SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 22(10). 863–877 (OCTOBER 1992)

Monitoring and Debugging Distributed Real-time Programs

PAUL S. DODD AND CHINYA V. RAVISHANKAR
Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, Michigan, 48109-2122, U.S.A.

SUMMARY

In this paper we describe the design and implementation of an integrated monitoring and debugging system for a distributed real-time computer system. The monitor provides continuous, transparent monitoring capabilities throughout a real-time system's lifecycle with bounded, minimal, predictable interference by using software support. The monitor is flexible enough to observe both high-level events that are operating system- and application-specific, as well as low-level events such as shared variable references. We present a novel approach to monitoring shared variable references that provides transparent monitoring with low overhead. The monitor is designed to support tasks such as debugging real-time applications, aiding real-time task scheduling, and measuring system performance. Since debugging distributed real-time applications is particularly difficult, we describe how the monitor can be used to debug distributed and parallel applications by deterministic execution replay.

KEY WORDS Program monitoring Debugging Real-time systems

INTRODUCTION

Distributed real-time systems are a complex environment from the perspectives of software design, development, testing, and operation. Monitoring distributed real-time systems is a difficult challenge that must be met if software designers, system architects, and performance evaluators are to measure, debug, test, and develop systems efficiently. A key requirement for real-time system monitors is low overhead, but even more important is predictable overhead.

Monitoring can be defined as the measurement, collection, and processing of information about the execution of tasks in a computer system. System characteristics may complicate this task. A real-time system requires the monitor itself to operate under strict reliability and performance constraints. The reliability constraints require that the monitored system and the monitor continue to operate in the presence of static or dynamic failures. The performance constraints require that the interference caused to the system by the monitor's presence must be predictable, minimal, and bounded. In particular, the monitor must not introduce or hide timing errors. [1]

Distribution also imposes constraints on the monitor. Distributed systems lack both global state information and a sense of global time. There is no total ordering defined over events that occur on different nodes. Monitored data must be collected from several sites and integrated to obtain a coherent view of the system. Further,

0038–0644/92/100863–15$12.50
© 1992 by John Wiley & Sons, Ltd.
Received 9 December 1991
Revised 11 June 1992

when tasks run in parallel, their behavior can be nondeterministic. The monitor must support the deterministic replay of applications for effective debugging.

This paper introduces HMON, a monitor for an experimental distributed real-time system called the Hexagonal Architecture for Real-Time Systems (HARTS) being developed in the Real-Time Computing Laboratory at the University of Michigan. Our goal is to provide a real-time monitor integrated with its environment to support tasks such as debugging distributed real-time applications, aiding real-time task scheduling, and measuring performance. We perform the monitoring transparently, so the programmer does not need to add special code to applications. The monitor provides continuous monitoring capabilities throughout a real-time system's lifecycle, from design and testing to production. The monitor supports the deterministic replay of distributed real-time applications to aid debugging. Our monitor is flexible enough to observe both high-level events that are operating system- and application-specific, and low-level events like shared variable references. Our method of monitoring shared variables is novel and transparent. The monitor uses software integrated into the system call libraries for flexible, transparent monitoring. We also dedicate some system hardware to the monitor to minimize interference with the measured system, but no special hardware is required. Our system is intended for general-purpose real-time multiprocessors. Although our techniques were developed for the HARTS environment, they may be applied to other real-time systems. Our approach is unique because we perform transparent monitoring and deterministic replay on a distributed real-time system without adding any special hardware such as a bus probe or a hardware instruction counter.

Monitoring and debugging are topics of active research. Software monitors [2-7] are popular because they allow users to view the monitored system at various levels of complexity or abstraction. These monitors are also independent of low-level hardware details, such as data bus width or memory cycle time, and are flexible because they are easily modified, not being etched in silicon. They can also be integrated into the operating system and programming environments. However, typical monitors are invasive and not applicable to real-time systems because of their unpredictable interference. They typically run on the same CPU as the monitored tasks, so their interference effects can be considerable. *Ad hoc* solutions such as turning off the real-time clock or altering timeout values during monitoring operations, as suggested in References 8 and 9, will not work when the system interacts with the real world or uses asynchronous interrupts.

Simulators have been used as flexible software debugging tools. [10] However, it is difficult to simulate a complex distributed real-time system, or validate the correctness of simulations. Simulated execution is also much slower than execution replay.

Passive hardware monitors [11-14] can provide detailed, low-level information about a system, such as communication activities, memory accesses, and I/O patterns with little interference to the monitored system. However, hardware monitors do not support the interactive modification of task execution that is necessary to support debugging. It is also difficult to use hardware monitors in computer systems which include such complexities as on-chip cache memories, coprocessor, and parallel or distributed processors.

A number of monitoring and debugging systems have been developed specifically for real-time systems. [15–19] In addition, some techniques developed for debugging

distributed systems may safely be applied to real-time systems during replay since interference during replay does not alter program execution. However, interactive debugging of real-time programs without deterministic replay is not sufficient because debugging commands can destroy the timing-dependent nature of real-time systems. Similarly, breakpoints must only be inserted during replay to maintain the consistent timing behavior of the system. Breakpoints inserted during normal execution, as in References 20 and 21, can nondeterministically alter the execution of a distributed real-time program. The halted process may timeout, since the real-time clock continues to run while the process is suspended. Tasks that are not suspended will continue to run and alter the system state. Breakpoints can safely be used during replay because event timing is maintained.

Tsai *et al.* [22] offer a low-interference monitoring-and-replay system for debugging real-time uniprocessors. There are four differences between their technique and ours. First, they do not support parallel or distributed systems. Second, their monitor records considerably more data than ours. For example, a six byte AND Immediate instruction on a MC68000 generates 256 bytes of log data. We do not monitor individual instructions, which saves space but lowers the resolution of our monitor. Third, their dual processor unit causes unpredictable interference on the target system by generating an interrupt for every event monitored. While HMON adds more overhead, it is predictable. Finally, their approach can reproduce asynchronous interrupts only if the CPU has a hardware instruction counter. Without this special hardware, they cannot perform deterministic replay. HMON supports deterministic replay without a hardware instruction counter.

Bugnet [23,24] is an older system that provides interactive replay for debugging distributed programs without using any special hardware. However, it does not reproduce the exact timing of events, and it uses checkpoints, which are too intrusive for real-time systems. HMON records all events that occur on a processor in a single event log and replays events without altering their timing.

Tai *et al* [25] suggest a system-independent language-level approach to deterministic execution replay for concurrent Ada programs. Synchronization events are used to replay the monitored application. However, their system does not support shared variable operations as synchronization events, and it does not consider issues such as real-time clock values or dynamic task creation. HMON supports shared variable operations, the real-time clock, and dynamic task creation, at the expense of losing language-level generality.

THE HARTS SYSTEM

HMON is a monitoring and debugging environment for HARTS, an experimental distributed real-time system used for research in the Real-Time Computing Laboratory at the University of Michigan. A primary feature of HARTS is its hexagonal mesh interconnection network [26] (see Figure 1). This network architecture has several attractive features for general-purpose real-time systems: simple interconnections, efficient message routing, and high fault tolerance. Each node on the network is directly connected to six neighbors using point-to-point serial links. The nodes themselves are shared memory multiprocessors, each consisting of one or more application processors, a network processor, an Ethernet processor, and a system controller. The application processors run the experimental HARTOS distributed

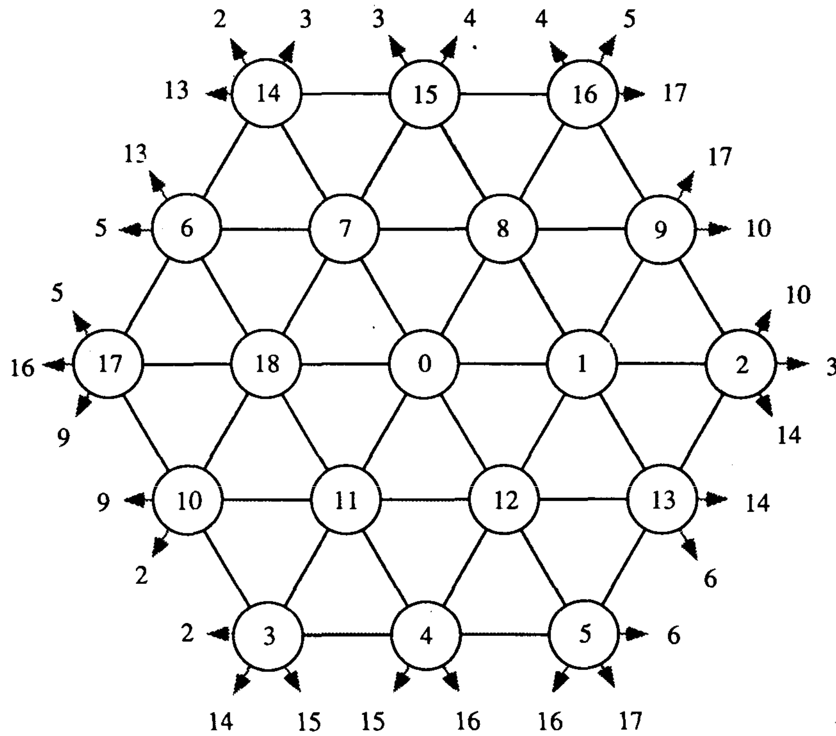

Figure 1. Hexagonal mesh network

operating system kernel, [27] which we are developing on top of the pSOS [28] uniprocessor real-time kernel. The network processor handles all internode and intranode communication to reduce operating system overhead on the application processors. Synchronous signals and messages can be used to communicate between tasks on the same application processor, on different application processors, or on different nodes.

The HARTS system provides a good testbed for monitoring and debugging since it supports applications that are parallel, distributed, or uniprocessor. HARTS also provides a testbed for monitoring and debugging applications with real-time characteristics.

MONITOR ARCHITECTURE

The HMON monitor is a distributed software monitor that runs on a dedicated application processor, called the *monitor' processor* (MP), on each node of HARTS (see Figure 2). Additional code to collect data runs on the network processor and the application processors of each node (see Figure 3). Each processor's local memory is accessible to other processors. The monitor processor logs the data on an external user workstation. Though the data collection code interferes with the system being monitored, in our system this interference is low, predictable, and accounted for in CPU and network scheduling. Since the monitoring code is always running, the interference is the same during normal execution as during development. Further,

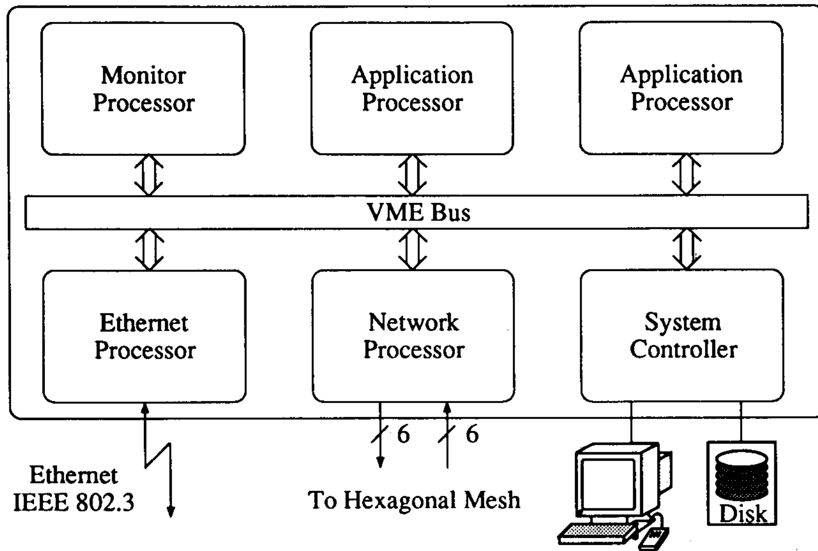

Figure 2. Structure of a HARTS node

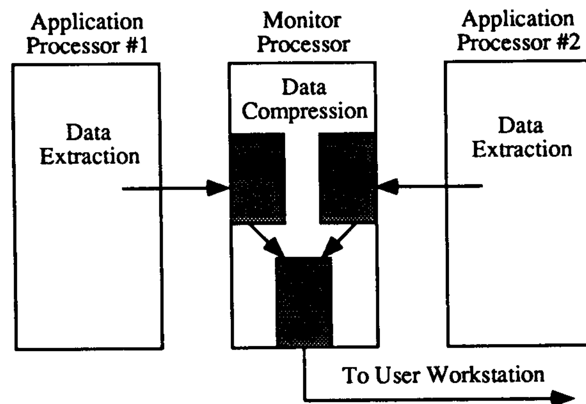

Figure 3. Monitor data collection

the interference does not change during replay. Since this deterministic interference can be measured, the monitoring code is a predictable part of the application.

The monitoring can be divided into three phases: data extraction on the application processors and network processor, data compression on the monitoring processor, and data logging on an external workstation. Data on monitored events is acquired through code inserted into the monitored system. We acquire much of our data by monitoring system calls and context switches transparently. HMON monitors interrupts and shared variable references to allow deterministic replay of tasks for debugging. HMON also provides monitoring calls which programmers can use to monitor any activity not monitored by default. The overhead of this monitoring is

low because the data records are only 16 bytes long, on average, and because monitored events are relatively infrequent. All extracted data is sent to the monitor processor, which orders and compresses it before sending it to the user workstation for logging.

The classes of events monitored are: pSOS system calls, HARTOS system calls, context switches, interrupts, shared variable references, and application-specific events. We now describe how we monitor each type of event.

Monitoring system calls

We monitor all pSOS and HARTOS system calls by modifying the existing system call libraries to include monitoring code. No kernel changes are necessary because only the library routines that marshal the arguments for C programs are modified. HMON monitors interprocess communication system calls, such as message and signal processing, to order distributed events in the manner described in Reference 29. Process management calls, such as creating and deleting processes, are also monitored to follow process interactions. Time management calls that set or read the clock are monitored to record real-time properties for debugging. The calling task's process ID and the call parameters are collected by the monitoring code. To collect data such as the message ID, remote communication system calls are also monitored by the network processor.

Monitoring context switches

We monitor all context switch events through a hook provided by the pSOS kernel. The process IDs of the tasks being switched in and switched out are logged. Task scheduling and CPU usage are determined by studying the order and timing of these events.

Monitoring interrupts

Interrupts are asynchronous sources of input data that can affect the execution of a real-time application. Recording the occurrence of an interrupt and the data transferred with it is straightforward. The difficulty is that the 'time' that the interrupt occurred is needed in order to replay it.

We insert monitoring code into interrupt handlers to log where each interrupt occurs within the dynamic execution trace of a process, so that its timing can be reproduced. Reproducibility is guaranteed by recording an *instruction counter* (IC) value for each interrupt. This IC value is a count of the total number of instructions that have been executed by the process. The IC can be maintained by special hardware [30] by counting the machine instructions as they are executed by the processor. Commercial processors do not have hardware instruction counters, so we simulate a counter by using a *software instruction counter* (SIC) to count backward branches, traps, and subroutine calls, as described in Reference 31. The SIC value and program counter value together define a unique state in a task execution. We can reproduce an interrupt by invoking the interrupt handler at the state defined by these values.

Monitoring shared variables

For tightly-coupled multiprocessors, such as the nodes in HARTS, shared variables enable fast, efficient interprocess communication. However, applications that use shared variables are difficult to debug since the order of shared variable operations is often nondeterministic. The key to debugging such applications is to monitor the order of shared variable accesses. We have developed a novel method to monitor these accesses with low overhead which enables the events to be deterministically replayed.

The HARTOS operating system does not currently provide support for shared variables. We therefore provide language library routines that coordinate access to shared variables and perform the monitoring. We chose to implement a mutual exclusion protocol because of its simplicity and low overhead. Concurrent reader protocols could be used instead of mutual exclusion on a more tightly coupled system than HARTS.

Protocol details

Our method for monitoring shared variables is based on the insight that read operations do not need to be replayed in the exact order that they originally occurred. The relative timing of other read operations does not matter, only that of write operations. In fact, a read operation can be replayed at any point between the two write operations that bound it—the last write operation before it and the first write operation after it. This insight enables us to support deterministic replay by only logging write operations.

Each shared variable is associated with a set of *access counters,* one for each process that uses the variable. Each access counter keeps track of the number of times the variable is read or written by the corresponding process. These access counter values are used to reproduce the order of access when the tasks are replayed for debugging.

The subroutine to synchronize read operations (see Figure 4) uses a semaphore to guarantee exclusive access to the shared variable. The subroutine increments the

```
Shared-Read (Variable, &Value, Process)
 {
   if (REPLAY) {
      CurrentEntry = ReadMonitorBuffer();
      while (Variable.AccessCount[Process] ==
             CurrentEntry.Variable.AccessCount[Process]){
         pause;
         CurrentEntry = ReadMonitorBuffer();
      }
   }
   P(Variable.Lock);
   Variable. AccessCount [Process]++ ;
   Value = Variable.Value;
   V(Variable. Lock);
 }
```

Figure 4. Read shared variable algorithm

reading task's access count for that variable and performs the read before releasing the semaphore. During replay, a task can only read a shared variable if all preceding write operations have been replayed. This will be the case if the task's access count is less than the value recorded in the next logged write event of the monitor log. If the two values are equal, a pending write must take place before the access counter changes, so the reading task must wait for the next write to be replayed.

The shared variable write subroutine (see Figure 5) also uses a semaphore to guarantee exclusive access to the shared variable. All of the variable's access counter values are recorded in the monitor log before the writing task's access count is incremented and the write operation is performed. During replay, a writing task must wait until all preceding read and write operations have occurred. Once the variable's access counts match the recorded values in the logged write event, all of the logged operations have been replayed so the write can take place.

Figure 6 shows a sample synchronized execution of three parallel tasks accessing a single shared variable. The access counter values written to the log and the values in memory after each operation is completed are listed to the right. During replay, the first two read operations are allowed to happen in any order. The third read would be forced to wait until after the first write. Only after this write would the next log entry hold the access counter values (5,8, 11) of the second write, allowing the read to occur since the recorded counter value (11) exceeds the current value (10). In addition, each write operation would be forced to be replayed in exactly the same order as logged.

The access counters we use are a class of monotonically increasing timestamps. Other researchers have used similar ideas in different contexts. For example, the

```
SharedWrite (Variable, NewValue, Process)
 {
    if (REPLAY) {
       Current Entry = ReadMonitorBuffer();
       while ((Process != CurrentEntry.Process) or
              (Variable.AccessCount [0..N] <
               CurrentEntry.Variable.AccessCount [0.. N] )) {
          pause;
          CurrentEntry = ReadMonitorBuffer();
       }
       P(Variable.Lock);
       IncrementMonitorBufferPointer();
       Variable.AccessCount [process]++ ;
       Variable.Value = NewValue;
       V(Variable.Lock);
    } else {
       P(Variable.Lock);
       WriteMonitorBuffer(Variable, Process, VariableAccessCount [0.. N]);
       Variable.AccessCount [Process]++ ;
       ariable.Value = NewValue;
       V(Variable.Lock);
    }
 }
```

Figure 5. Write shared variable algorithm

Process 1	Process 2	Process 3	Written To Log	Access Count After Operation
Read(S)				(5,7,9)
		Read(S)		(5,7,10)
	Write(S)		(5,7,10)	(5,8,10)
		Read(S)		(5,8,11)
Write(S)			(5,8,11)	(6,8,11)

Figure 6. Shared variable synchronization example

protocol described in Reference 29 uses monotonically increasing timestamps. To the best of our knowledge, the first use of monotonic timestamps for distributed debugging was by Schiffenbauer, [32] in a loosely-coupled environment of Xerox Alto workstations. This system used the notion of a *logical* clock which was automatically incremented by the Alto hardware at regular intervals. However, their approach differs from ours in significant ways. The logical clock corresponding to a process in this system progressed whether or not the process itself was active. Unfortunately, that view resulted in some problems. For example, they describe considerable difficulty in extending the concept to debugging a group of processes that share a monitor. That would suggest that their technique is not applicable to debugging processes that share variables.

Monitoring shared variable operations in order to replay execution was first introduced in a debugging system called *Instant Replay.* [33] Although HMON and Instant Replay monitor shared variables differently, the goals are similar. Instant Replay enables parallel programs to be debugged by reproducing their execution behavior. All process interactions are modeled as operations on shared objects. For each shared object, the system maintains a version number and a count of the number of times the version was read. Write operations increment the current version number. Each task logs the current version number of each shared object as it is accessed.

Instant Replay works well for parallel applications, but is not applicable to real-time systems because unlike HMON, it does not monitor timing behavior, context switches, or system calls. Instant Replay also loses some information about timing errors because each task records data in its own log. In order to detect scheduling errors and other timing errors, we use a common log for all tasks on a processor to totally order monitored operations. In addition, our method requires us to log data only for write operations, not for all references. Read operations are replayed correctly from the data logged by the write operations. Therefore, HMON uses less log space if the number of shared variable read operations dominates the number of writes. Finally, Instant Replay adds unnecessary complexity by modeling message passing operations as operations on shared memory. HMON handles message passing by monitoring the system calls that process messages.

Monitoring application events

Application-specific events are extracted from user tasks through calls to a monitoring procedure that the programmer inserts as appropriate. While such calls will affect the running time of the application tasks, the effects of the monitor code are predictable because its execution time is constant.

PROCESSING MONITORED DATA

Special monitor code runs on its own processor on each HARTS node in order to reduce the interference from the monitor. We refer to this processor as the *monitor processor.* We dedicate a processor to the monitor to allow monitored data to be processed locally on the node without hampering the real-time tasks on other processors. Because the monitor processor shares memory with the application processors, it is able to provide scheduling feedback and debugging support. Either the Ethernet processor or an application processor could be dedicated as the monitor processor. We chose an application processor to demonstrate generality and applicability to other shared memory multiprocessors.

A block of memory on the monitor processor is dedicated to holding monitored data for each application processor. Data extraction code on the application processors writes data directly to the monitor processor memory over the node's shared system bus. The monitor processor retrieves the data from the application processor buffers by periodic polling. Thus, application processors are not affected if a monitor processor fails. More importantly, monitoring can be enabled at any time, without changing the interference, simply by starting to poll. This means production systems can be debugged after development. Post-development debugging is identified in Reference 34 as an issue important to users. It is especially vital in real-time systems since they may be operated and maintained for tens of years.

The monitor processor creates a partial order for events on different application processors by periodic sampling. At regular intervals, the monitor processor samples the status of each application processor buffer into another log. Each pair of consecutive samples defines a time interval. Each monitored event occurs between two consecutive samplings, so events on any application processor during two different intervals are totally ordered.

Before sending the data from the node to an external user-level process running on a non-real-time workstation outside the HARTS system, each monitor processor compresses the log data in order to reduce overhead and transmission time. The user process receives and archives data coming in from all monitor processors so that the events can be replayed for debugging. The monitor processors use the Ethernet controller on each node to send their data to the user workstation. The real-time hexagonal mesh network remains unaffected by the transmission of data over the Ethernet.

USING THE MONITOR FOR DEBUGGING

The HMON monitor is intended to be a general-purpose monitor suitable for many tasks. One important task is debugging distributed real-time applications. A common and effective method of debugging is *cyclic debugging,* where the programmer follows a cycle of executing the application, watching for errors, modifying the code, and

re-executing to observe the results. However, distributed real-time applications do not lend themselves to cyclic debugging for two reasons. Firstly, they are often nondeterministic: two executions of the same application with the same input data may not generate the same output. Program execution is nondeterministic because the timing of interactions between tasks may vary. Secondly, modifying the code of a real-time application can nondeterministically change the results by altering the timing and order of events.

HMON supports debugging by providing deterministic replay of distributed real-time applications. While the code of the application must not be modified, HMON enables the programmer to deterministically replay an monitored execution in order to study an application in detail.

HMON provides support for deterministic replay on HARTS by allowing the user to correctly reconstruct the timing and execution of monitored events. Events during replay do not match the original execution in 'wall clock time' but the effects on the tasks are the same. Real time is mapped to software instruction counter and real-time clock values, which are recorded. In all respects, the replayed execution matches the original, although it is slower if breakpoints are used.

There are two sources of nondeterminism in HARTS: interrupts and shared variable accesses. Because interrupts cause an unscheduled transfer of control, their timing can alter the execution of an application. The order that each process gains access to shared variables can also alter the resulting execution because of race conditions.

HMON can reproduce an interrupt during replay because it records the task's software instruction counter value when an interrupt occurs. During replay, the interrupt is invoked by a trap instruction inserted into the application code at the same program counter and software instruction counter value. This technique guarantees that interrupts occur deterministically in replay.

Shared variable accesses are also replayed in the original order. During replay, HMON guarantees the same order of write operations by ensuring that the access counters have the same value as during the original execution. Write operations that start too soon are suspended until the access counters reach the correct values. Read operations need not occur in their original order, but they do need to occur between the same two write operations. HMON uses the access counters recorded in the log from the write operations to ensure that reads happen during the correct interval.

Tasks also interact through messages and signals. Tasks cause these synchronous events through systems calls. HMON can deterministically replay these events because it records their order. Recording the order of these events is sufficient because replayed tasks will generate the same data values if the order of events is the same. However, both the order and value of data items received from the environment and the real-time clock must be recorded since these are not generated by replayed tasks. Monitoring code collects this data during system calls.

Debugging is straightforward once deterministic replay is guaranteed. On a single processor, tasks are scheduled in the same order because all interprocess events and clock interrupts are replayed. Across processors, the monitoring routines ensure correct synchronization. Interprocess events that occur too soon during replay are suspended waiting for the matching remote event. The programmer can insert breakpoints in any task of a distributed application. When the breakpoint is taken, all tasks that interact with the halted task will eventually become suspended, waiting

for the halted task to complete some message, signal, or shared variable operation. Once the halted task is resumed, execution will continue for all tasks. Waiting tasks will not suffer from timing errors, such as real-time clock timers running out, because all interrupts are deterministically replayed as they occurred in the original execution.

Once a process is halted by breakpoint, the programmer can conduct detailed observations of the state of the processor, including register values and memory contents. Additional breakpoints can be set before execution is resumed. The entire execution trace can be replayed as many times as necessary.

Johnson [35] states that two fundamental capabilities of a debugger are control over execution flow and control over the program's data state. Debuggers for distributed real-time systems must also be able to control the timing of asynchronous events, task scheduling, and the order of interprocessor events. HMON provides full replay of the order and timing of events, as well as task scheduling. System execution can be studied in detail, but event order and timing cannot be altered since this could nondeterministically change the execution. Enabling a programmer to alter an execution during replay could greatly improve the debugging of distributed real-time systems. We are studying the feasibility and implications of this concept.

PERFORMANCE MEASUREMENTS

In order to determine the interference of the monitor, we have timed the performance overhead on several operations. This interference is low and predictable, so the real-time properties of the system are not violated. Timing measurements were made by performing each operation between 1000 and 1,000,000 times, recording the elapsed time, and then averaging over the number of iterations. Over so many iterations, any irregularities in the kernel clock are effectively suppressed. The clock values were read from the pSOS kernel software clock, which is accurate to 1 millisecond.

The first set of data displays the interference of the monitoring code on a sample set of pSOS system calls from a C-language application (see Table I). Each call takes an average of 20 μs longer.

Our second table displays the interference of monitoring on shared variable references (see Table II). These values represent the time required for a C-language application to call the shared variable synchronization subroutines. The monitoring overhead on individual read operations is only 17 per cent. The high overhead on writes will have lower impact on actual applications because reads commonly dominate writes. If the ratio of reads to writes is 4 to 1, the average overhead drops to

Table I. Monitor interference on pSOS System calls

System call		Time (μs) Without monitoring	With monitoring
spawn_p	(create process)	201·2	222·9
activate_p	(start process)	152·3	174·0
delete_p	(delete process)	208·4	230·1
signal_v	(send signal)	95·3	114·4
send_x	(send message)	119·9	139·5
req_x	(receive message)	110·7	129·9

Table II. Monitor interference on shared variable operations

Operation	Time (μs) Without monitoring	With monitoring
Read shared variable	21·82	25·57
Write shared variable	21·76	45·02

Table III. Monitor interference on FFT polynomial multiplication

Measurement	Degree of polynomials 8	16	32	64
Time (s)	6·908	18·116	44·973	107,771
Time with HMON	7·080	18·518	45·994	110·064
Time difference	0·172	0·402	0·971	2·293
HMON overhead (per cent)	*2·5*	*2·2*	2·1	2·1

Table IV. Monitor interference on busy beaver task

Clock interrupt frequency (Hz)	100	250	500	750	1000
HMON overhead (per cent)	0·01	0·07	0·29	0·66	1·2

35 per cent. We cannot compare our performance to that of Instant Replay, [33] since the overhead incurred on individual operations is not stated in Reference 33. However, the overhead on an entire application execution can be compared.

Our third table shows the interference on a sample bounded buffer application of polynomial multiplication with one producer and one consumer on parallel processors. The producer task generated two arrays of random polynomial coefficients in shared memory, and then signalled the consumer task. The consumer multiplied these polynomials using Fast Fourier transforms and then requested another set of numbers. This sequence repeated for 100 iterations. Our measurements are presented in Table III. While our overhead is greater than the 1 per cent incurred by Instant Replay, Table IV shows that most of our overhead is from monitoring 1000 clock tick system calls every second. We feel that the net overhead is low enough to be acceptable for a real-time system monitor, since it is predictable.

CONCLUSIONS

We have presented a monitoring and debugging system for a distributed real-time computer system that provides predictable, transparent monitoring. The monitor uses software support and some dedicated system hardware for flexibility. Our monitor enables deterministic replay of tasks by reproducing nondeterministic events. Our approach to monitoring shared variables is novel and helps detect timing errors. Our performance measurements have shown that the overhead of monitoring is

acceptable. Our approach is applicable to other general-purpose real-time multiprocessors since we do not add any special hardware to the system.

In the current version of HMON, monitoring data is collected on all pSOS system calls. HARTOS system calls are monitored at both the source and destination nodes. Context switches and interrupt handlers have been instrumented with monitoring and replay code. Shared variable synchronization, monitoring, and replay is supported. We provide a utility to instrument applications with software instruction counter code. Preliminary code for the monitor processor collects data from the application processors and displays the logs on a user workstation.

Additional work is continuing on this project to improve the debugging features. An improved debugger user interface will be developed using the Gnu Debugger. Code on the monitor processor and network processor will be further developed to enhance replays and network monitoring, and support CPU scheduling. Finally, a utility to analyze monitored data on a user workstation will be explored.

ACKNOWLEDGEMENTS

We are grateful to Prof. Kang Shin and Dilip Kandlur of the Real-Time Computing Laboratory at the University of Michigan for useful discussions during the progress of this work, and to one of the anonymous reviewers for bringing the early work of R. D. Schiffenbauer to our attention.

The work in this report is supported in part by the U.S. Office of Naval Research under Contract No. N00014-85-K-0122.

REFERENCES

1. J. Gait, 'A probe effect in concurrent programs', *Software–Practice and Experience,* **16,** 225–233 (1986).
2. H. Agrawal, R. A. DeMillo and E. H. Spafford, 'An execution backtracking approach to debugging', *lEEE Software,* **8,** (3), 21–26 (1991).
3. D. Bhatt, A. Ghonami and R. Ramanujan, 'An instrumented testbed for real-time distributed systems development', *Proc. 7th IEEE Real-Time Systems Symposium, 1987,* pp. 241-250.
4. P. Corsini and C. A. Prete, 'Multibug: interactive debugging in distributed systems', *IEEE Micro,* **6,** (3), 26–33 (1986).
5. G. S. Goldszmidt, S. Katz and S. Yemini, 'High-level language debugging for concurrent programs', *ACM Trans. Computer Systems,* **8,** (4), 311–336 (1990).
6. F. Halsall and S. C. Hui, 'Performance monitoring and evaluation of large embedded systems', *Software Engineering Journal,* **2,** (5), 184–192 (1987).
7. R. E. McLear, D. M. Scheibelhut and E. Tammaru, 'Guidelines for creating a debuggable processor', *Proc. Ist Int. Conf on Architectural Support for Programming Languages and Operating Systems,* published in *ACM SIGPLAN* Notices, **17,** (4), 100–106 (1982).
8. A. D. Maio, S. Ceri and S. C. Reghizzi, 'Execution monitoring and debugging tool for Ada using relational algebra', *ACM Ada Letters,* **V,** (2), 109–123 (1985).
9. P. K. Rowe and B. Pagurek, 'Remedy: a real-time, multiprocessor, system level debugger', *Proc. 7th IEEE Real- Time Systems Symposium, 1987,* pp. 230–240.
10. J. C. Huang, M. Ho and T. Law, 'A simulator for real-time software debugging and testing', *Software–Practice and Experience,* **14,** 845–855 (1984).
11. T. Bemmerl and G. Schoder, 'A portable realtime multitasking kernel with debugging support', *Proc. Int. Conf. on Software Engineering for Real Time Systems, 1987,* pp. 165–171.
12. B. Lazzerini, C. A. Perle and L. Lopriore, 'A programmable debugging aid for real-time software development', *IEEE Micro,* **6,** (3), 34–42 (1986).
13. B. Sundermeier, 'Real-time multiprocessing debugging', *Western Electronic Show and Convention Conference Record,* 27/3/1–5 (1987).
14. D. Wybranietz and D. Haban, 'Monitoring and performance measuring distributed systems during

operation', *ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems,* 1988, *pp.* 197–206.

15. D. Lyttle and R. Ford, 'A symbolic debugger for real-time embedded Ada software', *Software—Practice and Experience,* **20,** 499–514 (1990).
16. J. D. Schoeffler, 'A real-time programming event monitor', *IEEE Trans. Education,* **31,** (4), 245–250 (1988).
17. G. Schrott and T. Tempelmeier, 'Monitoring of real time systems by a separate processor', *Proceedings of the 12th IFAC/IFIP Workshop on Real-Time Programming,* 1983, pp. 69–79.
18. H. Tokuda, M. Kotera and C. W. Mercer, 'A real-time monitor for a distributed real-time operating system', *ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging,* published in *ACM SIGPLAN Notices,* **24,** (l), 68–77 (1989).
19. C. A. Witschorik, 'The real-time debugging monitor for the Bell System 1A processor', *Software—Practice and Experience,* **13,** 727–743 (1983).
20. R. Cooper, 'Pilgrim: a debugger for distributed systems', *Proc. 7th Int. Conf. on Distributed Computer Systems,* 1987, 458–465.
21. *C.* R. Hill, 'A real-time microprocessor debugging technique', *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugging,* published in *A CM SIGPLAN Notices,* **18,** (8), 145–148 (1983).
22. J. J. P. Tsai, K.-Y. Fang, H.-Y. Chen and Y.-D. Bi, 'A noninterference monitoring and replay mechanism for real-time software testing and debugging', *IEEE Trans. Software Engineering,* **16,** (8) 897–916 (1990).
23. S. H. Jones, R. H. Barkan and L. D. Wittie, 'Bugnet: a real time distributed debugging system', *Proc. 6th Symposium on Reliability in Distributed Software and Database Systems,* 1987, pp. 56–65.
24. L. D. Wittie, 'Debugging distributed C programs by real time replay', *ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging,* published in *ACM SIGPLAN* Notices, **24,** (1), 57–67 (1989).
25. K.-C. Tai, R. H. Carver and E. E. Obaid, 'Debugging concurrent Ada programs by deterministic execution', *IEEE Trans. Software Engineering,* **17,** (l), 45–63 (1991).
26. M. S. Chen, K. G. Shin and D. D. Kandlur, 'Addressing, routing and broadcasting in hexagonal mesh multiprocessors', *IEEE Trans. Computers,* **C-39,** (l), 10–18 (1990).
27. D. D. Kandlur, D. L. Kiskis and K. G. Shin, 'HARTOS: a distributed real-time operating system', *ACM SIGOPS Operating Systems Review,* **23,** (3), 72–89 (1989).
28. *pSOS-68K Real-Time Operating System Kernel User's Guide,* Software Components Group, Inc., 1986.
29. L. Lamport, 'Time, clocks, and the ordering of events in a distributed system', *Communications of the ACM,* **21,** (7), 558–565 (1978).
30. T. A. Cargill and B. N. Locanthi, 'Cheap hardware support for software debugging and profiling', *Proc. 2nd Int. Conf. on Architectural Support for Programming Languages and Operating Systems,* 1987, *pp.* 82–83.
31. J. M. Mellor-Crummey and T. J. LeBlanc, 'A software instruction counter', *Proc. 3rd Int. Conf. on Architectural Support for Programming Languages and Operating Systems,* published in *ACM SIGPLAN Notices,* **24,** (special issue), 78–86 (1989).
32. R. D. Schiffenbauer, 'Interactive debugging in a distributed computational environment', MS *Thesis,* Massachusetts Institute of Technology, 1981.
33. T. J. LeBlanc and J. M. Mellor-Crummey, 'Debugging parallel programs with instant replay', *IEEE Trans. Computers,* **C-36,** (4), 471–482 (1987).
34. R. Seidner and N. Tindall, 'Interactive debug requirements', *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugging,* published in *ACM SIGPLAN* Notices, **18,** (8), 9–22 (1983).
35. M. S. Johnson, 'Some requirements for architectural support of software debugging', *Proc. 1st Int. Conf. on Architectural Support for Programming Languages and Operating Systems,* published in *ACM S[GPLAN* Notices, **17,** (4), 140–148 (1982).