45. L. Rudolph and Z. Segall, Dynamic decentralized cache schemes, in *Proc. of the 11th Int'l. Symp. on Computer Archit.*, pp. 340–347 (June 1984).
46. S. S. Thakkar and M. Sweiger, Performance of an OLTP application on Symmetry multi-

processor system, in *Proc. of the 17th Int'l. Symp. on Computer Archit.*, pp. 228–238 (June 1990).

47. P. Sweazy and A. J. Smith, A class of compatible cache consistency protocols and their support by the IEEE Futurebus, in *Proc. of the IEEE 13th Annual Int'l. Symp. on Computer Archit.*, pp. 414–423 (June 1986).
48. D. R. Cheriton, G. A. Slavenburg, and P. D. Boyle, Software controlled caches in the VMP multiprocessor, in *Proc. of the 13th Int'l. Symp. on Computer Archit.*, pp. 366–374 (June 1986).