MONITORING AND MANAGEMENT-SUPPORT
OF DISTRIBUTED SYSTEMS

By

Dieter Haban, Dieter Wybranietz† and Amnon Barak*

TR-88-007

November 1988

ABSTRACT

This paper describes a tool for on-line monitoring of distributed systems. The tool consists of a hardware component and software level, i.e., a hybrid monitor, which is capable of presenting the interactive user and the local operating system with a high-level information and performance evaluation of the activities in the host system with minimal interferences. A special hardware support, which consists of a test and measurement processor (TMP), was designed and has been implemented in the nodes of an experimental multicomputer system. The main function of the TMP is to execute software for monitoring the local system behavior and to measure the performance of both the resident operating system and the application software. The TMP can also be used to execute low level operating system functions, to manage local resources and to trigger time driven events in order to reduce the overhead of the host operating system. The operations of the TMP are completely transparent to the users with a minimal, less than 0.1%, overhead to the hardware system. In the experimental system, all the TMPs were connected with a central monitoring station, using an independent communication network, in order to provide a global view of the monitored system. The central monitoring station displays the resulting information in easy-to-read charts and graphs. Our experience with the TMP shows that it promotes an improved understanding of run-time behavior and performance measurements, to derive qualitative and quantitative assessments of distributed systems.

† Department of Computer Science, University of Kaiserslautern, W. Germany.
* On leave from The Hebrew University of Jerusalem, Israel.
This work was supported in part by a grant from the International Computer Science Institute, Berkeley, CA 94704.
1. INTRODUCTION

Distributed systems have desirable capabilities such as improved reliability and availability, ease of modularity, incremental growth, configuration flexibility, and high performance. New technologies, increased research, and demand for improved performance, communication, and productivity yield new architectures and commercial systems. A few examples are the Hybercube [18], the Butterfly [4] and the Transputer [8]. Research in distributed operating systems and adequate programming languages are also under way [15, 20]. Despite these increased activities, there is a substantial lack of development of methods and tools for monitoring and measuring distributed systems which aid in verifying the above features, assisting in detecting and locating abnormal behavior such as performance bottlenecks, and exploiting new functions that can be performed by the monitoring sub-system.

In this paper, we describe an experimental hybrid monitoring system that was built as part of the INCAS [16] multicomputer project. Hybrid monitoring means a dedicated hardware device (attached processors) that can extract data from the host machine and can then execute software to process and evaluate that data, concurrently and independently of the host system. The specific monitoring system that we present is designed to perform three main functions. First, it monitors the host machine in a transparent manner, with minimal side effects. This function is used primarily to extract raw data. Second, the monitoring system can measure the behavior of the software in the host system by tracing its execution, evaluating its performance, and providing insights about this execution to the interactive user and the resident operating system. The third function that the monitoring system can perform is the execution of several routine tasks to assist the resident operating system in managing the increased number of operations which results from the distributed environment.

Traditionally, monitoring and measurement have been fundamental techniques in hardware engineering. The increased complexity of present computer systems necessitates such tools for
software development as well. In most existing computers, the hardware and software systems are not designed to be monitored, although monitoring tools in the form of stand-alone hardware devices, programs, and hybrid tools have been available for many years. Measurement and monitoring become even more difficult in distributed systems, since these systems feature asynchronous concurrent activities, nondeterministic and unrepeatable behavior and communication among processes that introduces unpredictable delays. The measurement task is further complicated due to a lack of central control, precise global time and accurate global state, since measurements have to be performed simultaneously in different nodes and the results must be collected and evaluated in some reasonable form, preferably in one location, in order to provide users with a meaningful view of their software.

Monitoring tools can be classified into pure hardware, pure software and hybrid monitors. A hardware monitor is a device that is not a part of the monitored system. Although such devices can be designed to have minimal or no effect on the host system, they generally provide only limited, low-level information about the activities of the host system. In contrast, software monitors can present information in an application-oriented manner. These monitors are usually contained within the measured system, sharing with it the same execution environment, thus producing some degree of interference in both the timing and space of the monitored program. Therefore, pure software monitors are not adequate for on-line monitoring and for real-time measurements during the execution time.

Hybrid tools [7,19] can be designed to benefit from the advantages of both hardware and software monitors with minimal effects on the monitored system. Such monitors typically consist of an independent hardware (device) which can perform the low level monitoring, i.e. information gathering, and a software level which is executed by this device capable of measuring, evaluating and displaying the performance of the host system. The monitoring system described in this paper is a real-time hybrid monitor which, in addition to the above mentioned function can also interact with the host system. One feature of this monitor is the small amount of its
interference with the application level. Another feature is the ability of several monitoring devices placed in different nodes of a multicomputer system to cooperate by communicating with each other.

The ability to combine on-line information and display execution patterns creates a powerful tool for evaluation and management of programs which are executed concurrently on several processors. The monitoring tool described in this paper allows the user to perform on-line observations of communication patterns, to detect bottlenecks and inactive processes, assistance in debugging and in general, they provide a multi-level view of the application and zooming capability with several degrees of granularity. The monitoring system can also combine local monitored data with remote information and knowledge about other nodes, then evaluate this information and make it available to the operating system of each node, thus supporting it in managing its tasks. Examples of such tasks are load balancing, locating resources and orphan management. These tasks become important in a system with a large number of nodes where efficient management requires knowledge the entire system or parts thereof and frequent evaluation of local and remote parameters.

In the next section, we specify the objectives which guided the design of the TMP monitoring system. Sections 3 and 4 describe the TMP software and hardware components. In section 5 we discuss means to improve the users' understanding of their distributed applications using the TMP concept. Section 6 demonstrates how the TMP system can be used for management support at the operating system level.

2. DESIGN OBJECTIVES

Monitoring program execution may require a significant amount of an application programmer's efforts, since the insights it provides cannot be replaced by those obtained from static analysis of program texts [17]. In addition, monitoring activities can be used to support the
operating system’s tasks. The role of on-line monitors becomes even more important in a multiprocessor environment when many threads of a program may execute concurrently. The design objectives which have been accomplished in the TMP monitoring system are aimed to assist the interactive user to gain better knowledge about the run-time behavior of their applications and the host operating system in the performance of its low level tasks. The specific objectives are:

- **Interference**: the interface between the monitoring device and the host system is transparent, it does not change the system behavior and has minimal effects on its performance.

- **Continuous Monitoring During Operation**: the monitoring sub-system can trace the host system, evaluate and supervise the execution of certain applications, and display information in real-time about their progress. We note that this service is particularly important when monitoring long-running (concurrent) processes, complex distributed programs with dynamically changing structures and when considering real-time programs that control critical systems such as a nuclear power plant or an airborne system.

- **Application-oriented Presentation**: the monitoring and measurement tool provides a user-friendly, graphic interface which can be used by application programmers and not only by experts. This means that, for observation purposes, the large amount of monitor and performance data is interpreted, evaluated, and presented in an application-oriented manner which reflects the semantics and organization of the application programs. In addition, the processing, evaluation and display of this vast amount of monitored data does not cause any overhead in time and space.

- **Preserve Data for Off-line Analysis**: the information gathered by the monitoring system can be recorded to provide more detailed measurement results about past execution and for further, off-line analysis.

- **Integration and Ease-of-Use**: the instrumentation of the monitoring system is incorporated into the hardware system during its design phase, resulting in an integrated approach. This
feature allows the users to take advantage of these tools during the development, debugging and execution of their applications. Since this monitoring scheme is transparent to the application level and requires minimal overhead, this approach is better than temporary instrumentations which change the system behavior upon removal.

- **Flexible Hardware and Software**: the monitoring hardware and software can easily be customized to various environments and application requirements. The tools are not tailored to one specific hardware or application. During operation the monitoring hardware and software can be interactively adapted to changing needs.

- **Feed-Back**: in order to allow the monitored system to dynamically respond to the results of the measurement, the monitoring sub-system is capable of funnelling its results back to the monitored system. This provides the host system with up-to-date information about its own activities, and can be used for fine performance tuning.

- **Operating system management support**: the monitoring system can execute routine, low level, operating system tasks, such as the local load or network management, thus reducing the overhead of the host system.

### 3. THE TEST AND MEASUREMENT PROCESSOR

In this section, we present the concept behind the Test and Measurement Processor (TMP). In the next section, we describe its hardware realization.

#### 3.1. The TMP concept

The main goal of the TMP is to perform efficient monitoring of the application software. This goal is accomplished by using events generated by the application software; then these events are categorized, time-stamped, processed, and displayed by the TMP hardware. By using semantic information about the monitored programs provided by the compiler, the monitoring software is able to present evaluated data in an application-oriented manner. The TMP is also
capable of executing local software for various processing and evaluation needs for its host. In an environment that consists of several nodes, each TMP can receive data from any other TMP over a network; symmetrically, each TMP is able to send data to other TMPs. Ideally, the TMPs communicate via their own network, thereby avoiding disturbances to the application level. However, the communication among the TMPs can also be accomplished by the communication facility of the host system. In Figure 1 we illustrate the principles of the TMP.

![Diagram of TMP principles]

Figure 1: Principles of the TMP

A key point of the measurement procedure is the identification of the type of events which should be monitored. Then the problem is how to insert these events into the monitored software. In the current study, we identified events which represent significant trends in the behavior of the system, e.g., process creation and deletion. The triggering points for these events are placed in the operating system kernel which then provides continuous information about the system behavior. We note that additional triggering points may also be inserted at the application level. Initial experiments with the INCAS [16] multicomputer system, showed that, typically, 600-800 such events were generated each second on each node. We note that the host overhead caused by the
TMP is lower than 0.1%, thus it becomes a permanent part of each node.

3.2. Instrumentation of the host system

An event is defined as a special condition that occurs during the normal system activity, such that it can be made visible to the TMP. As such, events represent changes in the behavior of the system. There are two kinds of events, standard and optional. Standard events are permanent and integral parts of a system. They are triggered by the kernel software, the communication sub-systems, and the supporting software. Standard events are intended to support monitoring and measuring during the normal system execution. Note that since standard events are an integral part of the operating system, monitored software need not be recompiled or relinked. Optional events are associated with the application programs. They are generated by the compilers or are placed manually into program code. Optional events serve for debugging purposes.

The instrumentation of standard events is an integral part of the operating system. In this case the minimal monitored activities should include the dispatcher, the kernel activities and the communication sub-system. Note that several degrees of desired details can be obtained. In a multicomputer system with a dynamic structure, additional events have been added to reflect the increased number of activities. Examples of standard events for a distributed system are listed in Table 1. For each event, we list the event class, followed by a list of parameters.

<table>
<thead>
<tr>
<th>Dispatcher Events</th>
<th>Communication System Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>start process &lt;procID&gt; &lt;procNo&gt;</td>
<td>receive message &lt;portID&gt;</td>
</tr>
<tr>
<td>stop process &lt;procID&gt;</td>
<td>send message &lt;portID&gt;</td>
</tr>
<tr>
<td>assign process &lt;procID&gt;</td>
<td></td>
</tr>
<tr>
<td>resign process &lt;procID&gt;</td>
<td></td>
</tr>
<tr>
<td>ready process &lt;procID&gt;</td>
<td></td>
</tr>
<tr>
<td>block process &lt;queueID&gt;</td>
<td></td>
</tr>
<tr>
<td>migrate process &lt;procID&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Kernel Events</strong></td>
</tr>
<tr>
<td></td>
<td>I/O init &lt;queueID&gt;</td>
</tr>
<tr>
<td></td>
<td>wait queue &lt;queueID&gt;</td>
</tr>
<tr>
<td></td>
<td>send queue &lt;queueID&gt;</td>
</tr>
<tr>
<td></td>
<td>receive queue &lt;queueID&gt;</td>
</tr>
</tbody>
</table>

Table 1: Examples of standard events
Dispatcher events trace the operations of the operating system dispatcher. The general model of a dispatcher is depicted in Figure 2. It includes events which reflect the following states of a process: ready, blocked, running, migrated, killed. The parameter procID (see Table 1) identifies the process object, and the parameter procNo identifies its type. Kernels events describe low-level activities of the operating system. These include initialization of queues for I/O devices, communication sub-system and synchronization mechanisms.

![Diagram showing the states of a process](image)

Figure 2: Events representing operations of the dispatcher

Communication events allow insights into the activities of the communication sub-system activities, including inter-process communications. The minimal types of operations that are monitored include sending and receiving of messages. The corresponding operations are send message <portID> and receive message <portID> events.

3.3. Processing of local events

The evaluation of global events is performed in two stages. First, the TMP software processes incoming events at each host, then it transfers condensed data to a central station. The second stage is done at the central monitoring station which combines all the obtained results to provide the user with a global view of his applications (this is further explained in a section below). We note that the central station provides a consistent global state by ensuring the causality relationship [12] among the locally collected activities. Another algorithm incorporated into
the monitoring system ensures that the received measurement data are collected at close time intervals by synchronizing the length and the overlapping of these intervals at the TMPs, as suggested by [9].

The monitoring software in each TMP is responsible for collecting data about the behavior of its host. Upon system startup, the monitoring software keeps track of initialization of waiting queues and creates a table of wait conditions (see Table 1). The entries of this table, which are accessible via a hash function over the queueID, contain necessary semantic data and evaluated execution summaries such as the name of the wait condition, average queue length, and maximum queue length.

To demonstrate a typical sequence of events and the corresponding TMP activities, we now follow the operations of the dispatcher. Initially, the event start process <procNo> <procID> informs the TMP that a process of a type <procNo> has been created and placed in the ready queue. Upon occurrence of the event assign process <procID>, the monitoring software is informed that the process has been activated (running); thus all the subsequent events result from the activities of that process until the next assign process event. Since each event is stored with a local time stamp, the TMP software can measure elapsed times between activities accurately. For example, the time difference between assign process and resign process or block process determines the elapsed time of a process. The time difference between block process and ready process determines the blocked time of a process. Cumulative execution and blocked times are stored in information tables for each process. In particular, for the idle loop, the idle time is accumulated. Due to the vast amount of accumulated information, execution times of procedures are kept only during debugging. Upon receipt of an event block process <queueID>, the length of the corresponding queue is incremented. In this case the parameter <queueID> enables the TMP to identify the wait condition and to measure the duration which the process spent in the blocked queues. Finally, upon receipt of event ready process <procID>, the corresponding length of the waiting queue is decremented.
The communication system events send message and receive message enable the TMP to keep track of the number of messages each process exchanges with other processes. Additional communication events include failed messages, retransmissions and timeout for send/receive operations. We note that in each case the size of the message is also measured.

Based on the above monitoring, the TMP software measures the host machine load, including the CPU utilization, the idle time, length of queues, times that processes spent in the queues, rate of I/O and the communication traffic. This list can be further extended pending the desired degree and the purpose of the measurement. We note that additional evaluation algorithms and new events can easily be integrated into the current software.

3.4. Event generation

The mechanism for generating an execution-time event consists of an instruction that is inserted at a specific, well-chosen point in the application code. Each event is marked by one store instruction which writes through the local processor cache. Therefore, each event is immediately visible on the system bus. The format of the instruction is:

STORE ADDR, VALUE

Each address represents one event class. In the current implementation, there are 256 different event classes, and therefore the range of the address field is bound to 256 addresses. Examples of event classes are:

\begin{verbatim}
  start process
  block process
\end{verbatim}

The VALUE of the store instruction serves as a parameter which specifies one event within each class. For example:

\begin{verbatim}
  start process  <processID>
  block process  <queueID>
\end{verbatim}
In some cases, a standard event may require two parameters. For example, the event \textit{start process} <\textit{procNo}> <\textit{procID}> is triggered by two instructions, one for each parameter. In this case, the operating system kernel ensures that these events are treated as one atomic operation.

In the current implementation, the TMP has 32 standard and 26 optional monitoring events. Most of the standard events are related to the initialization procedure and the startup configuration. Thus, upon system initialization, there is a burst of standard events with information about queues, process creations, establishment of communication channels, etc. During actual operation of the system, the majority of the standard events are related to the dispatcher and communication sub-system activities.

4. THE ARCHITECTURE

The TMP is a hardware device that was designed to be an integral part of each node in a multicomputer system. Typically, such a node consists of one or more processors, memory, I/O and communication devices. The TMP may be viewed as an additional device responsible for monitoring, recording, and evaluating the activities of the host node as well as its communication activities. The TMP can send its findings to the host operating system, to other TMPs and to a central monitoring station. The central station, which may be connected to all the TMPs, is used for interactive monitoring, global measurements, and distributed debugging. To further reduce the interference of the TMPs with the monitored system, all communication among the TMPs is done via a separate network. The scheme of the TMP architecture is given in Figure 3.
4.1. Hardware implementation

The design of the TMP hardware was aimed to allow implementation in any computer system. It is connected to a system bus and monitors events on this bus with negligible impact on the measured system. Figure 4 shows the TMP hardware and its integration into a node of the INCAS system [16]. The bus interface is a separate module which does not effect the other parts of the TMP. This module can be developed for use in any bus architecture.

The specific parts of the TMP hardware consist of a M68000 based processor with 1 MByte of local memory, a dual RS232 port for local interactions, a network interface to other TMPs, and an Event Processing Unit (EPU). The processor is used for the execution of monitoring software. This includes low-level monitoring and evaluation routines which are resident in the memory. Higher-level evaluation software as well as operating system functions can be loaded onto the TMP's memory. This software makes use of the preprocessed data condensed by the lower-level monitoring software. In the current implementation, this software can collect and process up to 10,000 events per second.
Figure 4: The TMP Hardware

The EPU consists of a local event buffer, a comparator, a clock and an overflow counter. The local event buffer of the EPU is used as a FIFO for collecting sequences of events. The depth of the FIFO is 16 entries. This depth was tuned to cope with a high arrival rate of events, and is based on an experience gained with an earlier prototype. Each entry in the event buffer consists of 80 bits: 8 for the event class; 32 for the event parameter; 36 for the time stamp (in μs); and 4 bits for control (CPU mode, overflow marker).
The comparator of the EPU is responsible for checking the addresses on the host bus. If an address falls within the range which represents event classes, the matched address and the next data on the bus are stored in the event buffer, along with the local time. The location of the address range can be adapted to the actual hardware environment by a switch. The last byte of the address determines the event class; thus it is the only byte which is stored by the EPU. The low level implementation of the TMP must ensure that this address range does not interfere with the main memory. Another function which is stored by the EPU is the processing mode (supervisor or user) of the host system. This information can be used to approximate the time spent for user or kernel modes.

The timer of the EPU is tuned to measure the time difference between events. The rate of this timer guarantees that no two successive events will have the same time. For example, in the current implementation, a time quantum of 1\(\mu\)s is used, thereby allowing up to 19 hours of measurements before the counter overflows. The overflow counter of the EPU is used to count the number of events lost due to a buffer overflow. Although the specific type of each event is lost, the TMP software is aware of this inconsistency. In order to detect the occurrence of an overflow, the first event placed in the buffer after such an overflow is marked in its control field. The contents of the overflow counter can be accessed by the TMP software using the read overflow counter and reset overflow counter commands.

Other commands for controlling the EPU include enable EPU, disable EPU, reset EPU, set time slice, set timer, reset timer, mask events, and read next event. For example, the commands enable EPU, disable EPU are used to enable or disable the collection of events by the EPU. Similarly, the command mask event is used to disregard events which are not relevant to a given application. Finally, we note that the TMP has access to the memory of the host processor. This property is used for debugging activities which can run on the TMP and for redirecting information gathered by the TMP to the host operating system.
It is possible to insert more than one TMP into one node without disturbing each other. Thus, each TMP may run different event processing software concurrently to perform different analysis tasks. The TMPs may also run the same event processing software, but each TMP is responsible for different event classes, resulting in their sharing the work. In both cases, the monitoring instrumentation and the application processes need no further preparation. In addition, the collection of data done by the TMP might include hardware monitoring, such as measuring the bus and I/O devices load. These measurements do not cause any interference with the operation of these hardware components and do not need any additional instrumentation.

5. IMPROVING THE UNDERSTANDING OF DISTRIBUTED PROCESSING

In this section we show how the monitoring sub-system can be used to help users to manage and actually improve their understanding of the run-time patterns of their applications, and the system programmers to gain insights about the performance of the operating system level. A discussion of further operating system management support by the monitoring sub-system is given in the next section. In the sequel, we refer only to the application level users, although the benefits to the system programmer are obvious.

Distributed programs may be viewed as a set of modules (processes), where each module may consist of other modules or processes. During the execution time these modules are dispersed among the nodes and are executed concurrently. Our goal is then to provide the users with sufficient information for observing the run-time behavior of their modules. Our basic assumption is that by these observations the users can gain better knowledge about execution patterns of the application software. They can then perform further analysis, perform more measurements and tune the execution patterns. We note that the monitoring system cannot automatically improve the performance of the application level, it can only assist the application level user to do so.
The information which is collected by the monitoring sub-system includes the following: process creation and deletion, context switches; process execution times, process blocked and ready times; performance of I/O operations, including sending and receiving messages; total communication volumes and establishments of new links; length of queues; machine load and idle times; counts of events, etc. Evaluation of this information, with several degrees of granularity, can then be used for process profiling and measurements, debugging, animation, identification of bottlenecks, zooming on a particular activity and for load distribution.

5.1. Visualization of Program Behavior

From the user point of view, the monitoring system consists of several layers. The lowest level is that of the individual TMPs which are placed in the nodes of the system for information gathering. After an initial filtering, this information is sent to the central monitoring station. The second level is a rule-based evaluation module capable of performing specific operations based on the information received, including generic operations, i.e., a sequence of pre-specified operations. Specific examples are given later in this section. The third part is the display monitor and the user’s point of interaction module. The display module, like the evaluation module, is a rule-based software which is responsible for displaying the results of the evaluation module on the screen. It uses a set of symbols and control tables, so that for each type of information, an appropriate display icon is used, i.e., graphs, charts, etc. The interaction module is responsible for interpreting the user commands and then relaying the appropriate control commands to the other layers.

The first step in the monitoring process is to gain knowledge about the internal structure of the application program and its semantics. This is accomplished by using information which is extracted during the compilation of the monitored program. This information includes the internal organization of the program, communication links, names and types of the program sub-modules, and how these components are linked to each other. This information is compiled when
a program is load-linked and is then stored in the program specific data base of the central monitoring station. Then, during the execution of the monitored program, this data base is updated continuously by the evaluation module, following the dynamic changes of the program modules, along with additional execution summaries obtained.

We now describe a simple example to illustrate how the monitoring system is used. Consider a program which consists of four interacting modules, see Figure 5 for details. Our first goal is to display a general overview of these modules, then to add the communication links (and volumes), and finally the performance of each module. Using a menu-driven interface, the user initially requests a display of all his program modules. At this time the user also specifies his preferred mode of display, e.g., graphs, icons, colors. Note that this is stored in the user's display data base for future use. After processing the first request, the central monitoring station obtains the names, types and locations of each program module from the program specific data base. Note that the TMPs provide information about all their respective activities continuously, thus the central station needs only to identify and filter the requested information. It then displays these modules on the screen using the display mode specified by the user's display data base. Next, the user requests the addition of the communication activities. This is done by the menu on the display module, which then activates the corresponding TMPs to transfer relevant information to the central station. At this time the communication lines will appear on the screen, including the total number of messages exchanged. We note that most of the work is done at the local TMP in each node and that only specific requested information for display is passed to the central station. Finally, upon another request by the user, the performance activity of each module is transferred by the TMPs to the central station, which then draws the bar graphs displaying CPU utilization, I/O rate, communication usage and accumulated time in the ready queue.
In the remainder of this section, we give a brief overview of some specific areas that may benefit by using our monitoring tools.

5.2. Communication

As a result of the measurements that the TMP performs, it is possible to gain knowledge about each process' communication volume and targets, starting from such low-level information about the message sizes, rate of transmissions, time spent in performing communication tasks,
establishment of links, IPC, communication buffer usage, rate of retransmissions and network failures. It is also possible to obtain knowledge about the system-wide communication activities or any subset of machines or processes.

It is evident that, normally, the user does not need all these details, therefore a ruled-based menu is provided to select specific requests. This menu also allows the definition of threshold functions which then trigger display of irregular communication patterns, i.e., stray messages or overloaded links. Note that in this last case, an average function is used by the evaluation module.

5.3. Synchronization, bottlenecks and deadlocks

From the monitoring point of view, synchronization problems, bottlenecks and deadlocks are states to which some processes lend themselves and are thus either saturated or not executing at all. Although these processes may look like any other processes in the system, nevertheless they may cause performance degradation or even system failure. Unfortunately, few mechanisms allow transparent detection of such cases, and thus the user's involvement is critical. The monitoring system can help in locating these kinds of irregularities, then allow further zooming on each cause separately, thus simplifying the evaluation process.

Specific states that trigger this part of the monitoring system includes: a prolonged idle or blocked process; improper clustering of sets of processes, i.e., many waiting for one; overloaded process due to improper allocation of communication buffers; improper balance between internal mechanisms, e.g., the speed of the communication controller and the software which interact with it. In such cases, heuristic threshold functions given by the user may assist in detecting such abnormalities.
5.4. Zooming through the program structure

Based on the evaluated measurement data, the user may interactively zoom through the hierarchical program structure and focus on parts of the programs, a single machine, a single processes, or even a sub-module. At each of these levels, it is possible to obtain measurements about the monitored modules displayed in charts and graphs [21]. Another service provided by the monitoring system is on-line information gathering for off-line analysis.

5.5. Debugging

The monitoring tool is a natural aid in the debugging process. To begin with, the user is informed about the state of execution of each process: failed, blocked or starved. Error detection using the TMP becomes a simpler job with the possibility of inserting optional debugging events to the programs. When executing such a program, and particularly when the program includes many threads, the centralized display of the debugging information is the only adequate source of on-line information [10].

5.6. Load balancing

Load balancing is a simple byproduct of the monitoring system. For proper performance, this algorithm needs to know the execution profile of each process, its I/O and communication requirements, as well as the combined load of all the processes in each node. For systems with static or user controlled load balancing, the monitoring system can provide this information, on-line and in real-time. It can also provide the user and the system manager with information about irregularities in the distribution of the load. In integrated systems, where load balancing is done automatically, by the operating system [2,3], the monitoring system can help the host system to manage its duties. This is discussed in the next section.
6. OPERATING SYSTEMS MANAGEMENT SUPPORT

Due to the capability of the TMP to monitor and interact with the host node, it is possible to use it to carry certain operating system tasks, thus reducing the amount of overhead of the host processor. In this section, we describe three such tasks: the load balancing algorithm, which is responsible for dispersing the load among the nodes; the remnant collection algorithm, responsible for removing any remnants due to remote node failures; and the process location algorithm, which helps in re-establishing communication links between processes.

The algorithms described in this section represent a class of tasks that are performed by a system with a large number of nodes. Each such algorithm consists of several low-level tasks, e.g., monitoring the I/O rate of a process or sending keep alive messages, and a higher level tasks, e.g., process migration. The high-level procedures, which are invoked with a relatively low frequency, involve decision making that in many cases depends on some threshold functions. Updating these threshold functions requires that the system carry the low-level tasks, which in turn are executed with relatively high frequency. It is at this level of activity that the TMP services are needed most. Since the TMP can monitor the node activities with minimal overhead, it can relieve the host processor of carrying many of these functions, particularly network-related chores. The TMP performs the monitoring and the initial processing of the results. Then it may decide to initiate certain activities by triggering the operating system of the host processor or simply by updating its threshold functions for future use. The TMP can also send and receive information from other TMPs over the network, containing information about their respective hosts.

6.1. Load balancing

In distributed systems, load balancing algorithms are used to improve the overall system performance by transferring processes from overloaded machines to less loaded machines. Despite the overhead which is incurred by these algorithms, it has been shown that their execution can significantly improve the performance of the system [11]. Many existing algorithms for
load balancing use distributed control, i.e., each machine performs the algorithm independently of the other. Typically, these algorithms consist of four main parts: local information gathering, information exchanges among the nodes, matchmaking and the actual process migration [3].

In the first part, each node gathers information about the length of the internal queues and monitors the processes' activities. Close examination of this monitoring reveals that, in order to make good load balancing decisions, it is also necessary to profile each process, i.e., to record its CPU usage, rates of I/O and Inter Process Communications (IPC), the node locations to which remote operations are targeted, ratio of remote vs. local operations, the process size, etc. Furthermore, this profiling must be done frequently due to possible changes in the characteristics of the processes.

The task of profiling all the host system processes results in a considerable amount of overhead for the operating system. Due to its causes, this overhead increases with the number of processes, thus further reducing the efficiency of the system. We note that it is due to this kind of overhead that many multicomputer operating systems will grind to a halt when required to handle a configuration beyond a certain size. A hardware device, such as the TMP, is one way to reduce this overhead.

The second part of the algorithm includes load information exchanges. Using the communication links of the TMP, this load information can be exchanged among the TMPs by using some dissemination algorithm [1, 6], while at the same time, each TMP makes the information received available to the local operating system. The matchmaking part is done by comparing the information about the local load to that of other nodes, and by using some pre-specified decision (threshold) parameters [14]. The role of the TMP at this level is limited to triggering process migrations due to special requirements by individual processes.
6.2. Remnant collection

In a distributed system, processes executing in one site may have allocated resources such as entries in system tables in remote sites. As nodes and communication links are inherently unreliable, failures may disconnect processes from such remote objects. After such a failure, these objects are futilely allocated, without actually being used, thereby wasting their site’s resources. Essentially, they have become remnant in the same sense that inaccessible objects allocated from a heap storage are termed garbage [5]. As crashed sites are seldom able to notify other sites of their failure, these resources have to be actively detected and reclaimed for reuse.

The following remnant collection scheme is implemented in MOS [2]. When an object is allocated, a timer is attached to it. Processes are responsible for resetting the timer of the objects they use. This is done whenever an object is accessed, or when it receives a special *keep alive* message. A remnant collection process periodically scans all the objects, and releases those whose timers have expired. The timers are implemented by tagging each object with a unique creation time, and using a fixed upper limit on objects’ lifetimes. Resetting the timer is done by resetting the tag. The special *keep alive* messages are periodically transmitted by each node to all accessed objects.

The role of the TMP in the remnant collection algorithm becomes clear when we consider all the operations involved. Initially, the operating system is responsible for allocating the resources and for setting their timers. This includes registration of the initiating process in the local TMP (in the node that hosts the process), and registration of the active object, including its timer, in the TMP of the remote node. This "remote" TMP is then responsible for updating the timer of the object, while the "local" TMP is responsible for sending the *keep alive* messages, validating from time to time that the process is still active. Note that if the original process migrates from one node to another, then part of this migration procedure includes updating of the TMP at the new site. Also note that if the object is migrated, then in addition to updating the
TMP at the new site, it is necessary to inform the original process(es) about the object's new location. An algorithm that performs this task is described in the next section. The other activity which is carried by the "remote" TMP is removal of the object if its timer is expired. This can be accomplished using several additional safety measures, such as further negotiations among all the TMP involved.

6.3. Process detection

In a distributed system, a user's process may initially be assigned to one node, and later, as a result of load balancing considerations, be migrated to another node. In order to support IPC and to preserve the user's interface, the operating system must maintain up-to-date information about the location of all processes. A simple scheme to maintain such information is to establish a "home" site for each process, naturally in the node where it was created, and to continuously maintain in this location information of its whereabouts.

Suppose that a node which contains the home of a process is crashed, losing as a result all information about that process. Despite the crash, if the process has migrated to another site, then it may not be affected, and normally there is no reason to stop its execution. We call such a surviving process an orphan process. The existence of an orphan process has several undesirable implications. First, if an orphan process cannot be located, then this may lead to a violation of the user's interface, since the user may lose control over his processes. Similarly, IPC to an orphan process can not be performed despite the fact that this process exists and may communicate with other processes.

A simple, probabilistic scheme to overcome the single home site failure is used in MOS [2]. This scheme is intended to reduce the likelihood that a process becomes orphan by using multiple homes in a fixed set of nodes called the home pointer sites, and by sending keep alive messages between the process and its home pointer sites. More specifically, for each new process a set of pointer entries is created and placed in the home pointer sites. Note that the number of these sites
depends on the hardware reliability and the required degree of fault-tolerance, and does not depend on the size of the configuration. The specific locations of the home pointer sites are determined by a universal hash function (on the process number), which is known to all the nodes.

The algorithms that are executed in order to maintain communication between each process and its home pointer sites includes establishing the entries when the process is created, then after each process migration and at fixed time interval, sending keep alive messages from the process to its home pointer sites. To locate a process, one need only contact its home pointer sites, using the process number and the hash function.

It is evident from this scheme that the algorithm for locating process relies on replication of links and on time dependent, keep alive messages. The first part requires a fixed amount of overhead and may be considered part of the process creation routine. In contrast, the keep alive messages require a considerable amount of overhead, since this part of the algorithm is executed as long as the process exists. If the time interval between two consecutive messages is small, then the overhead of maintaining this service is increased. If this time interval is decreased, it may slow down the detection process and eventually effect some processes. The role of the TMP is sending the keep alive messages and updating the timers at the home pointers entries. The TMP is also responsible for informing the host system when a timer expires, thus triggering the remnant collection algorithm.

7. CONCLUSION

The underlying philosophy of the TMP concept is to view monitoring as an integral part of a computer system. The TMP combines the advantages of software and hardware monitors while overcoming their deficiencies. The monitoring system does not change the behavior or the performance of the monitored software, and operates with minimal overhead to the host hardware. The algorithms running on the TMP operate in a decentralized mode and manage their node auto-
nomously; thus the TMP concept can also be applied in a system with a large number of nodes.

Experience with the TMP shows an improved understanding of run time behavior and system performance, and it promotes qualitative and quantitative assessments about the execution of distributed applications. It can also be used to perform management support to the operating system, thus releasing it from this kind of low-level routine task.

Our experience is that real-time monitoring implies a separate monitor processor that operates completely transparent to users. The advantages of our monitoring architecture are worth the additional hardware costs since the monitoring system reduces the software costs which occur when developing, debugging and monitoring complex (distributed) systems. In the future, a TMP-like facility may become an integral part of any node, thus helping to improve the reliability, availability, flexibility, and performance of systems, particularly systems with a large number of nodes.

ACKNOWLEDGEMENTS

The authors wish to thank the contributions of many participants from the INCAS and MOSIX projects, especially M. Abel, A. Gabel, T. Gauweiler, J. Lichtermann, F. Mattem, R. Remke, K. Rohfels and A. Shiloh. Thanks also to A. Collison for the editorial assistance.

REFERENCES


