Debugging Parallel Programs with Instant Replay

THOMAS J. LEBLANC AND JOHN M. MELLOR-CRUMMEY

Abstract—**The debugging cycle is the most common methodology for finding and correcting errors in sequential programs. Cyclic debugging is effective because sequential programs are usually deterministic. Debugging parallel programs is considerably more difficult because successive executions of the same program often do not produce the same results. In this paper we present a general solution for reproducing the execution behavior of parallel programs, termed *Instant Replay*. During program execution we save the relative order of significant events as they occur, not the data associated with such events. As a result, our approach requires less time and space to save the information needed for program replay than other methods. Our technique is not dependent on any particular form of interprocess communication. It provides for replay of an entire program, rather than individual processes in isolation. No centralized bottlenecks are introduced and there is no need for synchronized clocks or a globally consistent logical time. We describe a prototype implementation of Instant Replay on the BBN Butterfly™ Parallel Processor, and discuss how it can be incorporated into the debugging cycle for parallel programs.**

Index Terms—**CREW protocols, distributed debugging, execution replay, parallel programming, program instrumentation, shared objects.**

I. Introduction

DEBUGGING sequential programs is a well-understood task that draws on tools and techniques developed over many years. One early technique was to record snapshots of the entire program state, often consisting of many pages of hexadecimal digits, for perusal by the programmer. Debugging was a programmer-intensive operation, since there were few tools for analyzing the program state. Over time this approach was replaced by interactive debuggers, which allow the programmer to examine arbitrary details of the program state during execution. Debugging became more computation-intensive, since the computer was used to reproduce execution sequences with successively greater detail. As a result, the most common methodology used today to debug sequential programs is cyclic: the program is executed until an error manifests itself, the programmer postulates a set of underlying causes for the error, trace statements or additional breakpoints are inserted to gather more information about the causes of the error, and the program is reexecuted. This technique is effective because sequential programs are usually deterministic. That is, for a fixed input, each execution of a program will always follow the same execution path and produce the same results.

Debugging parallel programs is considerably more difficult because parallel programs are often not deterministic.[1] In our model parallel programs consist of multiple asynchronous processes that communicate using some form of message-passing or shared memory. *No assumption may be made about the relative speed of processes;* we can only assume finite progress by each process. Since parallel programs do not fully specify all possible execution sequences, the execution behavior of a parallel program in response to a fixed input may be indeterminate, with the results depending on a particular resolution of race conditions existing among processes. Therefore, cyclic debugging techniques for error isolation are not guaranteed to work because successive executions of the same parallel program may not produce the same results. We are left with two options for debugging parallel programs: we can either take snapshots of the program state during execution for later examination or we can provide a mechanism that guarantees reproducible behavior of parallel programs. Only the latter approach allows reliable use of cyclic debugging techniques.

The first alternative, in which the programmer analyzes snapshots of program state taken during execution, recognizes that multiple executions of parallel programs are indeterminate, therefore, all information necessary to diagnose program errors must be collected during a single execution. *Behavioral Abstraction* (BA) is typical of this approach [2]. BA provides a mechanism for the hierarchical definition of events in terms of sequences of primitive events that can occur during program execution. An event recognition tool monitors the stream of primitive events that occur during program execution and presents the user with an abstract view of the program's behavior in terms of a sequence of hierarchically-defined events. There are two disadvantages to this technique. First, BA requires that a user exhaustively describe interesting events which take place during execution in terms of a bottom-up specification. In creating the specification, the user must anticipate all interesting events related to an error before execution; there is no mechanism for gathering additional information about an error after it is observed. Second, the amount of information gathered tends to be voluminous. Since the technique is not cyclic, the user must collect enough

Manuscript received September 3, 1986; revised December 5, 1986. This work was supported by the National Science Foundation under Grant DCR-8320136 and DARPA/ETL under Grant DACA76-85-C-0001. The Xerox Corporation University Grants Program provided equipment used in the preparation of this paper. J. Mellor-Crummey was supported in part by a Sproull Fellowship awarded by the University of Rochester.

The authors are with the Department of Computer Science, University of Rochester, Rochester, NY 14627.

[1] We are interested in programs that exhibit *true parallelism* or, at the very least, appear to exhibit parallelism due to preemptive scheduling of processes. A concurrent program implemented by coroutines running on a single processor without the possibility of preemption can be debugged as if it were a sequential program.

information during execution to diagnose any error that might arise. Other work based on one-shot execution of parallel programs has the same limitation [1], [9], [21].

The second alternative for debugging parallel programs is based on reproducible program execution, which allows cyclic debugging techniques to be applied. Reproducible program behavior has been studied in several domains, including concurrent programs using semaphores and monitors for communication, systems based on nested atomic transactions, and systems comprised of loosely coupled processes that communicate via messages.

Carver and Tai have considered repeatable execution for programs consisting of concurrent processes that interact through semaphores and monitors [3]. In their approach, execution of a concurrent program is characterized by a sequence of P operations (termed a *P sequence*) on shared semaphores. The same idea can be used to produce an M sequence for monitors, which records a series of starts of monitor procedures. A P sequence is a sequence of ordered pairs; each pair corresponds to a P operation on a specific semaphore by a specific process. Thus, a P sequence is a total order of *all* synchronization operations that occur in a program. P sequences can be created by the programmer to test specific synchronization sequences of a concurrent program or can be reproduced during execution to provide repeatable execution. The disadvantage of this approach is that it requires that all P operations be serialized, thereby losing much of the potential for parallelism that exists in a program. While adequate for single processor systems that simulate concurrency, this technique would not be useful in a truly parallel environment. There, the serialization constraint could have such an impact on program performance that it would be impractical to monitor programs during normal execution. Use of this method would be then relegated to a distinct testing and debugging phase.

Chiu's technique for replaying a program's execution in an atomic transaction system involves checkpointing each version of all atomic objects and recording a timestamp for each atomic action during program execution [5]. A debugger uses this information to traverse action trees (corresponding to the nested atomic actions of a program execution) according to a serialization of their constituent atomic actions. Traversing an action tree permits viewing the state of atomic objects before and after each atomic update, as well as replaying execution through action sequences to isolate program flaws. The major drawback of this work is that the techniques are restricted to computations structured in terms of nested atomic actions. In addition, these techniques require significant storage overhead to maintain the necessary checkpoints of atomic objects, although the checkpoints may be required for recovery actions anyway.

Methods to reproduce the execution behavior of programs comprised of loosely coupled processes that communicate using messages typically require that the contents of each message be recorded in an event log as it is received [7], [13], [24]. The programmer can either review the events (messages) in the log, in an attempt to isolate errors, or the events can be used as input to replay the execution of a process in isolation. A similar event logging approach has also been used to monitor program performance [16]. There are several disadvantages to this approach. First, it has only been used in loosely coupled systems and there are reasons to believe it would not be well-suited to tightly coupled systems. Although the amount of data exchanged in messages could be very large, this technique exploits the fact that communication in loosely coupled systems takes place infrequently, primarily because of the high cost of communication. The additional time necessary to copy a message into an event log in local memory does not seriously affect performance when compared with the time required to send a message. This assumption does not necessarily apply to tightly coupled systems, where the cost of communication is lower, allowing more frequent communication. Another disadvantage is that the space requirements for the event log tend to be very large. Again, within the domain of loosely coupled processes, it is reasonable to assume the logs will grow slowly enough that they can be buffered in memory and then stored on external devices without seriously affecting the performance of the program. The third, and most important drawback, is that it is difficult to examine the global effects of process interactions using this technique, since the replay mechanism only operates on a single process in isolation. Previous attempts to replay groups of processes using this scheme require that a network-wide consistent time be maintained [7].

In this paper we present a general solution for reproducing the execution behavior of parallel programs, termed *Instant Replay*. Our emphasis is on providing repeatable execution of highly parallel programs in tightly coupled systems, although our approach naturally extends to loosely coupled systems. During program execution we save the relative order of significant events as they occur, not the data associated with such events. Since we do not require the contents of all process interactions (e.g., messages) to be saved, our approach requires less time and space to save the information needed for program replay than other methods. Our technique guarantees reproducible program behavior during the debugging cycle by using the same input from the external environment and by imposing the same relative order on events during replay that occurred during the original execution. Unlike previous techniques, Instant Replay is not dependent on the particular form of interprocess communication used. In addition, we provide replay for an entire program, rather than individual processes in isolation. Finally, we avoid introducing any global synchronization of events through the use of a fully distributed protocol; there is no centralized bottleneck and no need for synchronized clocks or a globally-consistent logical time. With these properties, Instant Replay is especially useful for debugging parallel programs on tightly coupled multiprocessors, where interprocess communication is cheaper, and therefore more frequent, than in loosely coupled systems.

In the next section we present Instant Replay, including our goals, assumptions, and approach. Section III describes a prototype implementation on the BBN Butterfly, a tightly coupled multiprocessor comprised of 128 MC68000 processors. In Section IV we discuss how Instant Replay can be incorporated into the debugging cycle for parallel programs.

Section V summarizes the advantages of our approach and describes our plans for future work.

II. Instant Replay

When debugging a sequential program, one can usually guarantee reproducible program execution by supplying the same input each time the program is executed. Successive executions with the same input produce the same behavior because sequential programs tend to be deterministic. The same is true of the individual processes in a parallel program.[2] If each process is supplied the same input values (corresponding to the contents of messages received or the values of shared memory locations referenced) in the same order during successive executions, it will produce the same behavior each time. In particular, each process will produce the same output values in the same order. Each of those output values may then serve as an input value for some other process. Therefore, in order to debug a parallel program, we do not need to store all input values for each process in an event log, since any input value corresponding to some output value can be recomputed during replay. By ensuring that each process sees the same input values at every step of execution, all processes will exhibit the same execution behavior during both the monitoring phase and replay. Instant Replay is based on this observation.

In our approach, all interactions between processes are modeled as operations on shared objects. A series of modifications to a shared object is represented as a totally ordered sequence of versions. Each version has a corresponding version number, which is unique with respect to a particular object. During normal program execution (i.e., *the monitoring phase*) we record a partial order of the accesses to each object. (It is a partial order because we do not need to impose an ordering on multiple processes that read a particular version of a shared object.) This partial order is specified by a sequence of version numbers for each object. To record the partial order the system maintains the current version number for each object and the number of readers for each version of each object. In addition, each process records the version number of each shared object it accesses. During program replay, we allow each process to recompute its output values, thereby providing input values for other processes. We use the record of object accesses recorded by each process to ensure that the same version of input values used by the process during the monitoring phase is used during replay. As long as the recorded information is available, the original program execution can be repeated over and over.

Our goal is to provide a flexible monitoring system, applicable in both loosely coupled and tightly coupled environments, that allows a programmer to replay arbitrary execution sequences produced by a parallel program. Since we cannot predict when it may be desirable to replay a particular execution sequence, it must be practical for the monitoring mechanisms to be in place during every execution. Therefore, our mechanisms must have minimal impact on program performance. Instant Replay provides reproducible behavior of parallel programs with minimal impact on performance by a) simulating the original external environment during replay, b) modeling all interprocess events as operations on shared data, subsuming both shared-memory and message-passing primitives, c) recording only the version number of data values that are input to each process, not the values themselves, thereby minimizing the amount of information recorded, and d) using a distributed data collection mechanism, so that no central bottleneck is present when a program is being monitored or replayed. We will explore each of these aspects in the following sections.

A. Simulating the External Environment

As with any cyclic debugging system, we assume that the original execution of a program and subsequent replays occur in equivalent virtual machine environments. Two virtual machines A and B are said to be equivalent with respect to program P if program P can exhibit the same behavior whether executed on virtual machine A or B. For practical reasons, we do not require equivalent physical machine states, since that would include the contents of all external devices, the exact value of the clock, and the internal states of all components. In particular, A and B need not have identical real-time clock values if P's execution does not depend on the real-time clock. Similarly, the contents of file F on machine A and B can differ if P does not attempt to reference F. If program P depends on physical details of its virtual machine during execution, it becomes difficult, if not impossible, to simulate the virtual machine during replay.

Real-time programs, in particular, may not be good candidates for Instant Replay because it is so difficult to simulate equivalent virtual machines.[3] We require that programs receive identical input from the environment during both execution and replay. However, it is not sufficient simply to supply the same input to the process, we must also supply it at the same points during program execution. This can be very difficult for real-time programs since they often receive input as a result of asynchronous interrupts. Without making special provisions to record when interrupts occur during program execution, which could severely degrade performance, we cannot accurately simulate the original virtual machine environment.

It is important to note that the problem of finding equivalent virtual machines also arises when debugging sequential programs; it is orthogonal to the specific problem of debugging parallel programs. We do not depend on a particular simulation of virtual machines, so any techniques developed for sequential program debugging can probably be used. Specifically, we assume that programs do not exploit the physical characteristics of any resources allocated by the system, therefore, we need only ensure that the amount of resources available during replay is at least the amount used by the program during the original execution. Any unsuccessful attempt to allocate resources during execution can be recorded, so that the same behavior can be recreated during replay.

[2] For now, we assume that processes do not contain nondeterministic statements. In particular, processes do not allow asynchronous interrupts.

[3] To our knowledge, *no* significant software debugging system exists for real-time programs.

B. Communication Through Shared Objects

If processes in a parallel program do not communicate, each process can be debugged using traditional techniques, since other processes in the program would have no effect on the execution path of a particular process. It is only when processes interact, via communication and synchronization primitives, that the potential for nonrepeatable behavior arises. Examples of process interactions include *P* and *V* primitives applied to a shared semaphore [8], monitor entry procedures [10], send/receive message primitives, and general sharing of memory. Instant Replay models all process interactions in a parallel program as operations on shared data. This characterization of process interactions is not restrictive since all communication and synchronization primitives can be reduced to operations on shared data. In particular, message passing can be modeled as communication through a shared port, mailbox, or memory segment.

Our approach exploits the fact that values exchanged between processes via shared data depend only on the initial values in shared objects, the order in which processes are granted access to shared objects, and the deterministic nature of processes. Operations on shared data objects can be separated into two classes: read operations, which do not change the state of an object, and write operations, which do. By recording the sequence of write operations on each shared object, it is possible to recreate the proper sequence of state transitions for all shared objects during program replay. Similarly, by recording the version number of each shared object read by a process, it is possible to recreate the proper input values for that process during replay. This is exactly the information we record during the monitoring phase.

Instant Replay requires that the set of operations on each shared object have a valid serialization. A set of operations has a valid serialization if the result of each individual operation is the same as it would be if the operations had all been executed in some sequential order. A protocol that ensures a valid serialization, such as a concurrent-read-exclusive-write (CREW) protocol [6], must be used for access to each shared object. In choosing a protocol, we look for one that guarantees serializability, while exerting minimal impact on shared object access and allowing maximal parallelism. If an access protocol that guarantees serializability for operations on shared objects is already present in the application or the system, it is not necessary to superimpose another. Therefore, our techniques are applicable to programs that incorporate results of current research efforts on how to structure interprocess communication to admit the most parallelism. For example, Lamport [12], Peterson [20], and Vitanyi and Awerbuch [25] present algorithmic solutions for the concurrent-reading-while-writing (CRWW) problem that permit concurrency among readers and writers, as well as among writers themselves. Instrumentation for Instant Replay can be added to systems that use such protocols, if a serialization order of operations on each shared object can be determined.[4]

[4] For a serialization of operations on a shared object to be possible the object must be *regular* [12]. An object is *regular* when all reads not concurrent with a write get correct values, and any read that overlaps a series of writes obtains either the value of the object before the first of the writes, or one of the values being written.

For the remainder of this paper, we will illustrate our technique using a CREW protocol for access to shared objects. A CREW protocol ensures a total order of writers with respect to each shared object, a total order of readers with respect to writers of each shared object, and a partial order of readers with respect to each shared object. Although we could use a protocol that requires mutually exclusive (ME) access to shared objects, resulting in a total order on accesses to each object, many parallel programs allow concurrent readers. An exclusive access protocol would artificially limit the parallelism in such programs. Since the execution path of a program can be characterized by a partial order on the operations with respect to each shared object, we will not require a total order.

In addition to being independent of a particular protocol, Instant Replay does not rely on a particular granularity of interprocess communication. The granularity of access to shared objects is implementation-dependent. Message-passing systems only require the protocol during shared buffer access; shared-memory systems may require the protocol to be used whenever shared storage is referenced.

C. Data Structures for Program Monitoring

In order to record the partial order of accesses to objects that characterizes an execution, we use a set of *process history tapes*. During the monitoring phase, a process history tape is used to record the version number of each shared object accessed by a process; it is modified only by the corresponding process. Since the relevant information is read and recorded as part of the access to an object, the monitoring phase imitates whatever parallelism is exhibited by the application.

Each history tape has a header containing several fields: a pointer to the current square on the tape, a pointer to the last non-blank square on the tape, and a pointer to the initial square on the tape. The two operations that can be applied to a history tape are ReadHistoryTape, which reads the value written in the current square, and WriteHistoryTape, which writes a value in the current square. Each of these operations advances the current square pointer of the tape.

Upon creation, each shared object is assigned a version number of 0. Also upon creation, each process is assigned a history tape that is initially blank. During each read or write operation on a shared object by a process, information about the object is recorded on the process's history tape. All history tapes created during the execution of a parallel program are linked together to form a tree. Each time a process spawns a child, a reference to the history tape of the child process is recorded on the history tape of the parent. This organization of history tapes enables each process history tape to be associated with the correct process during execution replay.

In addition to the information recorded on a process's history tape regarding interactions with shared objects and child processes, arbitrary details of a process's execution can be recorded on the tape for use during replay. Specifically, the resolution of certain interesting events can be recorded on the history tape in order to replay programs containing nondeterminism. The information recorded about such events can be used to recreate the same event during program replay. A mechanism to support the recording of these events would need to be added to the implementation of the programming

```
ReaderEntry (object,process);
   if mode = MONITOR then
      P(object.lock);
      AtomicAdd(object.activeReaders, 1);
      V(object.lock);
      WriteHistoryTape(process,object.version);
   else
      /* Find out version read during monitoring phase */
      key := ReadHistoryTape(process);
      while object.version != key do delay
   end if;
end ReaderEntry;
ReaderExit (object);
   AtomicAdd(object.totalReaders,1);
   /* Ignored in replay mode */
   AtomicAdd(object.activeReaders, -1);
end ReaderExit;
```

Fig. 1.

```
WriterEntry (object, process);
   if mode = MONITOR then
      P(object.lock);
      /* Wait for all current readers to finish */
      while object.activeReaders != 0 do delay;
      WriteHistoryTape(process,object.version);
      WriteHistoryTape(process,object.totalReaders);
   else
      /* Read version modified during monitoring phase */
      key := ReadHistoryTape(process);
      while object.version != key do delay;
      /* Read count of readers for previous version */
      key := ReadHistoryTape(process);
      while object.totalReaders < key do delay;
   end if;
end WriterEntry;
WriterExit (object);
   object.totalReaders := 0;
   if mode = MONITOR then
      object.version += 1;
      V(object.lock);
   else
      AtomicAdd(object.version, 1);
   end if;
end WriterExit;
```

Fig. 2.

language at the appropriate level (i.e., compiler code generation or language runtime support). Such a mechanism would be appropriate to record the statement alternative chosen in a nondeterministic selection statement, whether or not a timeout interval had expired during execution, and clock values returned by system calls.

D. Access Protocols for Shared Objects

In order to properly record a partial order of the accesses to each shared object, a protocol that ensures a valid serialization is needed. In this section we will describe such a protocol, a concurrent-read-exclusive-write (CREW) protocol that can be used to implement Instant Replay.

The CREW access protocol for shared objects consists of four procedures: entry and exit procedures for readers, and entry and exit procedures for writers. During the monitoring phase, these procedures enforce a CREW access protocol on shared objects and record a partial order of accesses to each shared object. During the replay phase, these same procedures are used to enforce the partial order recorded during the monitoring phase.

Each process that reads a shared object must use the entry procedure **ReaderEntry** (Fig. 1). This routine uses a semaphore associated with the object to ensure that readers do not attempt to access that object while a writer is using it. Once the reader is granted access by the semaphore, it increments the number of active readers using the object.[5] Writers are not allowed to modify the object as long as the count of active readers is nonzero. Once the count of active readers has been updated, the reader process releases the semaphore and records the version of the object it is about to read on its process history tape. Then, the reader is allowed to access the object. Eventually, the exit routine **ReaderExit** (also in Fig. 1) is called, which simply maintains a count of all readers for a particular version of the object and decrements the number of active readers for the object, thereby allowing writers a chance to proceed.

In replay mode, the entry procedure for readers proceeds as before, except that history tapes are not written, they are merely read and advanced as execution proceeds. Each reader process must wait until the version number for the target object is equal to the version number recorded on the reader's history tape. This ensures that the reader will see the correct version of the target object during replay. Once the reader has read the object, a count of readers for that version is incremented in the exit routine. This counter allows a writer to create the next version of an object only when all readers have finished with the current version.

Each process that modifies a shared object must use the entry procedure **WriterEntry** (Fig. 2). In this routine, the writer uses a semaphore associated with the object to gain exclusive access to the object. Once the semaphore is acquired, the writer process waits for all active readers to finish. No new readers can access the object since the entry routine for a reader must also acquire the semaphore. When all readers have finished with the object, the writer is free to access the current version of the object. The writer records the current version number of the object onto its process history tape as well as the number of readers for that version. The writer may then modify the shared object. Exclusive access is maintained because the semaphore is not released until the exit procedure is called. The **WriterExit** routine (also in Fig. 2) simply initializes the number of readers for the new version, increments the version number for the object, and releases exclusive access to the object by performing a *V* operation on the object's semaphore.

In replay mode, the object semaphore is not required for either readers or writers because the information on process history tapes, in conjunction with the counts maintained with the object, is sufficient to correctly order the operations on a target object. A writer must wait until the current version of the object matches the version number recorded on the writer's history tape. This ensures that the writer modifies the correct version. Next, the writer must make sure that the number of readers that have seen the current version of the object during replay is equal to the number of readers that saw that version

[5] We use atomic increment and decrement operations to maintain the reader counts for an object, thereby avoiding the need for additional synchronization.

in the original execution. Since the **ReaderExit** routine updates the count of total readers for the object version after completing the read, a writer cannot proceed until all reads of the previous version have finished. Following the write operation, the **WriterExit** procedure simply initializes the number of readers for the new version and then increments the object version number. Since this is the last operation performed by a writer, no reader will attempt to access the new version until the writer has finished.

This description of a CREW access protocol is intended to be illustrative, not definitive. Instant Replay requires neither a CREW protocol nor this particular implementation of a CREW protocol. As stated previously, we could use an ME protocol to guarantee a valid serialization. A different implementation would probably be required in a loosely coupled system, one that does not use shared memory. In particular, rather than accessing shared memory locations to read and record object status information, some parts of the protocol could be implemented as remote operations. Version numbers could be used to control access to message buffers on remote nodes, preventing buffer overflow problems during replay. Also, additional machinery (e.g., buffers) would need to be added so that the communication necessary for replay does not compete for the same limited resources used by the executing program. Nevertheless, regardless of the characteristics of a particular implementation of the access protocols, our basic approach is to record a partial order of operations on each shared object and ensure the same order during program replay.

III. Multiprocessor Prototype of Instant Replay

A prototype implementation of Instant Replay has been developed for the BBN Butterfly™ Parallel Processor. Several considerations motivated the choice of the Butterfly as a testbed. First, we have a Butterfly at the University of Rochester, but lack methods and tools for debugging parallel programs. This, combined with the current surge of software development for the Butterfly, created an urgent need we wanted to fulfill. Second, interprocess communication on the Butterfly is inexpensive, which tends to encourage development of communication-intensive programs. Third, communication on the Butterfly is available over a wide range of granularities; process interactions can occur through direct sharing of memory, or through the use of higher level primitives for message passing. Finally, the high degree of parallelism offered by the Butterfly provides a challenging test since highly parallel, communication-intensive applications will experience the greatest performance degradation using any program monitoring technique.

A. The BBN Butterfly Parallel Processor

The BBN Butterfly Parallel Processor at the University of Rochester consists of 128 processing nodes connected by a switching network. Each switch node in the switching network is a 4-input, 4-output crossbar switch with a bandwidth of 32 Mbits/s. Each processor is an 8 MHz MC68000 with 24-bit virtual addresses. A 2901-based bit-slice coprocessor interprets every memory reference issued by the 68000 and is used to communicate with other nodes across the switching network. All the memory in the system resides on individual nodes, but any processor can address any memory through the switch. A remote memory reference (read) takes about 4 μs, roughly five times as long as a local reference.

Chrysalis [19], the Butterfly operating system, consists largely of a protected subroutine library that implements operations on a set of primitive data types, including event blocks (structures used by processes to post a word of data to the event owner), dual queues (queues that hold a sequence of long word data enqueued by processes, or alternatively, a sequence of process handles corresponding to processes waiting to dequeue data as it becomes available), shared memory segments, and a global name table. Objects of these types can be shared among all processes executing on the machine. Low-level operations on these data types are provided by Chrysalis, many of which are implemented by microcode. These primitive operations provide a general framework upon which efficient high-level communication protocols and software systems can be built.

B. Monitoring Chrysalis Operations

Our prototype implementation provides programmers with encapsulated versions of the Chrysalis primitive operations on events, dual queues, shared memory objects, and processes. The encapsulated versions of the Chrysalis primitives enforce CREW access synchronization and record a partial order on the operations as detailed in the previous section. This implementation was done at the level of primitive Chrysalis operations to make replay available to all programs; it can be used in any software system developed on top of the Chrysalis operating system. In particular, recent system development efforts at the University of Rochester that can be easily modified to incorporate Instant Replay include LYNX, a programming language and runtime system for distributed computing [22], [23], and SMP, a message-passing system that supports multicast message communication among groups of processes [15].

While encapsulating the Chrysalis primitives for events and dual queues, it became apparent that providing a CREW protocol for all operations was inappropriate. Most of the operations on events and dual queues are atomic, which means that the operations must occur serially with respect to their target data object (a characteristic of the hardware). The CREW protocol allows concurrent readers of shared objects, but introduces additional cost. Since event and dual queue operations cannot exploit concurrent execution of readers, the expense of the CREW protocol is not justified. By replacing the CREW protocol with the simpler mutual exclusion (ME) protocol, we force the serial execution of the Chrysalis event and dual queue primitive operations, but reduce execution overhead by simplifying the entry and exit protocols. An ME protocol enables use of a single entry/exit routine pair and reduces the amount of information recorded on process history tapes, since we need not maintain a count of the readers for each version.

Using encapsulated versions of Chrysalis primitives in program code requires no additional effort beyond that necessary to use the original primitives. Additional program code is only necessary for regulating access to shared memory objects. Chrysalis provides primitives for sharing segments of memory. General sharing of memory objects as provided by the Butterfly hardware and Chrysalis primitive operations imposes no restrictions on memory access other than serializing word operations on each node, since the memory hardware has only a single port. To guarantee that operations on such shared segments conform to a CREW access protocol, it is necessary to use access entry and exit routines to control sharing of these segments. The programmer can control the granularity of operations bracketed by the access routines in response to performance concerns. By controlling the cost of the operations within an entry and exit routine pair, the programmer can balance the reduction of parallelism incurred when locking for long periods of time with the overhead of frequently executing the locking primitives. (Since the access protocol entry and exit routines have a small critical section requiring mutual exclusion, there is a serial nature to their execution.)

C. Case Studies

Two applications were chosen for experiments in program monitoring and replay: computation of a knight's tour of a chess board and Gaussian elimination. The knight's tour problem was chosen because there is an existing implementation on the Butterfly that exhibits extremely nondeterministic behavior. A parallel implementation of Gaussian elimination was chosen for study since, unlike the knight's tour program, no matter what execution path occurs when the Gaussian elimination program is run, the overall amount of computation performed by the program is constant. Also, our implementation of Gaussian elimination has already been the subject of a thorough performance study [14] and the statistics previously obtained about the program's execution behavior can be used as a baseline for comparison to determine the cost of our monitoring techniques.

1) Knight's Tour: A knight's tour is a path on a chess board for a knight that successively visits each square once and only once using legal chess moves. Our program to compute a knight's tour of a chess board consists of a master process and a user-specified number of slave processes. The master selects an initial position of the knight on the chess board and enters the corresponding board description in a global task queue. Next, the master creates a set of slave processes that cooperate to search for a knight's tour beginning with the initial board position. Each slave removes a set of board descriptions from the global task queue and replaces it with a new set of board descriptions which could be generated by adding a legal move of the knight from its previous position. The order in which these board descriptions are added and deleted from the task queue determines the breadth and depth of the search performed. Since the order in which slave processes are granted access to the task queue depends on the relative progress of the processes and resolution of memory contention for the task queue, successive executions of the program tend to produce different tours.

Calls to monitored versions of the task queue primitive operations (the task queue is a dual queue) were inserted in the program in place of the original calls to Chrysalis primitives. These modifications did not require substantial effort and caused no significant growth in code size. The effect on the performance of each individual primitive is substantial, since the original primitives are implemented in microcode and there is no such support for the history tape maintenance operations. However, the effect on overall program performance is difficult to measure due to the inherent nondeterministic nature of the knight's tour computation. We cannot obtain identical executions of the monitored and unmonitored versions of the program to compare execution times because such times vary wildly between successive invocations of the program. We were able to measure accurately the comparative execution times for a knight's tour program during the monitoring phase and the replay phase of the same execution. The difference between the two execution times was less than 5 percent.

Using 16 processors, three successive executions required 18, 38, and 52 s to find three different solutions for a 5×5 chess board; the executions used 12K, 36K, and 60K bytes, respectively, for history tapes.[6] Using 64 processors, a solution was found in 43 s and required 48K bytes for history tapes. It is not surprising that the amount of space required for the history tapes of the knight's tour program varies with the amount of time taken to find a solution. Communication is roughly a constant percentage of the computation and no matter how many processors are working on the task, communication speed, hence history tape space requirements, is limited by the need to serialize access to a single shared task queue. We estimate that the knight's tour program generates between 250 and 300 communication events per second; each communication event requires four bytes to record. From this we can estimate the space requirements for the history tape as a function of the time needed to find a particular solution.

2) Gaussian Elimination: To obtain an empirical comparison of the relative cost of monitored and unmonitored program executions, an existing program to solve a system of linear equations using Gaussian elimination was instrumented. In Gaussian elimination, the total amount of work performed by the program is independent of the precise ordering of interprocess events during execution; the computation for each pivot row depends on a fixed number of other rows.

The implementation of Gaussian elimination uses a broadcast message-passing system as the basis for communication among the cooperating processes in the program.[7] A single master process initializes shared data structures and then spawns worker processes to diagonalize the matrix. The master delegates rows of the matrix to each slave process participating in the solution. Each time the processing of a row is completed, the contents are broadcast by the process holding that row to each of the other slaves.

[6] Our current implementation uses a 32-bit word for each entry on a history tape, although 16-bit words would suffice for our case studies, as well as most other programs. Therefore, our space requirements are conservative and could easily be reduced by a factor of 2.

[7] The message-passing system used here is an early prototype of SMP [15]. The results described in this section are particularly relevant to programs based on SMP, or similar communication models.

```
Send Message
  Find buffer
  WriterEntry(buffer, myprocess)
  Copy message into buffer
  Set number of recipients
  WriterExit(buffer)

Receive Message
  ReaderEntryPoll(buffers, myprocess)
  Poll incoming message buffers
  Copy message into user area
  ReaderExitPoll(buffers)
  WriterEntry(buffer, myprocess)
  Decrement number of recipients
  WriterExit(buffer)
```

Fig. 3.

To instrument this application we replaced some dual queue and event primitives used for synchronization between the master and slaves with monitored versions of the Chrysalis primitives. The underlying message-passing system, however, required more extensive changes. Message passing was implemented using shared memory segments as communication buffers. Modifications to the send and receive primitives of the message-passing system were required to enforce the CREW access protocols, as detailed in Section II, for the shared communication buffers.

Although the code overhead and programming effort to make this transformation were more substantial than that required for the knight's tour, the size of the effort was still small. The original Gaussian elimination program contains 1059 lines of code. To instrument the program for Instant Replay, 24 lines of code were altered and 17 lines of code were added. Most of the changes to the source code files occurred in the message-passing module. Fig. 3 shows the skeletal form of the monitored message-passing routines.

The performance of the Gaussian elimination implementation was degraded by the enforcement of a CREW protocol on shared object access and recording the access order to shared objects. Fig. 4 depicts the performance of monitored and unmonitored versions of the application on a 400 × 400 matrix. The unmonitored program improves dramatically in performance as additional processors become involved in the computation, however, there is no significant improvement in performance when more than 32 processors are in use. In fact, performance begins to degrade slightly beyond 32 processors because the additional communication involved is not justified by the gain in parallelism [14]. Our first attempt at monitoring this program did not incorporate any optimizations and resulted in severe performance degradation when more than 8 processors were in use, as shown in Fig. 4. This experiment demonstrates the importance of efficient monitoring operations. Modifying the monitoring protocols to reduce the size of critical sections greatly improved the performance, but still managed to roughly triple the execution time of the program on 64 processors. Examination of the monitoring cost showed that the program was spending a great deal of time monitoring and recording noncritical polling operations on buffers.[8] To

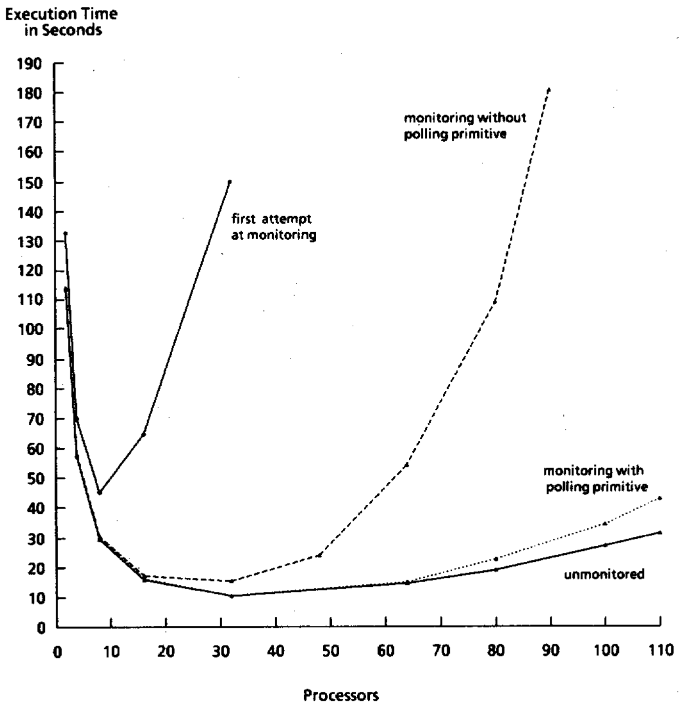

Fig. 4. Gaussian elimination of a 400 × 400 matrix using message passing.

lower the cost of monitoring, we devised a special entry procedure for use with the common programming idiom in which readers poll before reading a value.

Our implementation of message passing uses polling to find incoming messages. Whenever a process attempts to receive a message, a large number of buffers, one for each process in the computation, are polled. Our naive approach to monitoring operations considered each polling operation as an access to a shared object, which was duly recorded on the process's history tape. The realization that none of the polling operations, *except the last one,* are necessary for replay led us to devise a special entry procedure used in conjunction with polling. With this new entry procedure, the access ordering to a buffer is recorded only when a message is found. An indication of which buffer supplied the message and the version number for that buffer are recorded on the process's history tape. During replay, only the buffer from which a process received a message during the monitoring phase is polled. Use of this entry procedure eliminated recording of nonessential ordering information during the monitoring phase, saving both time and storage space for the information collected. The performance of the program using the special entry procedure is also shown in Fig. 4. The result: we were able to monitor a communication-intensive application for replay by imposing a performance overhead of less than 1 percent for up to 64 processors. In addition, we were able to replay the program in the same amount of time as was used by the original execution.

As we have already stated, Gaussian elimination is a communication-intensive program, which tends to produce large history tapes. Diagonalization of an 800 × 800 matrix on 64 processors requires 400K bytes for the history tapes. While this is not a small amount of space, it is worth

[8] New evidence has cast doubt upon the data used to plot the curve for monitoring without polling primitives. While monitoring with polling primitives is clearly preferable, we now believe the disparity between these two approaches is less severe than our graph suggests.

comparing the space requirements for our method with other techniques that save the contents of every message received by a process in an event log. Such an approach requires over 150 Mbytes of space! In general, Instant Replay will always take less space than an event log whenever large messages are involved, since we only require between 4 and 8 bytes for each message.[9]

IV. Instant Replay in the Debugging Cycle

Program replay makes it possible to repeat the execution of a parallel program as often as desired. Unfortunately, Instant Replay does not automatically debug programs, parallel or otherwise. How then do we use the replay capability to debug parallel programs? In this section we describe several techniques for error isolation that can be used together with our approach. We have already used some of these techniques in our own work; others require the cooperation of additional tools that we have not yet developed.

Our goal is to provide repeatable execution, so that it is possible to observe the same execution of a parallel program as often as desired. Any results that may have been ignored during previous observations can always be reproduced on demand for closer examination. This capability is especially useful for parallel programs since a) multiple processes tend to generate a lot of output, making it easy to miss important results and b) the programming environments for parallel architectures are not as mature as the programming environments for sequential machines, and often lack tools for collecting and analyzing output data. However, the most important reason for reproducible behavior is that it makes cyclic debugging possible.

The simplest form of cyclic debugging is to add output statements to an erroneous program that provide additional details about the execution of the program. Successive executions can be used to provide successively greater detail about those parts of the program under suspicion. This technique does not work with parallel programs in general because the output statements can change the relative timing of operations within the program and yield a different execution sequence. With Instant Replay, however, any number of output statements can be added to the program without changing the execution sequence provided by the replay mechanism. In fact, *any type of statement may be added to the program during replay,* as long as the additions do not affect the sequence of interactions with shared objects by each process. Thus, the programmer can debug parallel programs by adopting the same cyclic methodology for error isolation used in debugging sequential programs. We have found that this capability alone is a valuable tool for debugging parallel programs, particularly in the absence of other debugging tools.

Repeatable execution also makes top-down, interactive debugging possible. Hierarchical abstraction of detail is necessary to cope with the complexity of large software systems. Abstraction is particularly important in understanding the behavior of parallel programs. The programmer should not have to be concerned with the low-level details of execution of a parallel program, such as the interleaving of primitive operations. Instead, we are interested in the salient features of the execution that characterize its behavior. Our approach allows the programmer to start with a high-level view of a program's behavior, produced by normal output statements or an event mechanism similar to Behavioral Abstraction. By carefully refining that viewpoint, based on the information made available during successive replay, the programmer can study erroneous behavior at any level of detail desired. As a result, one can diagnose program errors in a top-down fashion without wading through voluminous, irrelevant detail at each step.

Another common technique used to debug sequential programs is breakpoint insertion. Breakpoints are added to the program at interesting points in the code. Execution is suspended at each breakpoint, allowing the programmer to examine the system state. Breakpoints only suspend a single thread of execution, however, which is not sufficient for parallel programs consisting of multiple threads of execution. Inserting a breakpoint in one process of a parallel program will have an effect on every process that communicates, directly or indirectly, with the suspended process. In particular, breakpoints can change the relative order of events during execution, producing a different execution sequence each time. Fortunately, we can provide reproducible execution *even in the presence of breakpoints.* No matter how many breakpoints are encountered during replay, we continue to order operations based on the contents of history tapes. A process that is suspended by a breakpoint will eventually cause all other processes to wait for some shared object to be read or written (assuming a connected graph of process interactions). When the suspended process is allowed to continue beyond the breakpoint, it will eventually catch up to the other processes and the entire program will continue executing. Thus, it is possible to cycle through breakpoints in many different processes during program replay, examining system state for a different process at each breakpoint.

This use of breakpoints also allows the programmer to examine the global state of the computation. Due to communication delays and a reliance on local viewpoints, it is impossible to take an instantaneous snapshot of global state. However, all we really need to see are *meaningful global states* [4], consistent states based on the *happened before* ordering of Lamport [11]. For example, if we suspend a process P at breakpoint X, all events that occurred before P reached X should be reflected eventually in all other processes. In addition, other processes should not be allowed to proceed beyond any point that requires process P to proceed beyond X. This view of a computation is the best we can hope for since, if all processes are stopped as the result of setting a single breakpoint, the *happened before* relation cannot distinguish between the global state represented by all suspended processes and an omniscient snapshot of the global state during normal execution. We provide exactly this notion

[9] Normally, four bytes per message are used, however, the polling entry procedure used by Gaussian elimination requires eight bytes.

of global state, and any notion that attempts to be more precise is not likely to be meaningful in a distributed system.

We can use breakpoints, in conjunction with Instant Replay, to provide the ability to halt distributed programs in a consistent state, as in [18], without the need for additional mechanisms. By setting a local breakpoint during replay we are, in effect, setting a breakpoint in the global state. When the local breakpoint is reached, we can see the exact state of the local process containing the breakpoint, and the exact state of all other processes as they block due to enforcement of the *happened before* relation. Differences between the state of each process in an instantaneous snapshot and what we see at a breakpoint reflect the natural degree of asynchrony between processes in the program.

A consequence of our breakpoint capability is the ability to support single-step execution of processes. Single-step execution can be used during debugging to trace the state transitions of an individual process or the effects of interprocess communication on the internal states of communication partners. We can replay a process using single-step execution because enforcement of the *happened before* relation ensures that asynchrony between processes remains within allowable bounds.

Instant Replay can also be used in conjunction with an event log technique to allow repeatable execution of a subset of processes involved in a computation. As we have described it, our approach requires that the input to each process be recomputed during replay, rather than retrieved from an event log. This is both an advantage and a disadvantage. While our technique requires less time and space during the monitoring phase, it also requires that all processes be reexecuted during replay. Global replay is a disadvantage if the computational requirements to replay a program are very large, particularly when it is unnecessary to recreate the entire original set of processes to isolate an error. By using an event log together with Instant Replay, we can reexecute the subset of processes in which we are interested and simulate the rest.

There is a tradeoff between the expense of maintaining an event log during normal execution and the expense of reexecuting all processes during replay. The event log approach and Instant Replay represent two extremes, wherein the expense is shifted from the monitoring phase to the replay phase. However, a compromise between our technique and the event log approach is possible. When frequent replay of a subset of processes in a computation is desired, as would be the case when using cyclic debugging to isolate errors, it is possible to collect additional information in an event log during replay that would eliminate the need for reexecution of the entire program during subsequent replay. We can record in an event log all external inputs to the subset of processes of interest. This record would include both inputs from the external environment and inputs from processes not under scrutiny. Interactions involving processes to be reexecuted during replay are recorded, as before, as partial orders on history tapes. On subsequent executions, only the designated subset of processes would be reexecuted and their interface with the external environment, including the other processes, would be simulated using the event log. Since we assume that the debugging methodology is cyclic, the set of processes that are simulated by an event log will grow larger as we look at fewer processes in greater detail (i.e., top-down debugging). Note however that we would continue to use Instant Replay in the monitoring phase because it has the least impact on normal program execution and can be used to generate event logs during the debugging cycle.[10]

Finally, we can use Instant Replay, together with techniques developed by Miller [16], [17], for both causal analysis and performance monitoring of parallel programs. These techniques use a program history graph, which represents interprocess events and the elapsed time between related events, to analyze the behavior of the program. It is possible to change some aspects of the history graph to analyze the effect of changes in the execution environment [17], however, there is no guarantee that modifying system parameters, such as expected communication delay and processor load, will not change the execution behavior of the program. By using Instant Replay to guarantee repeatable execution behavior, it is possible to change cost labels in the history graph and replay the program under new assumptions, *without changing the execution behavior of the program*. (Of course, the replay mechanism would have to be modified to incorporate changes to the history graph, such as the message delay time.) In particular, one could examine the effect of communication costs on overall program performance by artificially varying the delay associated with communication. It is important to note that performance results derived from such an exercise are estimates, since the program is forced to obey a particular execution sequence in the presence of varying performance parameters. However, it is still possible to learn a great deal about parallel programs using these techniques, particularly when used with programs whose executions are less sensitive to race conditions.

V. Conclusions and Future Work

One of the most important tools for analyzing and debugging software is the interactive debugger. Cyclic debugging with an interactive debugger requires the ability to reproduce program behavior on demand. We have described the design and implementation of a system for reproducible execution of parallel programs. In summary, Instant Replay:

- provides reproducible execution of parallel programs
- is not dependent on any particular form of interprocess communication
- makes possible global replay of programs, rather than processes
- introduces no centralized bottleneck, either during monitoring or replay
- does not require synchronized clocks or globally consistent logical time

[10] In extraordinary circumstances where even a single replay is impractical, process history tapes and a partial event log could both be recorded during the monitoring phase.

- allows modifications to programs during the debugging cycle
- has only minor impact on program performance during the monitoring phase
- has reasonable space requirements
- is applicable to both loosely coupled and tightly coupled environments.

There are two potential disadvantages to our approach. First, we record a version number for each access to a shared object. If the granularity of communication is very small (e.g., one-byte messages), we could use less space by simply storing data values (i.e., the event log approach). Second, we require that all processes in a program reexecute during replay. Even though we have shown how to use event logs to eliminate some processes during successive replays, no iterative technique is well-suited to programs that are impractical to reexecute.

Nevertheless, our experience has shown that Instant Replay is effective, efficient, and practical. Additional experience with our technique is necessary, however. We must perform further empirical studies to determine the performance cost of our monitoring technique on other programming environments. Specifically, we intend to explore applications of our techniques to message-based communication in loosely coupled systems and lightweight tasks and shared memory in tightly coupled systems. Our case studies, while very different in programming style, do not address all of the programming models we wish to support.

Several optimizations to reduce further the time and space needs of our technique are also under consideration. An example of such an optimization was described in Section III. Other optimizations based on similar idempotent operations are possible. Another interesting optimization is based on the observation that some parallel programs (or segments of programs) *are deterministic.* The Gaussian elimination program is a good example. The processes that perform Gaussian elimination proceed in lockstep; no monitoring operations are necessary to reproduce behavior. It is possible to reduce contention and space needs for monitoring if we can determine that some sequence of interprocess operations yields a deterministic schedule. Clearly this information is application-specific and may only be obtainable with programmer assistance. Nonetheless, this approach is worth exploring for large parallel systems with deterministic components.

Finally, we intend to explore the impact of Instant Replay on the development of a general-purpose programming environment for parallel architectures. Additional tools will be constructed as a part of any such environment (e.g., source-level single-process debuggers for parallel programs, tools to monitor execution with graphical displays, compilers to automatically instrument programs), and we will want to integrate our program replay capability with those tools as they are developed.

Acknowledgment

We are grateful to our colleagues in the Department of Computer Science, University of Rochester for their many comments and criticisms. In particular, N. Gafter made numerous suggestions that substantially improved the presentation.

References

[1] F. Baiardi, N. DeFrancesco, and G. Vaglini, "Development of a debugger for a concurrent language," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 547-553, Apr. 1986.
[2] P. Bates and J. Wileden, "High-level debugging of distributed systems: The behavioral abstraction approach," Dep. Comput. and Inform. Sci., Univ. Massachusetts, COINS 83-29, 1983.
[3] R. Carver and K. Tai, "Reproducible testing of concurrent programs based on shared variables," in *Proc. 6th Int. Conf. Distrib. Comput. Syst.*, Cambridge, MA, May 1986, pp. 428-433.
[4] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, pp. 63-75, Feb. 1985.
[5] S. Y. Chiu, "Debugging distributed computations in a nested atomic action system," Dep. EECS, Massachusetts Inst. Technol., Cambridge, MIT/LCS/Tech. Rep.-327, Dec. 1984.
[6] P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent control with readers and writers," *Commun. ACM*, vol. 14, pp. 667-668, Oct. 1971.
[7] R. S. Curtis and L. D. Wittie, "BugNet: A debugging system for parallel programming environments," in *Proc. 3rd Int. Conf. Distrib. Comput. Syst.*, Miami, FL, pp. 394-399, Oct. 1982.
[8] E. W. Dijkstra, "The structure of the 'THE' multiprogramming system," *Commun. ACM*, vol. 11, pp. 341-346, May 1968.
[9] H. Garcia-Molina, F. Germano, and W. H. Kohler, "Debugging a distributed computing system," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 210-219, Mar. 1984.
[10] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Commun. ACM*, vol. 17, pp. 549-556, Oct. 1974.
[11] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558-565, July 1978.
[12] ——, "On interprocess communication," Digital Equipment Corporation's Western Research Lab., Tech. Rep., Dec. 1985.
[13] R. J. LeBlanc and A. D. Robbins, "Event driven monitoring of distributed programs," *Proc. 5th Int. Conf. Distrib. Comput. Syst.*, Denver, CO, May 1985, pp. 515-522.
[14] T. J. LeBlanc, "Shared memory versus message-passing in a tightly-coupled multiprocessor: A case study," in *Proc. Int. Conf. Parallel Processing*, St. Charles, IL, Aug. 1986, pp. 463-466.
[15] T. J. LeBlanc, N. M. Gafter, and T. Ohkami, "SMP: A message-based programming environment for the BBN Butterfly," Dep. Comput. Sci., Univ. Rochester, Rochester, NY, Butterfly Proj. Rep. 8, July 1986.
[16] B. P. Miller, "Performance characterization of distributed programs," Ph.D. dissertation, Computer Science Division (EECS), Univ. California, Berkeley, Tech. Rep. UCB/Computer Science Dep. 85/197, Jan. 1985.
[17] B. P. Miller, "Parallelism in distributed programs: Measurement and prediction," Dep. Comput. Sci., Univ. Wisconsin, Madison, Tech. Rep., May 1985.
[18] B. P. Miller and J. D. Choi, "Breakpoints and halting in distributed programs," Dep. Comput. Sci., Univ. Wisconsin, Madison, Tech. Rep. 648, July 1986.
[19] W. Milliken *et al.*, Chrysalis Programmer's Manual, Version 2.2, BBN Lab., June 1985.
[20] G. L. Peterson, "Concurrent reading while writing," *ACM Trans. Programming Lang. Syst.*, vol. 5, pp. 46-55, Jan. 1983.
[21] R. D. Schiffenbauer, "Interactive debugging in a distributed computational environment," masters thesis, Computer Science Division, Dep. EECS, Massachusetts Inst. Technol., Cambridge, MIT/LCS/Tech. Rep. 264, Sept. 1981.
[22] M. L. Scott, LYNX Reference Manual, Butterfly Proj. Rep. 7, Dep. Comput. Sci. Univ. Rochester, Rochester, NY, Mar. 1986.
[23] M. L. Scott, "The interface between distributed operating system and high-level programming language," *Proc. Int. Conf. Parallel Processing*, St. Charles, IL, Aug. 1986, pp. 242-249.
[24] E. T. Smith, "Debugging tools for message-based, communicating processes," in *Proc. 4th Int. Conf. Distrib. Comput. Syst.*, San Francisco, CA, May 1984, pp. 303-310.
[25] P. Vitanyi and B. Awerbuch, "Atomic shared register access by asynchronous hardware," in *Proc. 27th Ann. Symp. Found. Comput. Sci.*, Toronto, Ont., Oct. 1986, pp. 233-243.

Thomas J. LeBlanc received the B.S. degree in computer science from the State University of New York in 1977 and the M.S. and Ph.D. degrees in computer science from the University of Wisconsin, Madison, in 1979 and 1982, respectively.

Since 1983 he has been an Assistant Professor in the Department of Computer Science, University of Rochester, Rochester NY, where his research focuses on software support for parallel and distributed systems, including programming languages, operating systems, and program debugging. He is now exploring these issues in the BBN Butterfly environment, a tightly coupled multiprocessor consisting of 128 MC68000's. His paper (with S. A. Friedberg) was the recipient of the Best Paper Award at the 5th International Distributed Computing Systems Conference. He also received the Distinguished Presentation Award at the 1986 International Conference on Parallel Processing.

Dr. LeBlanc is a member of IEEE Computer Society and the Association for Computing Machinery.

John M. Mellor-Crummey received the B.S.E. degree in electrical engineering and computer science from Princeton University, Princeton, NJ, in 1984 and the M.S. degree in computer science from the University of Rochester, Rochester, NY, in 1986.

From 1984 to 1986 he was a Sproull Fellow in the Department of Computer Science, University of Rochester, where he is currently working towards the Ph.D. degree in computer science. His research interests include parallel processing, distributed systems, VLSI, programming languages, and compiler design.