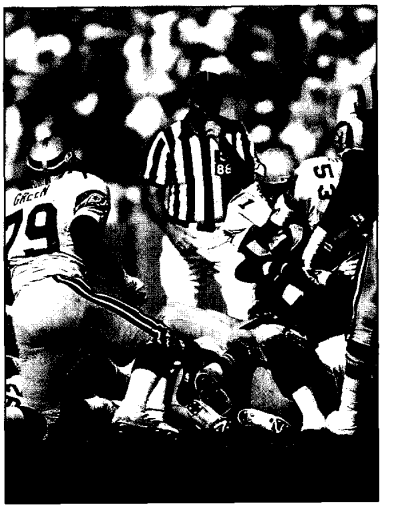
A Noninvasive Architecture to Monitor Real-Time Distributed Systems



Jeffrey J.P. Tsai, Kwang-Ya Fang, and Horng-Yuan Chen
University of Illinois at Chicago

A real-time distributed computing system consists of a collection of communicating and cooperating processors or computers (nodes) that work toward a common goal. Critical timing constraints are imposed on them.

The monitoring of real-time distributed computing systems involves the collection and interpretation of information, such as event time stamps, synchronization sequences, race conditions, register status, transaction identifications, and interrupt activities. This information can be used for testing and debugging or improving the performance of distributed computing systems.

However, monitoring a real-time distributed computing system is much more difficult than monitoring a centralized, sequential computing system because of

(1) *Multiple asynchronous processes:* Since real-time distributed computing systems feature asynchronous parallel processes, their computation shows nondeterministic and irreproducible behavior. This behavior, caused by race conditions and unpredictable synchronization among processes, makes the results observed in a distributed computing system harder to understand and, in many cases, hard to reproduce. Usually, when a software system is executed on a real-time distributed computing system (such as a local network), deducing the execution order of instructions belonging to different processes of the system is impossible.

Equipped with a programmable qualification control unit for observing system states of a target node, this monitoring system can support performance evaluation, testing, and debugging of real-time distributed computing systems.

(2) *Critical timing constraints:* For a real-time distributed computing system, the correctness of the system depends on its behavior with respect to time. Often, processes of a real-time distributed computing system are closely coupled to processes existing in the real world (such as a weapon system or a chemical plant). The correctness of a real-time distributed computing system, which depends on the performance of underlying processors, is determined by meeting the timing constraints imposed by the real-world processes. Any interference in the monitoring activity in the real-time distributed environment is intolerable.

(3) *Significant communication delays:* The nodes of a real-time distributed computing system can be geographically dispersed. This may introduce a significant communication delay, which can in turn cause improper synchronization among the processors and make it difficult to de-

0018-9162/90/0300-0011$01.00 © 1990 IEEE

termine the precise global time and accurate global state.

As a result of these differences, the execution of a real-time distributed computing system is characterized by nondeterminism, timing constraints, and the low visibility of system behavior. These characteristics make the monitoring of the global states and the execution flows of a real-time distributed computing system almost impossible. Thus, the global timing and state that represent the system's runtime behavior are difficult to observe. In addition, almost all existing monitoring systems are designed to share computing resources.[1-8]

These monitoring systems apply constant interference to a target distributed computing system, change system behavior, and degrade system performance. Therefore, these techniques are not suitable for monitoring real-time distributed computing systems. We need a new approach for system monitoring and data collection that doesn't interfere with system execution behavior.

In the past few years, research has been actively carried out in an attempt to monitor the execution behavior of distributed and parallel systems.[3,6-8] Joyce et al. presented a distributed monitoring system that can collect, interpret, and display information concerning interactions among concurrently executing processes.[3] To allow interprocess events to be easily detected, Joyce modified the processes to be monitored by loading them with a version of interprocess communication protocol that incorporates monitoring. The modified processes, called "monitorable processes," suffer a certain degree of execution speed penalty.

Plattner described an experimental real-time monitoring system that can be used to observe the behavior of real-time processes, to collect trace data, and to detect illegal or unexpected process states.[6] This monitoring system is capable of monitoring processes written in a high-level programming language, in real-time, and on a symbolic level. Plattner's experience with this experimental monitoring system shows limitations of the monitoring activity that can be achieved by the monitor.

Snodgrass views monitoring as an information-processing activity and asserted that the relational model is an appropriate formalism for structuring the information generated by a distributed computing system.[7] His work focuses on the query of a relational database model so that the user can specify what information is to be collected. This approach controls the amount of monitoring data collected and thus alleviates the invasive nature of a monitor.

Tokuda et al. presented a real-time monitor that is being built for a distributed real-time operating system.[8] This real-time monitor is an invasive monitoring system in the sense that it needs extra kernel support.

Although much research has been done and many tools for monitoring have been developed, they are still not practical for monitoring real-time distributed computing systems due to a monitor's invasive nature. In this article, we present a hardware architecture for a noninvasive monitoring of real-time distributed computing systems.

Our principal objective here is to develop a *real-time* monitoring system to ensure minimal interference in the execution of a target distributed computing system. This system should also provide users with a tool to support either top-down or bottom-up approaches for testing and debugging, as well as performance evaluation of real-time distributed computing systems.

This monitoring system extracts information directly from traffic on the internal buses of a target distributed computing system. Since modifying a target distributed computing system or sharing computing resources with the target system is not required, our architecture does not interfere with the execution of the target system and guarantees the preservation of the timing constraints.

In the following sections, we describe a model of real-time distributed computing systems; present the hardware architecture, operation, and implementation of our noninvasive monitoring system; introduce our approach to the noninvasive monitoring of real-time distributed computing systems; demonstrate how this approach can be used to support the testing and debugging of real-time distributed computing systems; summarize the advantages of our approach; and discuss the direction of future work.

A model of real-time distributed computing systems

A real-time distributed computing system, used for real-time distributed applications, consists of a hardware part and a software part. The hardware part shown in Figure 1 includes a collection of nodes and a communication network. Each node has its own CPU, memory, peripherals, and communication interface. The software part, which executes on each node of a real-time distributed computing system as shown in Figure 2, includes the operating system, the communication module, and a collection of application processes. The operating system is in charge of scheduling the application processes and managing the local resources. The communication module is responsible for internode communication.

To simplify the description of the monitoring system, we assume that the target system is a dedicated real-time distributed computing system. Furthermore, our approach toward noninvasive monitoring is based on the following assumptions* about the monitored system:

(1) It is a distributed/multiprocessor system with a master node (processor) and a set of slave nodes as shown in Figure 1. The master node controls all interactions among the slave nodes, and each node has its own local storage. Since the monitoring system is independent of the communication subsystem, there is no further restriction on the topology of connection of real-time distributed computing systems.

(2) Communications among processes residing on a node take place via shared variables, while the interactions among processes residing on different nodes take place through message exchanges.

(3) Process migration among nodes is not allowed during the monitoring of distributed computing systems. Thus, the communication subsystem is dedicated to communication activities among nodes.

(4) Processes perform input and output operations by asking for service from the operating system. Input to a target distributed computing system is asynchronous. This assumption, together with the following assumptions, eases the identification of process-level and function-level events (see the section entitled "Monitoring in different abstraction levels").

(5) For the application software, we assume that procedural calls are implemented by system calls, while recursive calls of a procedure are not allowed. In a procedural call, the symbolic name of a

* The main purpose of these assumptions is to ease the identification of the events of interest and hence to simplify the trigger conditions and reduce the complexity of the postprocessing mechanism.

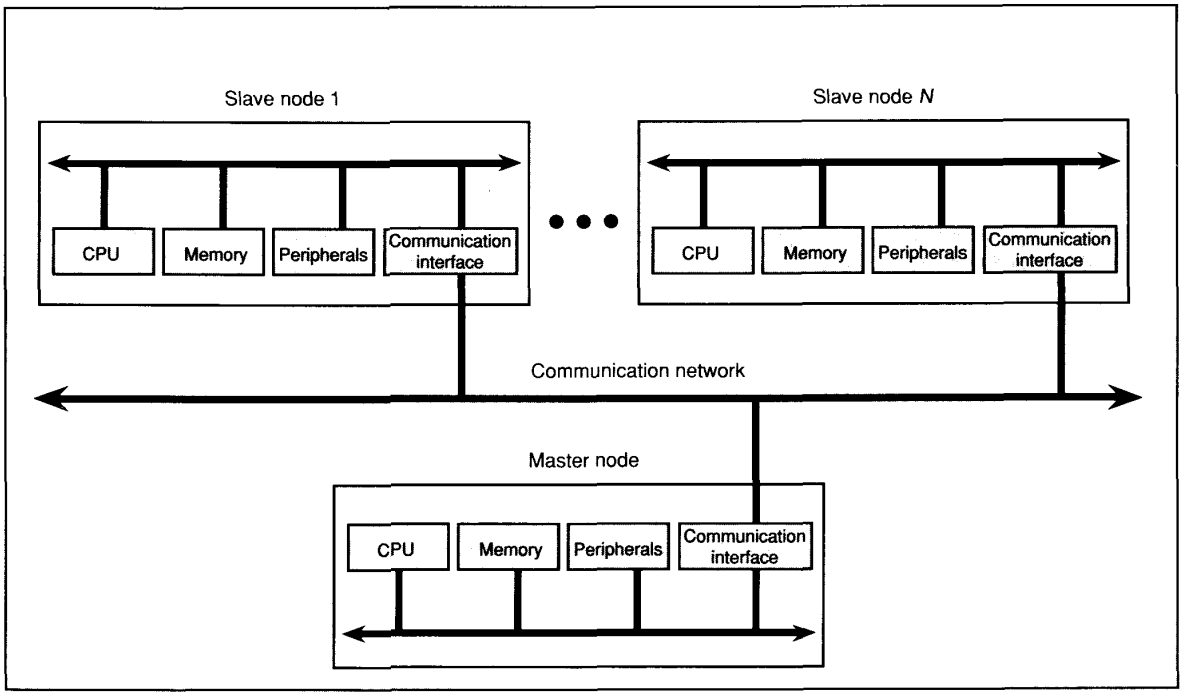


Figure 1. Hardware view of a real-time distributed computing system.

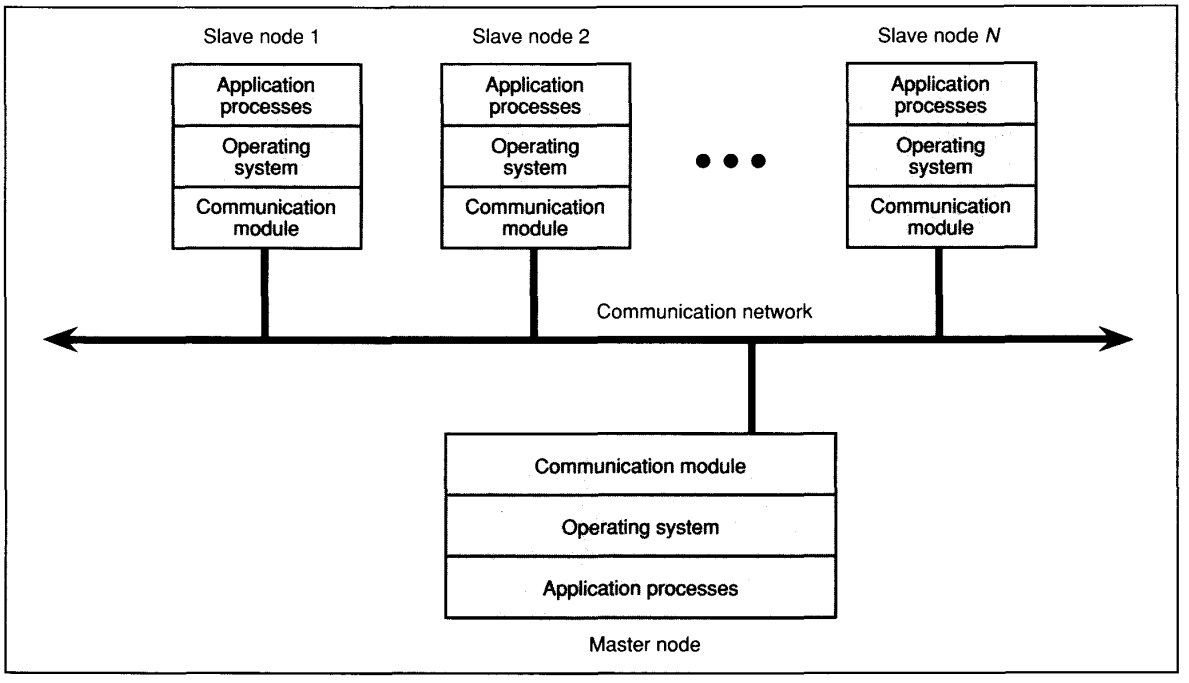


Figure 2. Software view of a real-time distributed computing system.

procedure can uniquely be identified. Device interrupts can be separated using unique device identification in the collected trace. All computational entities (processes/procedures) should be addressable using names and descriptors maintained in a system symbol table.

(6) Our current design is based on the MC68000 computer. If the target system is not the same kind of hardware architecture (for instance, in modern distributed systems that utilize caching and prefetching), the monitoring system architecture and the postprocessing mechanism will require modification.

An architecture for real-time distributed computing system monitoring

The system architecture of the noninvasive monitoring system, as shown in Figure 3, consists of two major components: the *interface module* and the *development module*.

The interface module can be considered as the front end of the monitoring system. This module is specially designed to attach to and interface with the specific real-time distributed computing system for system monitoring and data collection. Its main function is to latch the internal states of the target system based on predefined conditions set by the user.

The development module is the host computer for the interface module. This module is a general-purpose microprocessor-based system that contains all the supporting software for the initialization of the interface module and postprocessing activities.

Since the noninvasive monitoring system does not steal CPU time from the target real-time distributed computing system, it does not interfere with target system execution. The monitoring system provides noninvasive features of system monitoring and execution history recording by connecting itself with the internal buses of the monitored node and by latching data directly from them. Referring to the system architecture shown in Figure 3, the monitoring system, after being connected to a node of the target system and initialized, continues monitoring and collecting events of interest until it detects a "stop" condition, specified by the user. Execution history recording is controlled by the qualification control unit (Figure 4) in the interface module. The QCU samples the state of the target processor via its internal buses during every processor cycle. A user-selected state or breakpoint is used as a trigger to initiate and to stop recording of the runtime information.

The interface module. The main function of the interface module is to copy the internal states of the target node's processor and, under predefined trigger conditions, to start recording data from the buses of the target node onto the memory buffer unit. As shown in Figure 4, the interface module consists of five functional units:

- the interface control unit (ICU),
- the dual processor unit (DPU),
- the dual processor memory (DPM),
- the qualification control unit (QCU), and
- the high-speed buffer unit (HSBU).

The monitoring system is connected to the target node via the ICU. The DPU contains a dual processor identical to the target processor and the DPM, which is used for the synchronization of the dual processor with the target processor. The main function of the DPU is to snapshot the target node processor's state at the beginning of recording the target node's execution behavior.

To achieve a true noninvasive condition during the time of execution history recording, the DPU is designed to imitate the target processor after it has been initiated and synchronized until the recording of the execution history is started. Therefore, the corresponding state of the target processor at the beginning of execution history can be kept in the dual processor.

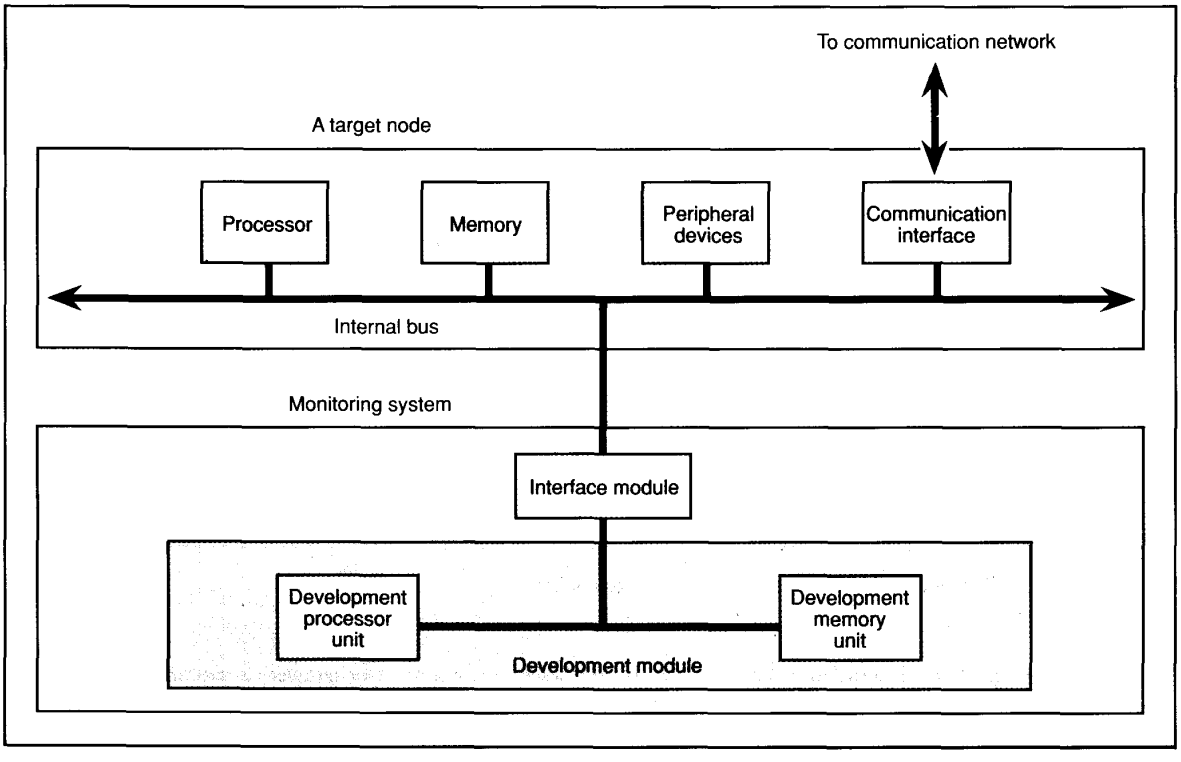


Figure 3. System overview.

The QCU processes the preset trigger condition into a control signal to initiate and stop data recording. Usually, the speed of the target node's system buses is very high. To store this high volume of latched data, we need a lot of high-speed memory, which is very expensive. Hence, we adopt a cheaper design method in the HSBU. Instead of using a lot of high-speed memory, we use a high-speed register as a buffer between the target node's system buses and the development module's memory. The latched data is first written into a high-speed buffer and then transferred into the development module's memory.

To initialize the monitoring system, a low-priority interrupt is sent by the DPU for initializing and synchronizing itself with the target node's processor. This interrupt is the only interference in the target node's processor by the monitoring system.

The purpose of sending such a low-priority interrupt signal is to avoid interfering with the target processor, especially when the target system is performing critical timing tasks. Because of its low priority, the impact of the interrupt on the target node is assumed to be minimal.

If the target system is performing a critical computing task, or if it is in a heavy work-load condition, the response to the low-priority interrupt may be delayed. The interference caused by this interrupt is minimal and reasonably tolerable because the initialization of the interface module is only executed at noncritical and nonbusy times.

This interrupt signal triggers an interrupt service routine in the target node to start initialization of the interface module by copying all register contents to the DPU. After the DPU is initialized and both of the processors are synchronized, the monitoring system is ready to monitor the target node's execution behavior and collect runtime information.

After the first step of initialization, the dual processor starts to mimic the operation of the target node processor without physically accessing the memory and peripheral devices of the target node.

This synchronization remains until a starting trigger is matched in the QCU. In this case, the starting trigger is matched and the QCU generates and sends control signals to the HSBU and DPU to indicate that a preset condition has been matched. As soon as the control signal is received, the dual processor isolates itself from the target node to freeze and save the internal

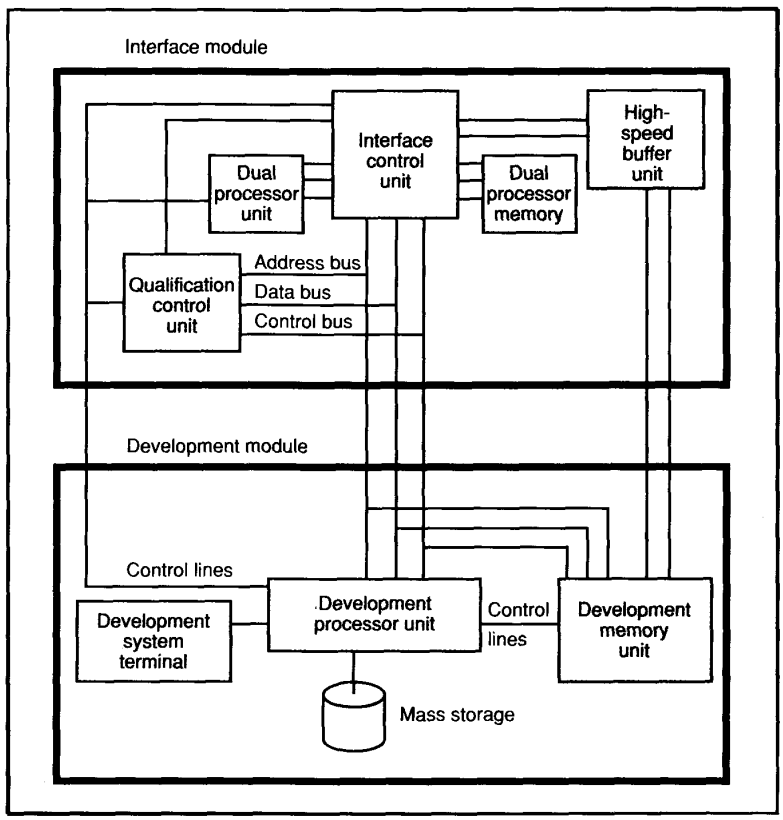


Figure 4. Interface module and development module.

state of the target processor in the dual processor memory and, simultaneously, the HSBU starts loading data from the buses of the target node.

The data acquisition rate is determined by the speed of the communication bus of the target system. The monitoring activity continues until the preset stopping trigger is matched in the QCU. At this point, the QCU sends a signal to the HSBU to stop monitoring the target node's execution behavior. The recorded information is then transferred to the development memory unit for postprocessing.

Interface control unit. The function of the ICU is to establish a connection between the target node and the monitoring system without creating electrical interruptions for the target processor. When the monitoring system and target node are connected, the data, address, and control signals are connected by a two-stage tri-state buffer switch, as shown in Figure 5.

The first-stage switch is controlled by an initialization done signal. The second-stage switch is controlled by a control circuit in the interface module. The On or Off status of the second-stage switch indicates whether the dual processor is running in synchronization with the target processor or running independently. When the synchronization process is completed (Off), the switch becomes active (On) and the logical connection between the target processor and the dual processor via data and control buses is established.

When a starting trigger is matched in the QCU, the dual processor separates itself from the target processor and starts running independently. At this time, the second-stage switch is deactivated and the logical connection between two processors is disconnected.

The ICU is target node-dependent and constructed to support synchronization and concurrent operation with the dual processor on the same bus as the target processor. The ICU reconfigures the memory structures so that, during the dual processor's imitation of the target processor, memory write operations from the dual processor are intercepted without physically accessing the target node's

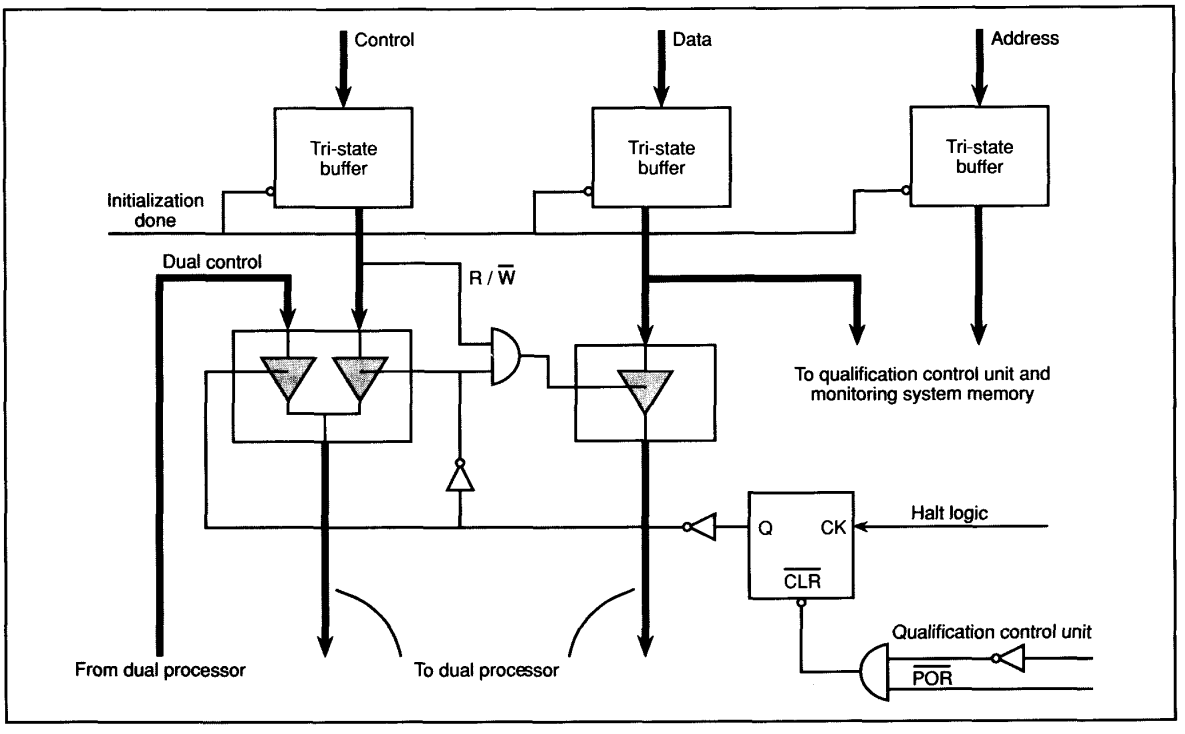


Figure 5. Interface control unit (ICU).

memory, avoiding memory contention caused by two processors.

Dual processor unit. The DPU is designed to capture the target processor's internal state at the beginning of monitoring. To achieve this, the DPU must be initialized and synchronized with the target processor. After both processors are synchronized, the dual processor reads the target processor's data bus; thus, it will read the same instruction as the target processor and will start mimicking the target node processor. The sequence of activities performed in the initialization of the DPU can be briefly described as follows:

(1) Reset the dual processor and interrupt the target node processor at low priority.
(2) Wait for the target processor to transfer its internal state.
(3) Synchronize the dual processor with the target processor.

At the time of power-up, the dual processor is reset to prepare it for the transfer of information from the target processor and to interrupt the target processor at low priority to get its attention for transferring the data. The DPU then waits for the response from the target processor so it is synchronized with the target processor.

Synchronization allows the dual processor to concurrently mimic the target processor's activity until the recording triggers are activated. Different synchronization techniques can be used to create a communication link between two processors. The direct synchronization technique, which was originally designed to utilize the processor's ability to communicate with slow peripheral devices, is used to achieve the required minimal interference with the target processor.

To mimic the behavior of the target processor, the program counter, status flags, and internal registers of the target processor need to be stored to the dual processor. After the target processor responds to the interrupt from the development processor, the interrupt routine transfers all the internal registers of the target processor to a dummy I/O peripheral: the dual processor.

The dual processor is reset by the "intake" signal from the target processor and starts its own service routine to read the data bus of the target processor through a dummy I/O port. Multiplexers are used to switch the data bus from the dual processor to the target processor. Whenever the dual processor issues an I/O read signal to the dummy I/O port, this signal is combined with the I/O address to switch the dual processor's data bus toward the target processor's data bus. Thus, whenever the dual processor reads that particular I/O device, the data is read from the target processor's data bus.

Whenever the target processor writes to a dummy I/O address, it temporarily puts itself into a wait state. It does this by using I/O write control of the target processor as a "device-not-ready" signal to the target processor via a flip-flop. This flip-flop remains set until the dual processor again reads its dummy I/O port, which resets it.

There are two comparators, C1 and C2. C1 is connected to the target processor's data bus, and C2 is connected to the dual processor's data bus. These comparators compare the data flowing through the data bus to detect the last instruction in the target processor's service routine (which is instruction RTI, for return from interrupt).

When this instruction is fetched by the target processor, the C1 comparator issues a "match" signal. This output signal is connected to the device-not-ready line of the target processor to force it into a wait state. Similarly when the dual processor

fetches its last instruction (which can be set by the designer), the C2 comparator will also issue a "match" signal to force the dual processor into a wait state by pulling the dual processor's device-not-ready line active.

Since both processors share the same clock, their wait states will be the same. After the dual processor is forced into a wait state, the "match" signal is used to switch the data, control, and address buses of the dual processor to the target processor. After switching, the dual processor can only read the data and the address buses of the target processor.

At the time of the write operation, the write signal of the dual processor is used to switch its data bus to a register. Thus, at every write operation, the dual processor will write to this register and, at every read operation, it will read the target processor's data bus. In addition, the dual processor will get all the incoming control signals from the target processor, although the outgoing signals of the dual processors are floated.

After the delay of all this switching, the match signal of the comparator (C2), which is connected to the dual processor, is used to bring both the processors out of their wait states by pulling the device-not-ready line inactive. When the target processor comes out of the wait state, it executes the return instruction. The dual processor and the target processor will receive the same instruction since they share the same data bus. At this moment, two processors are performing the same operations with the target processor as the master of the bus control.

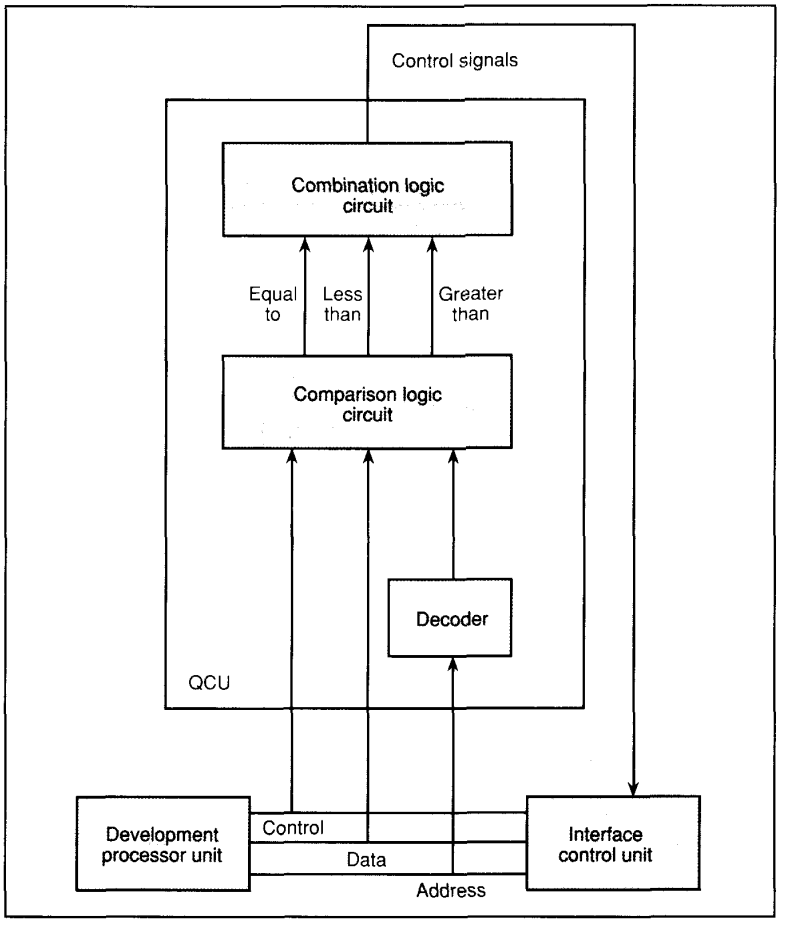


Figure 6. Qualification control unit (QCU).

Qualification control unit. Figure 6 shows the block diagram of the QCU. The development processor unit can set the QCU to various conditions. Besides the decoder, the design shown in Figure 6 is simply divided into two basic blocks. One is the comparison logic circuit, which is composed of a number of comparators. Each comparator can issue three basic condition signals, "equal," "greater than," and "less than," representing these different conditions based on the result of the comparison. The other circuit is the combination logic that combines the basic conditions generated by the comparison unit and generates control signals.

The QCU monitors system execution by sampling the distributed computing states of the target processor during each bus cycle. If the sampled state matches any user-specified conditions, the recording of distributed execution history is triggered. At this point, the dual processor receives an interrupt signal from the QCU and separates itself from the target processor. An image of the target processor's state corresponding to the beginning of the execution history is therefore frozen in the dual processor.

The QCU consists of a logic circuit that compares the signals on the buses of the target processor against the preset trigger conditions. The QCU may be designed with multiple counters to record the prehistory, posthistory, or some combination relative to the occurrence of a trigger condition.

In addition to simple concurrent comparisons, the trigger condition may also be set up to be sequenced in hardware. That is, a given trigger can be initiated to become active only if another trigger occurs first. The length of the execution history is controlled by a trace counter to count the number of cycles being recorded, or by the detection of a stopping trigger specified by the user. When a predefined length of the execution history has been recorded, the recording process is automatically stopped.

The QCU is actually a hardware implementation of software breakpoints defined by the user through the development processor. The comparison logic circuit in the QCU is primarily a programmable logic array design that gives the maximum flexibility required to support general software breakpoint conditions. The specific timing constraint is a critical issue in the design of the QCU because the comparison operation must be completed within one bus cycle and the control signals are also generated accordingly.

The development module. The development module, which consists of the development processor unit and the devel-

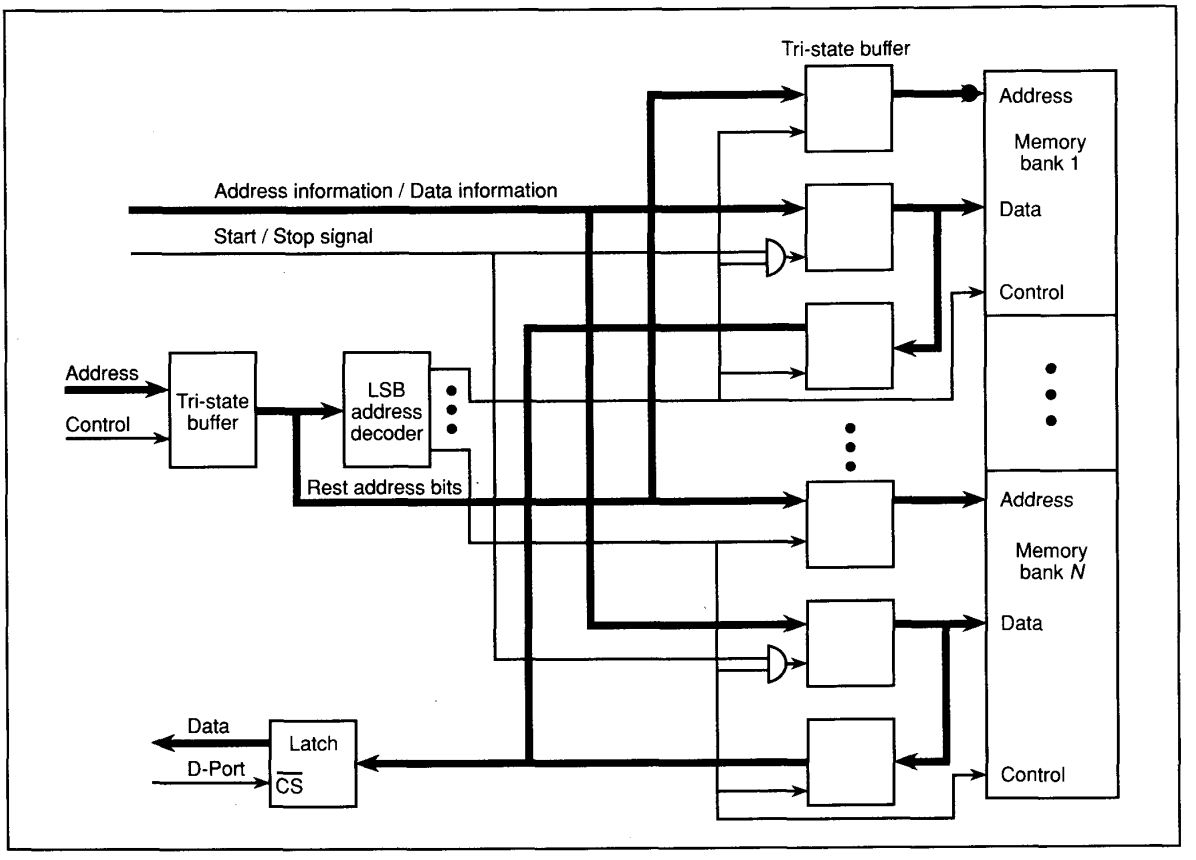


Figure 7. Memory interleaved technique.

opment memory unit, is the central processing unit of the monitoring system and functions as a host machine. The development module is basically independent of the target node processor. Independence is achieved by separating the target-dependent functions into the interface module. The development module provides an interactive interface to the user and is responsible for all the testing and debugging activities, including

(1) initialization of the monitoring system,
(2) controlling the interface module to latch the target node execution history, and
(3) performing postprocessing on recorded execution history.

Postprocessing, a very important activity supported by the development processor unit, is basically system dependent. Its main function is to process the recorded execution history (see the section entitled "Postprocessing of traces"). The other functions of the development processor are

(1) To initialize the interface module. On "system power-on" or "system reset," the development processor initiates a system hardware initialization and loads the checkpoint (breakpoint) conditions into the QCU.

(2) To provide a user-friendly interface. The development processor provides a user-friendly interface under various situations such as prompts for trigger condition setup or display messages on various system activities.

The development memory unit consists of two parts: the development processor memory and the memory configuration unit. Through the HSBU, the program execution history from the target node is latched into the memory configuration unit.

The memory interleaved technique, applied in designing the memory configuration unit, is used to give sufficient time for memory access when the bus cycle of the target processor is much faster than the memory used (see Figure 7). By applying this, cheaper memory (such as dynamic cell memory, which has higher storage density but needs more access time) can be used in the development processor memory.

Memory is divided into several banks depending on the speed of the memory and the length of the target processor bus cycle. The slower the memory or the faster the bus cycle, the more memory banks are required. The memory bank is identified by the least significant address bits, and the remainder of the address bits specify address within a memory bank. The least significant memory address bits are decoded to generate a memory bank selection control signal that selects a different memory bank at each bus cycle so sufficient time is given to access the memory bank.

Monitoring in different abstraction levels

Different levels of monitoring provide different levels of detail for different application purposes. Higher level monitoring is concerned with information pertaining to events such as internode communication. By comparison, lower level monitoring is concerned with information that pertains to events such as the step-by-step execution trace of a process.

The monitoring system presented here supports different abstraction levels of monitoring according to application purposes. It can monitor process-level activities (for example, internode or intranode communication) and function-level activities (for example, procedure calls) as well as instruction-level activities (for example, step-by-step instruction trace). A detailed description of process-level monitoring is given in the following section.

The monitoring system achieves different levels of monitoring by setting the QCU according to the classes of events it is interested in. Based on the purpose of monitoring, the users may set up their own trigger conditions on the QCU to record specific classes of events.

Due to the flexibility of the QCU, the monitoring system can support either bottom-up or top-down approaches to testing and debugging, as well as system performance evaluation of real-time distributed computing systems. For instance, to reduce the complexity of the debugging procedure, we may adopt a top-down approach to the monitoring of a real-time distributed computing system.

Initially, the monitoring system is attached to the master node to observe execution of the target distributed system. The execution behavior of the master node is watched and recorded. To examine the interprocess interaction between nodes, additional monitoring systems need to attach simultaneously to these corresponding nodes.

Finally, if the user is interested in the behavior of an individual process, the monitoring system is attached to the node where the process resides so the events pertaining to the behavior of the process can be collected by setting up the QCU appropriately.

Process-level monitoring. As stated above, monitoring process-level activities provides a high-level view of the target system's behavior. To characterize this behavior, we identify the following events with specified key values as typical process-level events:

(1) *Creating and terminating process:*
   (a) Creating process: parent process identification, child process identification, time.
   (b) Terminating process: parent process identification, terminating process identification, terminating state, time.

(2) *Process synchronization:* process identification, operation (P/V), semaphore identification (see the sidebar), the value of the semaphore, time.

(3) *Interprocess communication:*
   (a) Direct communication: process identification, operation (send/receive), process identification (send to or receive from), message, time.
   (b) Indirect communication: process identification, operation (send/receive), mailbox identification, message, time.

(4) *External signals:*
   (a) Hardware clock interrupt: process identification (running process), process identification (resumed process), time.
   (b) I/O completion interrupt: process identification (running process), process identification (resumed process), message (I/O buffer), time.

To monitor the process-level events listed above, we need to preset two sets of trigger conditions into the QCU of the interface module: starting trigger conditions, which start the recording process, and stopping trigger conditions, which stop the recording process. The trigger conditions are identified such that the recording process is started when one or a sequence of events occurs. The recording process is stopped when the key values of the event(s) are included in the recorded data block.

The control flow of monitoring can be described as follows. The monitoring system is initialized and connected to the individual node of the target distributed computing system. The identified starting and stopping trigger conditions are initialized in the QCU of the interface module.

During system execution, the QCU samples the states of the target system. The QCU continues sampling and recording the process-level events until the execution of the monitored system is stopped or until a "stop" condition, specified by the user, is detected.

The activities of the process-level monitoring of a real-time distributed computing system can be summarized as follows:

(1) connect the monitoring system to a node of the target distributed computing system without electronic interruption,
(2) load trigger conditions for start and stop of process-level events recording,
(3) initialize the interface module,
(4) record process-level information, and
(5) transfer recorded information to secondary storage for post-processing.

Setting trigger conditions for process-level monitoring. To identify the starting and stopping trigger conditions, we next describe how the process-level events

Semaphores

Semaphores are a commonly used mechanism for implementing process synchronization. A semaphore is an integer variable *S* for an associated group of waiting processes upon which only two operations, P and V, may be performed:

P(S): **if** S > 0 **then**
        S = S - 1
    else the executing process is suspended and placed in S's waiting group
    endif

V(S): **if** S's waiting queue is nonempty **then**
        remove one waiting process and wake it up for execution
    else S = S + 1
    endif

occur in a real-time distributed computing system. In our discussion, the operating system is treated as an interrupt-driven process.[9] When an interrupt (or trap) occurs, the hardware transfers control to the operating system, which then determines what kind of interrupt has occurred. A process requests system resources or services from the operating system by executing a system call, a privileged instruction with parameters to specify what is requested.

When a process executes a system call, a software interrupt occurs and the operating system transfers control through a predefined interrupt vector to a system service routine. This system service routine executes the request on behalf of the process.

If the interrupt is an I/O completion from a device, the operating system awakens the process waiting for this I/O completion. All other process-level events are handled by the operating system in the same manner. For example, in the case of an indirect communication, messages are sent to and received from mailboxes. Each mailbox has a unique identification that distinguishes it. In this scheme, a process may communicate with some other processes by a number of different mailboxes.

A mailbox may be associated with more than two processes. The operating system provides two system calls to let processes send or receive messages with the mailbox's identification and the messages as parameters of the system calls.

Based on the description above, we can set the trigger conditions to identify system calls related to the process-level events. We summarize the following five conditions as starting trigger conditions for the process-level events described in the section entitled "Process-level monitoring":

IF ((system call interrupt) AND
(process related activities))
OR ((system call interrupt) AND
(I/O request))
OR (I/O completion interrupt)
OR (system clock interrupt)
OR (program error interrupt)

THEN trigger the recording process.

No matter what type of interrupts are caused by the events, the operating system will return control to an application process after the service for the interrupt is finished. This is done by the operating system executing a privileged instruction that changes the system mode to user mode. So, the stopping condition for all the events can be:

IF (instruction changes the system
mode to user mode)
THEN stop the recording process.

The information concerned with process-level events is collected from the target distributed computing system as a data block that contains the key values for the event(s). The development memory unit containing the collected data can be arranged to be large enough to save the event history.

When the QCU is detecting the next trigger condition, collected data in the development memory unit can be saved to a secondary storage so the development memory unit can be used to receive new data. By considering the time for the operating system to perform a context switch before it can respond to the new interrupt, it is easy to calculate and allocate enough memory space in the development module so continuous data recording and system monitoring can be achieved. If the context switch time is not long enough for transferring the collected data into the mass storage, the memory space required for the development module can be estimated as described below.

In the worst case, let R be the largest number of events happening continuously, and let s be the largest system service routine size (in bytes). Then the required memory space, S, will be

$$S = c \times s \times R$$

where c is a constant. Usually, it is very difficult to estimate the exact value of R. However, for a certain target system, an estimated average value of R, r, can always be obtained through experiments. Thus, on average,

$$S = c \times s \times r$$

where r is the event coming rate (happening rate), and it is target system dependent.

Since the "buffer" technique can be used in designing the development memory unit, the constant c in these equations can be very small (for example, $c = 2$). In general, the memory space, required in storing the collected data, is application- and target-system dependent. For instance, the less frequently communication/synchronization occurs among processes, the less memory space is required.

Postprocessing of traces. As described above, system behavior is monitored and recorded in the machine-level code, which is hard to comprehend and not easy to use. The collected data contains not only the key values of events but also other irrelevant instructions and data. Furthermore, interpretation of the collected data will be target-processor dependent. Therefore, it is important to eliminate the useless information and provide only the event information to the user.

When collecting traces from the target system, virtual addresses are the predominant signals on which to trigger and collect. The interpretation of what went on in the target system (that is, the identification of events as well as the mapping of physical addresses and their corresponding logical addresses) is done with the help of system tables.

In postprocessing the collected data, redundant data irrelevant to the event are removed. The collected data are then reorganized into different-level logical views of events determined by the user. The collected data is processed and reorganized into an intermediate data set, called the integrated process-level execution log. The IPEL contains only the key values and control data to identify the events and is constructed in such a way that some higher level logical views can be derived easily.

As shown in Figure 8, the IPEL consists of two related data structures called the process creation table and the event chain. The PCT presents the processes and their relations. Each entry of the PCT, corresponding to a process in the system, gives the process identification, its parent process identification, and the time the process was created. The event chain consists of a single-linked list of process-level events that describes what has happened in the system. Each node of the event chain that corresponds to an event contains the key values of the event (as defined in the "Process-level monitoring" section) and two pointers. One pointer is used to link all the events in time sequence (time link) so the logical views related to multiple processes can be derived by searching the events on the time link. The other pointer links all the events related to one process in time sequence, called the process link, such that the logical views concerning a particular process can be derived by searching the events on the process link.

Here, we present a simple example to illustrate the postprocessed execution information. The program shown in Figure 9 is monitored on a Unix operating system.

We assume that the monitoring starts at the beginning of the execution of the program. Figure 10 shows the intermediate data set after postprocessing the recorded information.

Applications

Performance evaluation, testing, and debugging are common tasks in developing and maintaining real-time distributed computing systems. These tasks require detailed insight into a program's execution behavior, which is not possible to comprehend by simply examining the static aspects of the program. Conventional monitoring facilities provide tools to monitor system behavior and collect execution data to support testing, debugging, and measuring system performance. However, due to the invasive nature of monitoring, conventional approaches are inadequate for monitoring real-time distributed computing systems. In this section, we describe how our approach to monitoring system behavior can be used to support the testing and debugging of real-time distributed computing systems.

In demonstrating the presence of program errors or the localization and correction of erroneous program code, it is necessary to repeatedly examine the execution

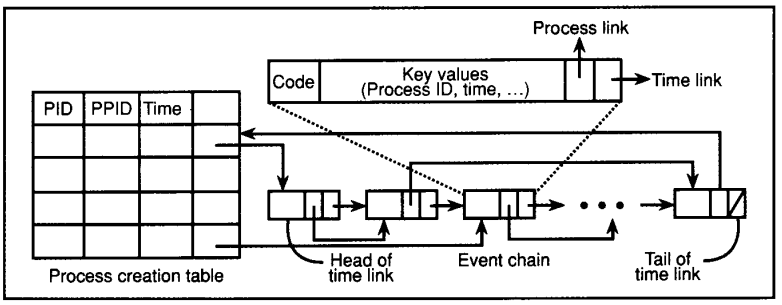


Figure 8. Logical view of the integrated process-level execution log (IPEL).

```
main()
{ int  pid;
  int  code = 0;

  pid = fork();          /* create a new process */
  if (pid == 0)
    code++;              /* segment of new process */
  else {                 /* segment of parent process */
    wait();              /* wait for the termination of the child process */
    pid = fork();        /* create another new process */
    if (pid == 0)
      code++;            /* segment of new process */
    else                 /* segment of parent process */
      wait();            /* wait for the termination of its child */
    }
}
```

Figure 9. An example program.

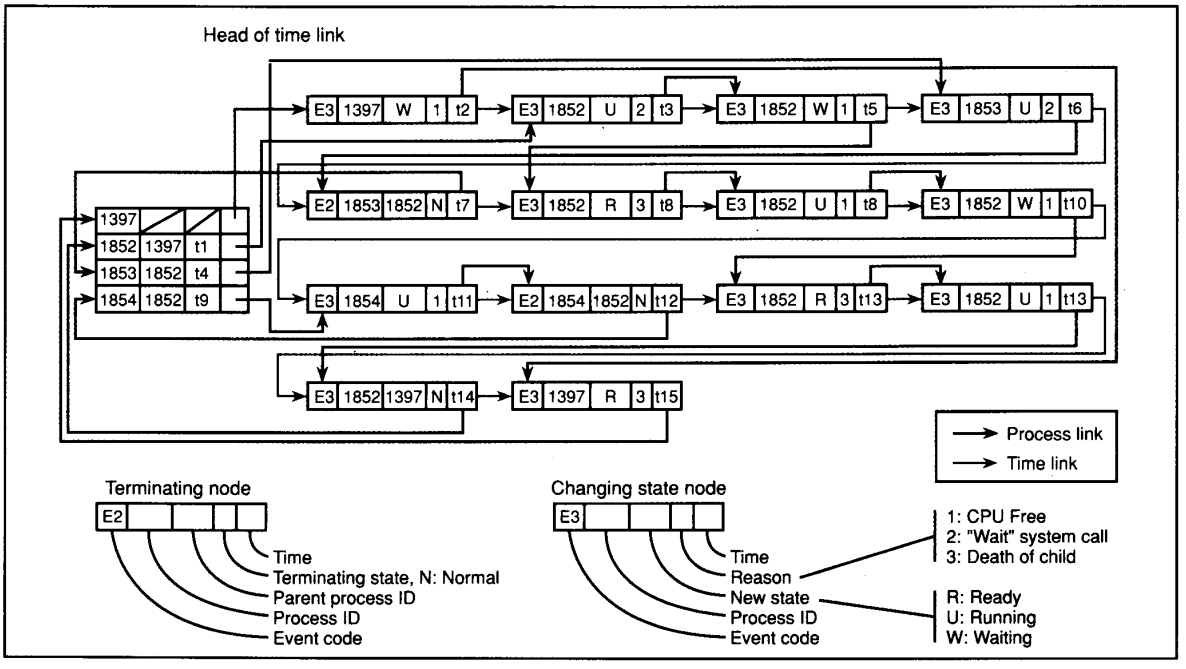


Figure 10. An example of the integrated process-level execution log (IPEL).

behavior of a program. This requirement makes monitoring program execution essential. To simplify the description of the monitoring procedure, we adopted a top-down approach to testing and debugging. (Our monitoring architecture supports a bottom-up approach as well.)

Initially, the master node is monitored, and its execution behavior is recorded to identify erroneous processes in each node. The execution behavior of the master node contains all interactions among distributed nodes and will be compared by the user against the given description of the whole system behavior. When the processes that reside in the master node are believed to function correctly and the suspect slave nodes are identified, you can test and debug the suspect slave node in similar fashion. The same testing and debugging procedure is applied to suspect nodes one by one until the erroneous processes are detected. The erroneous processes will then be examined thoroughly by applying the monitoring system to these processes, if necessary.

Generally, the testing and debugging strategy for a real-time distributed computing system is target-system dependent. Based on the assumptions given in the section entitled "A model of real-time distributed computing systems," we can apply the monitoring system to facilitate testing and debugging real-time distributed systems as follows:

(1) monitor the master node (and collecting trace) by using the noninvasive monitoring system,
(2) identify suspected faulty slave nodes and erroneous processes residing in the master node by examining the collected traces,
(3) monitor suspected slave nodes, if any,
(4) identify erroneous processes by examining the traces,
(5) repeat steps 3 and 4 until no more suspected nodes are left, and
(6) examine erroneous processes one by one to identify bugs by using conventional debugging tools (if a user wants to focus on a suspected process whose details had been skipped).

Through the development of real-time distributed computing systems, it has been found that many bugs do not surface until late in the system's life cycle and that they are difficult to debug. Among such persistent errors, synchronization and timing errors are the most difficult to detect. The following examples show how synchronization and timing errors can be detected by using our approach. In this illustration, we assume that the faulty node is identified and the user is trying to locate the erroneous process(es) on this node.

	Process P1	Process P2	Semaphore X
1		V	1
2	P		0
3	V		1
4	P		0
5		P	−1
6	V		0
7		V	1
8		P	0
9	P		−1
.	.	.	.
.	.	.	.
.	.	.	.

Figure 11. Interprocess synchronization via P/V operations.

Synchronization errors. As an example of debugging synchronization errors, let us consider the process synchronization performed between a pair of processes, P1 and P2, residing in the same node. We assume that communications between these processes are performed via P and V operations on a shared semaphore *X*.

In the postprocessing phase, the recorded execution history, which contains accesses to a shared semaphore X that characterizes the process synchronization of the target node, is organized to form an event chain. The events of interest, the synchronization primitives in this case, are derived from the execution history. Meanwhile, the information concerning all computation entities (that is, processes, shared objects, etc.) are also identified.

A typical sequence of interactions between processes is shown in Figure 11. By enforcing the P/V operation sequence to operate on the shared objects, we can add output statements to an erroneous program that provides additional details about the program behavior while preserving the synchronization behavior. This is not possible with conventional debugging tools because the output statements can violate the timing constraints of a real-time program and yield a different synchronization sequence.

Based on examination of the execution history, we assume the suspected synchronization error is located in process P2. Based on the information from examining the execution history, the user can identify bugs, if any, within a process by using the replay mechanism.[10] In the error isolation and correction step, the user can choose to examine the execution history of process P2 or, if necessary, choose to repeat execution of the target system by enforcing the P/V operation sequence to perform on the shared object *X*. During the program replay, we allow each process to reproduce its behavior over and over, thus providing opportunities for closer examinations of the process behavior. Any results that may have been ignored during previous observations can always be reproduced on demand for closer examinations.

Timing errors. Depending on the application and environment, timing constraints imposed on a real-time distributed system vary widely. Dasarathy gave a classification of timing constraints for a real-time system.[11] In general, there are two categories of timing constraints:

(1) performance constraints that set limits on the response time of a system and
(2) behavior constraints that make demands on the rates at which users apply stimuli to the system.

For a telephone switching system, for instance, the typical temporal restrictions are: "After the first digit has been dialed, the second digit shall be dialed no more than 20 seconds later," or "The caller will receive a dial tone no later than two seconds after lifting the phone receiver." To simplify the example, we represent the interesting events in a target system by the abstract identifications. For example, in case of a telephone switching system, E1 stands for lifting the phone receiver, E2 for sending a ring-back tone to the caller, etc.

Let us suppose that two events, E2 and E3, should occur within 60 seconds after event E1, that E3 should be delayed by at least 15 seconds after E2, and that it should occur within 30 seconds after E2. Figure 12 gives the graphic representation of these requirements. In the monitoring phase, the time at which each event happened is recorded. Given this information, we can easily examine the execution history against timing constraint requirements. If

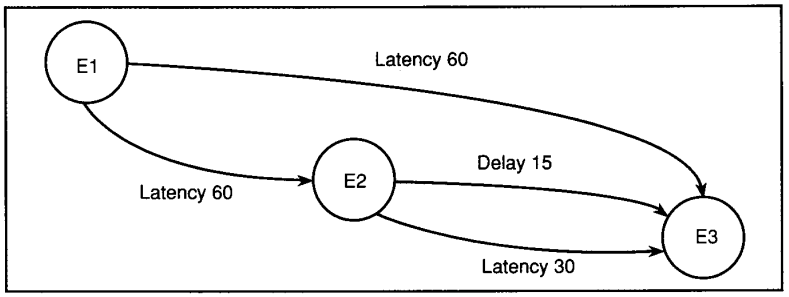


Figure 12. An example of timing requirement specification.

timing violations are found, the replay mechanism can be used to reexamine the program behavior and isolate the errors.

In this article, we have presented a noninvasive hardware architecture for monitoring real-time distributed computing systems. The noninvasive hardware module for monitoring is being used to develop a testing and debugging system for real-time distributed computing systems. Our work has focused on the monitoring of process-level events.

Since there is no need to modify the target distributed computing system, our approach guarantees the preservation of timing constraints on real-time distributed computing systems. Without interference in the execution of the target system, the monitoring system will not change system behavior and performance. A second advantage is due to the flexibility of the programmable QCU, which can be set by the user to monitor different classes of events.

For future research, the monitoring system will be used to support different abstraction-level monitoring for different application purposes, such as the performance evaluation, testing, and debugging of real-time distributed computing systems.■

Acknowledgments

We are grateful to Y.D. Bi, Tom Moher, and the anonymous referees for their comments on earlier versions of this article. This work is supported in part by the National Science Foundation under Grants CCR-8809381 and CCR-8708771 and an award from the Institute of Electrical and Electronics Engineers and the Engineering Foundation Society.

References

1. H. Garcia-Molina, F. Germano, Jr., and W.H. Kohler, "Debugging a Distributed Computing System," *IEEE Trans. Software Engineering*, Vol. SE-10, No. 2, Mar. 1984, pp. 210-219.
2. D. Haban, "DTM — A Method for Testing Distributed Systems," *Proc. 6th Symp. Reliability in Distributed Software and Database Systems*, CS Press, Los Alamitos, Calif., Order No. 737, 1987, pp. 66-73.
3. J. Joyce et al., "Monitoring Distributed Systems," *ACM Trans. Computer Systems*, Vol. 5, No. 2, May 1987, pp. 121-150.
4. R.J. LeBlanc and A.D. Robbins, "Event-Driven Monitoring of Distributed Programs," *Proc. 5th Int'l Conf. Distributed Computer Systems*, May 1985, pp. 515-522.
5. B. Plattner and J. Nievergelt, "Monitoring Program Execution: A Survey," *Computer*, Vol. 13, No. 11, Nov. 1981, pp. 76-93.
6. B. Plattner, "Real-Time Execution Monitoring," *IEEE Trans. Software Engineering*, Vol. SE-10. No. 6, Nov. 1984, pp. 756-764.
7. R. Snodgrass, "A Relational Approach to Monitoring Complex Systems," *ACM Trans. Computer Systems*, Vol. 6, No. 2, May 1988, pp. 157-196.
8. H. Tokuda, M. Kotera, and C.W. Mercer, "A Real-Time Monitor for a Distributed Real-Time Operating System," *ACM SIGPlan Notices*, Vol. 24, No. 1, Jan. 1989, pp. 68-77.
9. M. Maekawa, A.E. Oldehoeft, and R.R. Oldehoeft, *Operating Systems: Advanced Concepts*, Benjamin/Cummings Publishing Co., 1987.
10. J.J.-P. Tsai, K.Y. Fang, and H.Y. Chen, "A Replay Mechanism for Noninterference Real-Time Software Testing and Debugging," *Proc. Conf. Software Maintenance*, CS Press, Los Alamitos, Calif., Order No. 1965, Oct. 1989, pp. 209-218.
11. B. Dasarathy, "Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them," *IEEE Trans. Software Engineering*, Vol. SE-11, No. 1, Jan. 1985, pp. 80-86.

Jeffrey J.P. Tsai is an assistant professor of computer science in the Department of Electrical Engineering and Computer Science at the University of Illinois at Chicago. His research interests are in requirements specification languages, real-time software testing and debugging, artificial intelligence, and software engineering.

Tsai received the PhD degree in computer science from Northwestern University, Evanston, Illinois. He is a member of the IEEE, the IEEE Computer Society, ACM, the American Association of Artificial Intelligence, and Upsilon Pi Epsilon.

Jeff Kwang-Ya Fang is founder and president of Utek, Inc., a Chicago-area software development and consulting firm, and an adjunct assistant professor in the Department of Electrical Engineering and Computer Science at the University of Illinois at Chicago. His research interests are in logic design, real-time systems, and software testing.

Fang received his MS and PhD degrees in computer science from the Illinois Institute of Technology in 1976 and 1982, respectively. He is a member of the IEEE Computer Society.

Horng-Yuan Chen is a PhD student in the Department of Electrical Engineering and Computer Science, University of Illinois at Chicago. His research interests include software testing and debugging, distributed computing systems, and expert systems, particularly in the testing and debugging of real-time software systems.

Chen received the BS degree in mineral and petroleum engineering from the National Cheng-Kung University, Taiwan, in 1980, and the MS degree in computer science from the University of Illinois at Chicago in 1986. He is student member of the IEEE Computer Society.

The authors can be contacted at the Department of Electrical Engineering and Computer Science, University of Illinois at Chicago, Chicago, IL 60680.