Application-Dependent Dynamic Monitoring of Distributed and Parallel Systems

David M. Ogle, Karsten Schwan, and Richard Snodgrass, *Senior Member, IEEE*

***Abstract*— Achieving high performance for parallel or distributed programs often requires substantial amounts of information about the programs themselves, about the systems on which they are executing, and about specific program runs. The monitoring system presented in this paper collects, analyzes, and makes application-dependent monitoring information available to the programmer and to the executing program. The system may be used for off-line program analysis, for on-line debugging, and for making on-line, dynamic changes to parallel or distributed programs to enhance their performance. We employ a high-level, uniform data model for the representation of program information and monitoring data. We show how this model may be used for the specification of program views and attributes for monitoring, and we demonstrate how such specifications can be translated into efficient, program-specific monitoring code that uses alternative mechanisms for the distributed analysis and collection to be performed for the specified views. The model's utility has been demonstrated on a wide variety of parallel machines, including several kinds of multiprocessors and a local area network.**

***Index Terms*— Application-dependent monitoring, distributed programs, dynamic monitoring, parallel programs, program adaptation, program configuration, program monitoring, program performance.**

I. Monitoring the Performance of Parallel Programs

ONE goal of writing programs for distributed and parallel architectures is enhanced performance. Attaining high performance often requires acquisition and use of substantial amounts of information about programs, about the systems on which they are running, and about specific program runs. Since such information is difficult to predict accurately prior to a program's execution [15], [72], programmers must experiment with and measure their distributed or parallel programs. For example, experimentation may be needed to determine the performance effects of a program's load on processors and communication links or of a program's usage of certain operating system facilities [15], [12].

Manuscript received February 25, 1991; revised December 1, 1992. This work was supported in part by the National Science Foundation under ECS-8307216 and ISI-8902707 and by an internal grant from the Department of Computer and Information Science at Ohio State. It was performed while D. Ogle and K. Schwan were at the Ohio State University and R. Snodgrass was at the University of North Carolina, Chapel Hill.

D. M. Ogle is with the Experimental Systems Department, IBM Corporation, Research Triangle Park, NC 27709.

K. Schwan is with the College of Computing, Georgia Institute of Technology, Atlanta, GA 30332.

R. Snodgrass is with the Department of Computer Science, University of Arizona, Tucson, AZ 85721.

IEEE Log Number 9210515.

This paper discusses the program monitor used by a system for experimental parallel programming. This programming system, called the Issos system, was able to utilize monitoring information to modify a program during its execution (dynamically) or prior to or after its execution (statically). Such changes, henceforth termed *adaptations*, were initiated by programmers, or were performed under program control without programmer intervention (e.g., for dynamic load balancing). In this fashion, the monitor and program construction system jointly facilitated the experimental evaluation of the performance of parallel or distributed programs.

The three interesting attributes of the Issos monitor are the following.

1) *Application-dependent monitoring:* The monitor allows programmers to specify the information to be collected, analyzed, and displayed for each application program, resulting in the generation of efficient, program-specific configurations of the monitor's collection and analysis mechanisms.
2) *Dynamic, distributed data collection and analysis:* The monitor both collects *and* analyzes information regarding the execution of a program in real-time. As a result, the monitor may be used to generate on-line displays of execution information, and it may be combined with tools that use such on-line information in order to change or steer the running program. Furthermore, for dynamic program changes that concern the improvement of program performance (program tuning), the overhead and latency of distributed information collection and analysis must not significantly reduce the performance gains realized by those changes. In response to these needs, the monitor is designed to permit tradeoffs in the amount of information collected and analyzed, the extent and the accuracy of analysis, and the extent of program adaptation. Such tradeoffs are realized in part by use of alternative means of information collection, such as tracing versus sampling.
3) *Use of a language-independent data model:* The monitor is independent from specific target programming or program execution systems. What is to be monitored is specified via the entity-relationship (ER) model [9]. Such monitoring specifications are stated with novel language constructs that also specify the desired latency and precision of monitoring, thereby permitting the configuration of collection and analysis mechanisms for different uses of the monitoring system and for different distributed and parallel execution environments. In addition, use of

the E-R model for description of all other information required for program monitoring and program adaptation (ie., compilation- and load-time program information, hardware configuration knowledge, etc.) facilitates the monitor's integration with the other tools used for program construction, execution, and adaptation. All tool interactions for purposes of program adaptation and information sharing are performed using a main-memory database [61], [62], [49], [66] that implements the E-R model.

Our approach builds upon research originally performed with the Cm* multiprocessor at Carnegie Mellon University [63], [65] that explored the use of the relational database model for representation of and access to monitoring information collected for a parallel program. However, due to our focus on the dynamic use of monitoring information, we are not concerned with long-term information storage using temporal databases [64]. Instead, our monitor must be able to analyze collected data instantaneously or with tolerable delays, with acceptable and variable overheads. In this fashion, we are able to meet the constraints in terms of both execution efficiency and interactions between the monitor and the application program (for dynamic adaptations) and the programming environment (for static adaptations). Variable overheads and delays are attained in part by variation of the distribution and parallelization of the monitor's collection and analysis functions across the nodes of the parallel and distributed target hardware. Such distribution and parallelization are performed automatically by generation of application-specific, customized collection and analysis code based on declarative language statements specified by application programmers.

This paper discusses the monitor's design and prototype implementation for three hardware and operating system configurations, thereby demonstrating the target machine indepedence of our approach: a seven-node custom multiprocessor running an experimental real-time operating system [57], a ten-node Encore MultiMax multiprocessor, and a local area network of Sun-3 workstations using a Pyramid mainframe as a file server. The set of applications with which the monitoring system is used includes several simple parallel and distributed programs written with the Issos system, such as the distributed quicksort program used as an example throughout this paper, and it includes two substantial applications written outside of Issos for evaluation of the monitoring system: 1) the on-line monitoring of properties such as "job load" for more than 100 Sun-3 workstations and 2) the on-line monitoring of communication load on the various subnetworks used for workstation connectivity.

In the remainder of this paper, we first present the low level data collection, analysis, and storage mechanisms that comprise the monitor. We then discuss the monitor in terms of the information model presented to the user, emphasizing how the user may specify monitoring at this fairly abstract level. A significant challenge to the monitor is translating constructs in the information model into the low level mechanisms. We discuss this translation in detail, and examine heuristics that are appropriate for each of the three hardware configurations on which the monitor has been implemented. One use of monitoring information, dynamic adaptation, is illustrated with the sample distributed quicksort program. We conclude with a summary, a comparison with related research, and a discussion of future work.

II. A Monitoring System

The monitor is responsible for the collection and analysis of distributed program information. Its overall structure is shown in Fig. 1 for one hardware configuration, a distributed research network connected by an EtherNet network. The *resident monitor*, residing on each network node, collects and analyzes monitoring information about processes executing on that node. The resident monitors report to a *central monitor* executing on the network node on which the *monitoring database* is stored. The central monitor collects and analyzes distributed information, interacts with the other tools in Issos, and provides a user interface.[1]

The monitor was operated stand-alone and it was used within the Issos system for parallel programming and program adaptation. The different components of the Issos system are depicted in Fig. 2.[2] The solid, labeled lines between the modules indicate the information exchanged prior to program execution. The dotted lines indicate some of the information exchanges during program execution (e.g., when using the system for dynamic load balancing). This figure depicts the *generation* of an instrumented, compiled, and loaded application; the run-time instrumentation of an application during *execution* is depicted in Fig. 1. The function of each component as it relates to the monitoring system is described below (see [62] for a more extensive description of this environment).

- *Program construction system* (PCS): The PCS is used for program entry, editing, compilation, and initiation of linking and loading. It describes a parallel program as a set of objects interacting via invocation, and it provides the initial description of each program to be monitored, in terms of the information model used within the environment, rather than in terms of the object-based program model. It also permits the specification of the adaptations to be performed for each program compilation and run [49].
- *Adaptation controller* (AC): The AC performs and supervises the specified program adaptations. It requests and receives information from the monitor in order to perform adaptations and it receives instructions from the user concerning the adaptations to be performed.
- *Loader and operating system* (OS): The Loader and OS are responsible for distributed loading, linking, startup, and execution of the object-based parallel program. They are also responsible for making available to the monitor and AC certain information regarding the distributed

[1] As other monitoring or performance display systems (e.g., the Paragraph system [22]), we have not been concerned with the system's failsafe operation. Redundancy or the use of stable storage would be required at the central monitor and database if the system were used for on-line failure analyses in distributed systems subject to network or workstation failures.

[2] We note that most parallel programming systems [63], [4] will contain similar sets of tools.

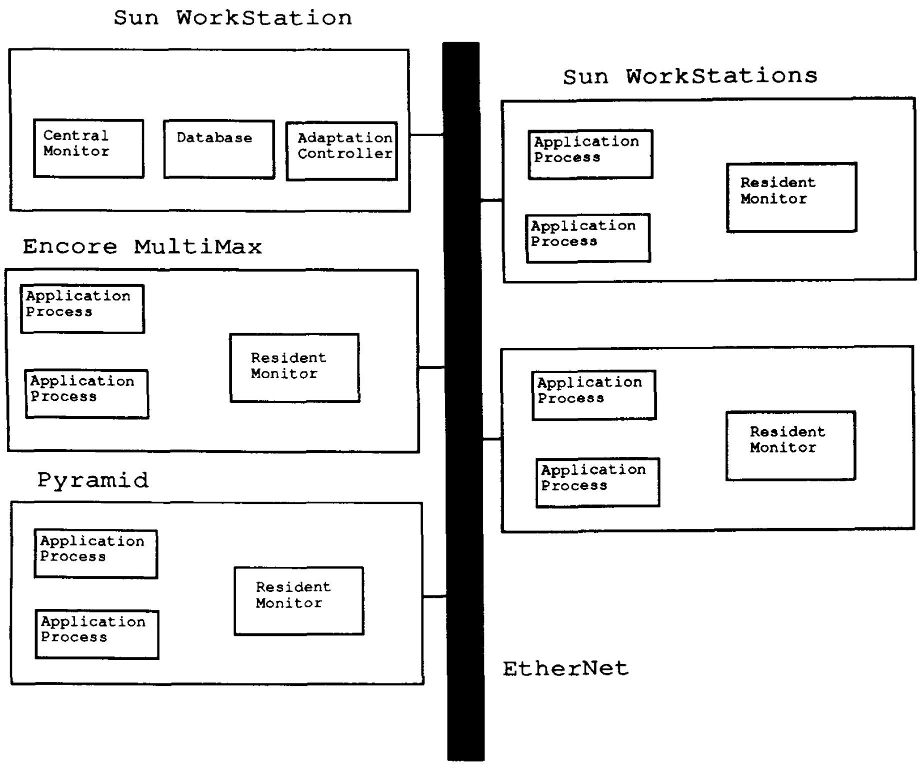


Fig. 1. The distributed monitoring system.

program, such as the mapping of objects to processes and the mapping of names used in object invocations to socket identifiers used by the processes implementing an object's operations [2] (see the dotted lines).

- *Monitor*: The Monitor is responsible for collecting, analyzing, and making available the program information required by the AC (as indicated by the dotted lines).

We will discuss the additional information in Fig. 2 (e.g., sensors and probes) in the next section. Also note that collected and analyzed monitoring information as well as initialization-time program information available from the loader and operating system (see the dotted lines in the figure) may be shared with the adaptation controller either directly or via the database.

A. Collection and Analysis Mechanisms

Information can be collected either by sampling or by tracing. *Tracing* consists of the reporting of all occurrences of an event within a certain interval of time. Tracing is synchronous with the occurrence of an event; it is performed when all occurrences of an event must be known (e.g., when collecting history information) or when each occurrence of an event must be followed by a certain action [65]. On the other hand, *sampling* is the collection of information at the request of the monitor. Sampling may be asynchronous with the occurrence of an event; it is useful when an immediate reaction to an event is not necessary.

Sensors are small pieces of code residing within the program being monitored. A sensor may perform either sampling or tracing, and reports information, such as current value and time, to the resident monitor. When to report such information is determined in part by the user at the time of monitoring specification. If a sensor also contains analysis code, it is termed an *extended sensor*. Sensors are generated automatically by the monitor based on the programmer's specifications of the events to be monitored. However, the insertion of the generated sensors into the application code must be performed manually; automatic placement requires the use of dependency analyses like those used in parallelizing compilers.

A sample sensor implementation in Unix BSD 4.3 is shown at the top of the next page.

This sensor traces the value of a program variable `bad_header_chksums` in a network device driver. It assumes the use of Unix sockets for the transmission of information from the sensor to the monitor. Our multiprocessor implementation of such a sensor uses shared memory to implement the required message sending primitive.

A *traced sensor* begins tracing when it is *enabled* by the resident monitor; it stops tracing when it is *disabled*. For example, a sensor tracing the value of the variable `Request_Queue_Size` in some particular process of a distributed application using the monitor generates an output each time the value of that variable is changed. The status of the sensor (i.e., enabled or disabled) is kept in the address space

```
if(status[1] == 1)
{
 sensor_struct.command      = NEW_VAL;
 sensor_struct.sensor_num   = 1;
 sensor_struct.sensor_time  = gettimeofday();
 sensor_struct.int_result   = number_bad_header_chksums;
 sendto(monitor_socket_send,&sensor_struct,sizeof(sensor_struct),0,
        &monitor_sin_send,sizeof(monitor_sin_send));
}
```

of the sensor's process and is checked when the sensor is encountered during execution of the application code.

The resident monitor receives trace data via *event records* generated by sensors. Event records contain 1) a command identifier flagging the information as sensor data, 2) a sensor number, identifying the reporting sensor, 3) the time at which this event was recorded, and 4) a sensor-specific value. Event records are communicated to the resident monitor by notification or by message. Communication *by notification* implies that the receipt of the record by the resident monitor is synchronous with the execution of the sensor. Communication *by message* implies that the composition and the receipt of the event record are asynchronous, since the message may be queued for an unknown period of time. For example, when collecting history information regarding the values of the variable `Request_Queue_Size`, event records can be received asynchronously (by message) if the resident monitor need not immediately know about the occurrence of each change in the variable's value. However, if the resident monitor has to react immediately to the event that `Request_Queue_Size` has exceeded some threshold value, then it must be interrupted synchronously with the event (i.e., it should be notified).

A *sampled sensor* simply returns a single event record in response to a sampling request from a resident monitor, again by message or by notification.

Probes are a collection mechanism with behavior and performance characteristics that differ from sensors (see Section IV-B and [25]). *Probes* are code fragments residing within the *resident monitor* (rather than the application) on each network node. Probes directly access the address spaces of individual processes on that node, thereby providing a convenient mechanism for sampling.[3] The main advantage of probes over sampled sensors is that the application code need not be changed for probing, so that the information to be probed may be defined dynamically. Furthermore, when monitoring parallel programs executing on shared memory machines, the use of probes versus sensors can reduce program perturbation due to monitoring [65], [10] (also see Section IV-B).

[3]As indicated in Fig. 2, the Unix implementation of probes with `signal` operations requires that some probe code also resides in the target address space (see [45] for a description of probe implementation in distributed Unix systems).

B. Storage Mechanisms

Any monitoring system must address the storage of the program information it produces. Since the primary use of our monitor is dynamic monitoring, we first store all monitoring information in data structures mapped to main memory using the operating system's virtual memory mechanisms, thereby reducing the latency of access to such information. For performance reasons, this collected data, termed the *database*, does not contain raw data. It contains analyzed data derived from the information collected using sensors and probes. All information stored in the database is tagged with time stamps and locations of occurrence, for use by dynamic and post-execution analysis of monitoring information. The data structures being used are straightforward template structures derived from the information model used for description of monitoring information. They are explained in detail elsewhere [49], [45].

Although the virtual memory database can grow to significant size, for long-term persistent information storage, we currently use ad hoc file structures, but would prefer using a large-scale historical database [64] in order to be able to perform efficient, additional post-execution analyses. Sample post-execution analysis of such stored information may concern additional analysis of interest to the programmer or the adaptation algorithms, or it may concern the reproduction or replay of program execution [32], [31]. However, note that post-execution analysis in our system is restricted to those queries that are possible to answer with the partially *analyzed* information contained in the database.

C. Run-Time Monitor Interface

The monitor supports three execution-time operations, which are ultimately invoked by programmer-stated monitoring specifications.

1) Turn a particular sensor on or off. This operation is performed to trigger a sampled sensor, or to begin or end the trace of a specific program attribute, such as the values of one of its variables. The issuer of this command may request to be notified when a condition or set of conditions regarding the variable's values becomes true.
2) Probe the current value of a program attribute.

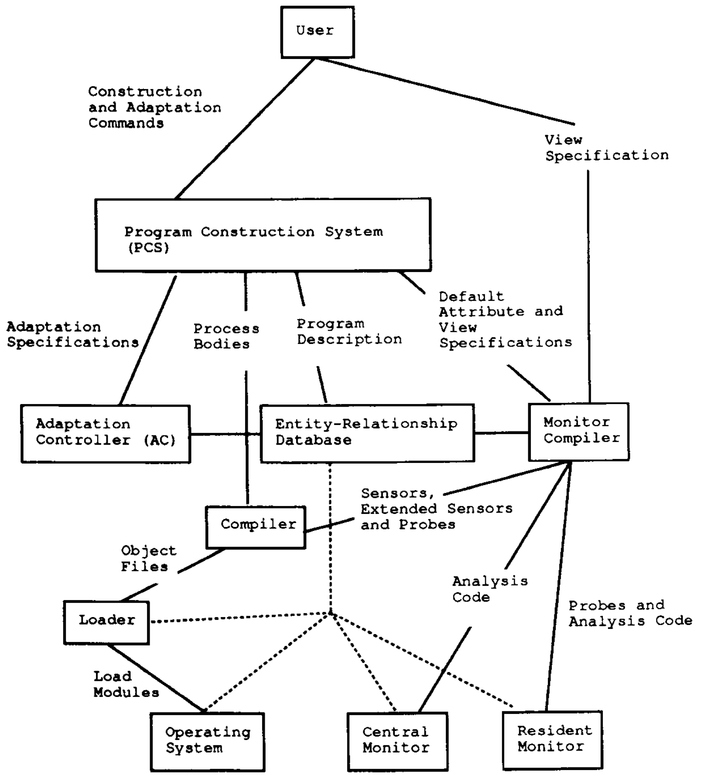

User
Construction and Adaptation Commands
View Specification
Program Construction System (PCS)
Adaptation Specifications
Process Bodies
Program Description
Default Attribute and View Specifications
Adaptation Controller (AC)
Entity-Relationship Database
Monitor Compiler
Sensors, Extended Sensors and Probes
Compiler
Object Files
Analysis Code
Probes and Analysis Code
Loader
Load Modules
Operating System
Central Monitor
Resident Monitor

Fig. 2. The Issos parallel programming and program adaptation environment.

3) Retrieve the value of a program attribute which the monitor is or has been tracing. The value is retrieved from the monitor's database.

The next section presents a model of information for use by the monitoring system and shows that this model is an appropriate basis for monitoring specifications. Such specifications are compiled into efficient collection and analysis mechanisms that employ the operations just presented.

III. An Information Model for Program Monitoring

In order to make the monitoring system independent of specific languages, compilers, operating systems, etc., we describe in terms of an abstract information model based on the E-R information model [9] the programs for which monitoring is to be performed, the hardware and software environment in which the programs execute, the data to be collected, and the calculations to be performed. Our model includes typed entities, typed relationships between entities, and typed sets of both. The model can incorporate static information about parallel programs and about their execution environments [62], thus capturing compile- and load-time program information, hardware configuration, and others. Specifically, in the Issos system, the program construction system generates an E-R program description and records it in a main memory database accessible to all system tools, based on the program's language specification and on its knowledge of the program's run-time representation. Other system components (e.g., the loader) store information (e.g., hardware configuration information, processor allocation maps [62]) in the same repository.

The objective of this paper is not to describe and defend the E-R model, the database supporting it, and their usefulness for tool integration. Elsewhere we provide details on the Issos system and describe a more efficient, persistent and distributed database implementation [49], [46], [17], [18]. Other approaches to tool integration also exist [69], [23], [50], [51]. Here we simply describe a sample program represented with the E-R model, so that the reader may understand the monitor's view of a parallel program.

A sample parallel program: Consider a parallel sort program like the various versions of parallel quicksort described in the literature [14]. In the object-based Issos programming system, this program is represented at run-time as objects interacting via invocation messages, much like the representation of distributed programs in Eden [2]. The sort program consists of several objects, including a `Queue` object which contains a process that maintains a queue of unsorted subranges of the array being sorted and a `Sort` object which contains multiple internal processes performing the actual sorting of the array.

While the Issos run-time system represents the program as a set of objects, the monitor views the program with the E-R model, thereby using a description that is independent of the execution environment's program representation. With the E-R model, this program may be described as consisting of entities of type `process`, each with a default attribute `Process_Id`. These entities participate in the relationship `Communicates_With`, itself associated with the attribute `frequency`. Thus, we permit the association of attributes with entities and with relationships. The `Communicates_With` relationship describes *all* possible message (or invocation) traffic within the program. If users desire to use subsets of such information, such as the communications between processes P1 and P2, or P4 and P5, *views* defining such subsets may be created using E-R model operations.

Regarding monitoring, the values of some attributes (e.g., `Process_Id`) may be supplied automatically by the parallel programming system. Other attributes must be defined by the programmer. For example, in the sample quicksort program, the user may wish to be notified when the size of the `Queue` object exceeds some predefined threshold. In that case, the programmer's monitoring specification would explicitly define the attribute `queueSize` of the `Queue` object. The `queueSize` attribute may be mapped to a variable called "q_size" in the application's code. However, attributes may also be defined in terms of multiple variables used in the application program. In general, an attribute's value is an expression over one or more variables. Attributes are type-checked by the PCS, through which the application was originally coded. As an example, again consider the `Queue` object of the sample quicksort program. In order to evaluate workload balancing among multiple processes performing the sort of unsorted queue subranges, the user may wish to monitor the attribute `requestDuration` for each element of that queue. This attribute is not predefined and is not maintained as a variable in the code and therefore, cannot be generated by the PCS. Instead, such an attribute must be computed for each request from the source code variables `beginRequestTime` and `endRequestTime` which are maintained in the code in this case.

Monitoring specification: Two simple languages are used for the specification of program monitoring in the context of the E-R model: the attribute language and the view language. All nondefault, monitorable attributes of a parallel program must be explicitly defined with the *attribute language*. For example, the following specification concerns the attributes `queueSize` (mapped to C variable "q_size") and `requestDuration` of the object called `QueueManager`.

```
ATTRIBUTE DEFINITION FOR OBJECT
 QueueManager
    requestDuration: (endRequestTime -
     beginRequestTime);
    queueSize:          q_size;
END ATTRIBUTE DEF
```

In our C-based implementation of the monitoring system, the expression `(endRequestTime - beginRequestTime)` must be a legal C expression.

As indicated in Fig. 2, all attribute specifications are compiled into probes and into attributes of entities stored in the database. This is also the case for the attributes of relationships that can be monitored. Probe implementations (as code fragments in resident monitors and as code executed as signal routines in target processes) are linked and loaded with the target application's processes and resident monitors, and they are registered with the central monitor. Similar actions are taken for the sensor implementations and the analysis code derived from the view specifications explained next.

Attributes that can be monitored constitute the basis from which the set of actual events to be monitored is drawn. That set is specified with the *view language* by programmers as a collection of *monitoring views* stated as entities, relationships, and sets in the database. Each such view specifies 1) the involved entities (or relationships) and attributes, 2) the time at which the view is considered active, 3) performance and correctness criteria, and 4) the action to be taken when the view is active.

The sample view below concerns the queue sizes of two `QueueManager` objects. The *view language* syntax accepted by the prototype described later is similar to that appearing below.

```
VIEW DEF Both_Queue_Limits_Exceeded
 (thisqueueSize:
 QueueManager[1].queueSize)
        ACTIVE WHEN (QueueManager[1].
         queueSize > 24)
                AND (QueueManager[2].
         queueSize > 24);
        CORRECT TO WITHIN 25 MS;
        NOTIFY 119 OF seventh@cis.ohio-
         state.edu WITHIN
         1000 MS;
END VIEW DEF
```

This view is defined to be *active* when the value of `queueSize` is greater than 24 in both instantiations of `QueueManager`; this boolean expression on attributes is termed the `ACTIVE` predicate. When a view is active, the value of its derived attributes, mentioned in the *target list* of the view (in this case, the target list consists of the single attribute `thisqueueSize` computed by the expression `QueueManager[1].queueSize`), are computed and made available by the monitor to other environment tools, such as the adaptation controller and the graphical display. Since the monitoring system need not collect, record, and display information at times at which the view is not active (ie., when `queueSize` ≤ 24), the `ACTIVE` predicate reduces the amount of work the monitor must do, thereby reducing monitoring perturbation.

The `CORRECT` clause potentially further reduces monitoring perturbation by relaxing the timing constraints imposed on the calculation of the active predicate. This clause allows users to express allowable tolerances in monitoring due to network delays and unsynchronized processor clocks. In this case, if the queue size of each queue manager exceeded 24 within a window of 25 ms, then the predicate is considered satisfied. If the `CORRECT` clause is omitted, then the view is active only when both queue sizes are *simultaneously*[4] greater than 24.

[4] In a distributed system, the meaning of *simultaneous* may be defined in terms of the maximum network delay incurred for message transmission [45].

The NOTIFY clause instructs the monitor to directly communicate in some manner the new value of the view's attribute to the application or AC process at port address 119 on machine seventh@cis.ohio-state.edu whenever the view becomes active. The maximum latency of this notification can also be specified; here the process expects to be notified within one second. Again, a long latency provides the monitor flexibility in reducing its overhead. Monitoring message traffic may be reduced by buffering values in user processes, resident monitors, or the central monitor. For example, the large value specified here allows the central monitor to buffer multiple messages before performing a single message send to the AC with the buffered messages. One implementation used in our system simply "flushes" all buffers of any relevant resident monitors after some maximum permissible delay in reporting has occurred. If the NOTIFY clause is omitted, the monitor is instructed to simply update the entry corresponding to the view's attribute(s) in the database when the view becomes active.

Note that the analysis phase of monitoring often requires the comparison of time fields among multiple events that may have occurred on different nodes of the distributed system. We have not explored any novel methods for event or time synchronization [29]. For most of our applications, it has been sufficient to assume that processor clocks are synchronized to within tens of microseconds.

In conclusion, views as defined above are useful for specifying dynamic monitoring for several reasons:

- The real-time attributes of program monitoring, such as maximum monitoring delays, etc., are easily specified.
- Performance specifications regarding monitoring are easily stated, which results in the generation of efficient collection and analysis using probes, extended sensors, and analysis code in resident and central monitors. In essence, the monitor can tailor its collection and analysis mechanisms to single applications or even single execution runs of applications.
- Language and system independence are achieved by expressing views in terms of attributes rather than in terms of program variables present in the application. In addition, any program described with the E-R model may be monitored, including operating system components and systems software (e.g., the Unix network file system [45]).

See [27] for a more extensive discussion of language issues in the specification of monitoring. The attachment of graphical representations to monitoring views is the subject of other past and future research performed by our group [60], [26].

IV. Data Collection and Analysis

It should be apparent from the examples in the previous section that the attribute and view languages permit users to express application-dependent monitoring views in a high level, declarative fashion. Important aspects like perturbation and latency may be expressed as constraints, rather than procedurally. This implies that programmers need not understand the details of information collection and analysis, of monitoring system setup and distribution, etc. However, it also requires that such declarative specifications be automatically mapped to the low level collection and evaluation mechanisms discussed in Section II by the monitoring system's compiler. Such mappings must ensure that the information collection and analysis meet possibly stringent real-time constraints, while minimally perturbing the application as it executes. Fortunately, mappings may be varied along several dimensions. This section will describe how the monitoring system's compiler may determine appropriate mappings.

Recall the sample view given previously.

```
VIEW DEF Both_Queue_Limits_Exceeded
 (thisqueueSize:
 QueueManager[1].queueSize)
        ACTIVE WHEN (QueueManager[1].
         queueSize > 24)
                   AND (QueueManager[2].
         queueSize > 24);
        CORRECT TO WITHIN 25 MS;
        NOTIFY 119 OF seventh@cis.ohio-
         state.edu
          WITHIN 1000 MS;
END VIEW DEF
```

Concerning such a view, the monitoring system's compiler must deal with the following questions.

- For each attribute in the view's target list and action predicate (e.g., QueueManager[1].queueSize), should each variable mentioned in the expression providing the attribute's value (e.g., queueSize) be sampled, traced, or probed?
- When should sampled sensors be sampled, and when should traced sensors be enabled and disabled (e.g., do we need to constantly sample both QueueManager[1].queueSize and QueueManager[2].queueSize)?
- Where should each subexpression of the action predicate and the expressions in the target list be performed: in the relevant sensor, in the application code, in the resident monitor, or in the central monitor? Note that in this example, the target list does not contain an expression to be computed—QueueManager[1].queueSize.
- Should the sensor send event records to the resident monitor, or directly to the central monitor?
- How long should event records be queued in the application process, and in the resident monitor? How long should notifications be queued?
- Should notifications be generated only in the central monitor, or also in the resident monitor, or even in the application program?

We emphasize that these decisions are not independent, and that some alternatives are not available for all views. For instance, if the action predicate mentions two attributes associated with objects residing on different processors served by separate resident monitors, then the analysis *must* be done in the central monitor, as neither the sensors nor the resident monitors have sufficient information to evaluate the action predicate. As another example, if the analysis is to be performed in the resident monitor, then obviously event records should not be sent directly to the central monitor.

A. Tradeoffs in the Generation of Collection and Analysis Code

The monitor must make the decisions listed above when generating the sensor and probe code, using the following information available to it.

- The location of the resident monitor process in relation to the application process(es), either on the same processor or on a dedicated processor in the same multiprocessor.
- The cost of event record generation in the application process.
- The communication cost between the application process and the resident monitor, and between the resident and central monitors, expressed as a fixed overhead plus the cost per byte of event records sent.
- The latency constraint of evaluating the action predicate, as expressed in the `CORRECT` clause of the view.
- The notification latency constraint, as expressed in the `WITHIN` portion of the `NOTIFY` clause.
- The relationship between program variables referenced (indirectly through the attributes) in the target list and the action predicate of the view, specifically, whether one object or multiple objects are involved, the process or processor co-residency (or lack thereof) of objects involved, an estimate of the selectivity of subexpressions in the action predicate, and the approximate evaluation cost for the target list and action predicate.

To illustrate the manner in which these considerations affect the decisions made during sensor generation, consider the distributed monitoring system in Fig. 1, consisting of sensors, probes, and central and resident monitors. In this system, the analysis of data being collected may be performed either by individual sensors, by the resident monitor, by the central monitor, or by any combination thereof. Several implementation tradeoffs result.

One tradeoff is monitoring overhead versus communication cost. Analysis that is performed by an individual sensor, which is then termed an *extended sensor*, reduces the sensor's degree of interaction with its resident monitor, thereby reducing monitoring overhead. For simple analyses that are relatively inexpensive compared to the cost of communication, extended sensors may be preferred. More complex analysis must be performed elsewhere, so that needless perturbation of the process being monitored is avoided.

A second tradeoff involves computation within the resident monitor. If the analysis is performed within a resident monitor, its interactions with the central monitor are reduced. However, excessive analysis within a specific resident monitor may lead to an undue computational load and process switching overhead being imposed on the same processor on which the application processes being monitored are executing. This may not be tolerable for certain multiprocessor or real-time architectures [57], as shown below.

Additional tradeoffs concern the central monitor. If the central monitor does not perform analysis and simply forwards unanalyzed data to the agent that requires the monitoring information (in the case of Issos, the user or the adaptation controller—see Fig. 1), excessive communication may result between central monitor and the "user." However, the agent itself may decide what analyses should be performed; it retains complete freedom regarding the questions that may be asked about the data being collected (consider the post-execution analysis performed in some monitoring systems). Alternatively, such freedom may be sacrificed by performing analysis within the central monitor, thereby reducing the degree of interaction with the "user." However, such centralized analysis is again limited due to restrictions in the bandwidths of sensor to resident monitor to central monitor communications.

B. Generating the Collection and Analysis Code

A four step analysis, analogous to that used in query optimization in traditional database management systems [71], may be followed during the generation of collection and analysis code distributed across the application program and the monitoring system.

- Step 1: Generate all possible *view implementation plans* that preserve the semantics of the target list and of the action predicate.
- Step 2: Discard those plans that violate the latency constraints expressed in the view definition, using a simple analytical model that estimates the maximum latency of a given plan [45].
- Steps 3 and 4: Choose the plan from among the remaining plans that minimizes the monitoring perturbation, as predicted by the analytical perturbation model [45].

A view implementation plan generated by this process provides answers for each specific view to the questions listed above. Next, we examine each of these steps in more detail.

Step 1: Plan generation. As an example, consider the view given on page 768. A sample implementation plan for this view would be the following.

> Install a traced sensor in `QueueManager[1]` in the queueing routine. This sensor generates an event record containing the value of the variable recording the queue size whenever a queue operation is invoked. Install a similar traced sensor in `QueueManager[2]`. Both sensors send event records to the resident monitor, with no queueing, which sends them to the central monitor, again without queueing. The action predicate is evaluated in the central monitor. If it is satisfied, the `thisqueueSize` attribute for the view is recorded in the main memory database, and a notification is sent to the proper process, without queueing.

Another view implementation plan would be the following.

> Install a traced sensor in `Queuemanager[2]` that generates event records when the value of the variable recording the queue size transitions above or below the threshold 24. These event records are sent directly to the central monitor, which then probes the value of the same variable in `QueueManager[1]`. If that value is also above 24, it is recorded in the

main memory database and a notification is added
to a notification queue, flushed every 750 ms.

A fundamental requirement of any generated view implementation plan, termed a *feasible plan*, is that it preserve the semantics of the target list and action predicate (the latency constraints are ignored in this step). While both of the plans outlined above are fine in this regard, the following is not, assuming an environment consisting of workstations communicating via an Ethernet.

Trace the value of the variables recording the queue size in both `QueueManager[1]` and `Queuemanager[2]`, and evaluate the action predicate in the resident monitor.

Since `QueueManager[1]` and `QueueManager[2]` may be executing on different workstations, event records from both may never be present within any one resident monitor, preventing the evaluation of the action predicate.

To generate all possible view implementation plans, the monitor's compiler should incorporate all of the following choices in all possible combinations: sampling, tracing, or probing the value of each attribute mentioned in the view; performing each subexpression in the sensor, in the resident monitor, or in the central monitor; having each sensor send records to the resident monitor or directly to the central monitor; and generating notifications only in the central monitor or also in the resident monitor. In the sample view, there are two attributes mentioned, and three possible subexpressions, generating approximately 2000 view implementation plans.

Clearly, the space of all possible view implementation plans may be very large for complex views or architectures. Enumerating the feasible plans may be simplified by eliminating easily detected infeasible plans (and entire related collections of plans) early, by postponing timing decisions (such as how long event records should be queued) until the third step, and by reorganizing the action predicate so that variables from the same entity occur together in the subexpressions. For example, simply recognizing that `QueueManager[1]` and `QueueManager[2]` may be executing on different workstations eliminates some 1500 plans. In most cases, the feasible plans constitute a small subset of all possible plans.

Step 2: Applying latency constraints. Two latency constraints may be specified: `CORRECT WITHIN` and `NOTIFY WITHIN`. For each feasible view implementation plan produced by the first step, the monitor applies a simple analytic model to estimate the delay between the occurrence of the event and either the evaluation of the action predicate (and target list) or the receipt of notification. The model includes estimates of CPU time to process messages and perform analyses, as well as estimates of message transmission time. Details of the analytical model, as well as its validation, are given elsewhere [45], [25]. Here we will apply the model to the two sample feasible view implementation plans discussed above. For the first one, the latency involves the time to execute the sensor, the time to transmit the event record to the resident monitor and then to the central monitor, the processing involved in the resident and central monitors for this message transmission, and the time to perform the analysis at the central monitor and to send a notification message.

For the distributed Unix implementation of the monitoring system, message transmission was measured as roughly 3 ms between processes on the same machine, 4 ms between processes on the same subnet, and 10 ms between processes across multiple subnets under conditions of low Ethernet traffic (using a 10 MBit Ethernet). Each event record is first sent to the resident monitor on the same machine (3 ms) and then to the central monitor (10 ms). The total processing cost, dominated by several context switches, is less than 2 ms, implying a total latency on the order of 15 ms, which is less than the specified 25 ms. The notification latency is estimated at 26 ms, significantly less than the 1000 ms requested. Similar analyses for this view implementation plan mapped to the Encore Multimax and to the GEM real-time operating system executing on an Intel 8086-based multiprocessor would show that the specified latency constraints would be met there as well.

Now consider the second view implementation plan presented above, where the value in `QueueManager[1]` is probed, when executed on the same local area network. When the value of `queueSize` in `QueueManager[2]` exceeds 24, the sensor sends a message to the monitor, which takes approximately 3.5 ms. The monitor then probes the value of `queueSize` in `QueueManager[1]`. If `QueueManager[1]` resides on a different node, then probing takes approximately 3 messages, 1 intranode and 2 internode messages, or about 17 ms. Since 17 ms is less than the specified correctness value of 25 ms, this plan also satisfies the latency constraint. Note, however, if the user had instead specified `CORRECT TO WITHIN 15 MS`, the first plan would have been acceptable, but the second one would not have been.

The accuracy of the model is important only when the estimate is similar to the latency constraint specified by the user. In the example above, the model could have been off by 50% without changing the result of accepting both plans. Our general strategy has been to apply the model conservatively, recognizing that some plans may nevertheless be prematurely rejected due to inaccuracies in the model, with another plan chosen that also meets the latency constraint yet perhaps exhibits greater perturbation. In this example, approximately 150 feasible plans also satisfy this latency constraint.

Step 3: Applying per event record perturbation analysis. After steps 1 and 2 have been performed, the remaining view implementation plans are correct, but they can differ markedly in performance. In this step, the monitor applies a simple analytical model, similar to that for latency, to estimate the perturbation each plan would impose on the executing application process. This model must be applied carefully, as the absolute *perturbation*, expressed as the total CPU cost added to the execution time of the application process, depends on the total number of event records generated, which of course is unknown *a priori*.

The perturbation model is applied by partitioning all remaining plans into collections. Each plan in a collection will generate approximately the same number of event records as other plans in that collection. Then, for each plan, the CPU overhead is estimated for the processor on which the application process is executing (CPU overhead on processors dedicated to monitoring will not perturb the application). This estimate is on a *per event record* basis, and is thus quite accurate. Those plans with an estimate higher than the minimum for the collection are eliminated, leaving one plan per collection. In our example, there would be 24 collections, with these two plans in different collections.

Step 4: Choosing the final plan. After Step 3, two plans would remain feasible (the second plan may be eliminated in favor of one that traced `QueueManager[1]` and then probed `QueueManager[2]`, which would have an identical cost per event record). At this point, plans differ in both their perturbation per event record *and* in the number of event records generated. To make a final choice, the monitor must estimate the relative number of event records generated among the alternative plans (estimating the *absolute* number of event records generated is not necessary or possible), then use this to estimate the total relative perturbation, selecting the plan with the lowest such perturbation. This analysis thus estimates perturbation per notification generated, using *subexpression selectivities*, that is, given the subexpressions in the action predicate, the actual percentage of their evaluations resulting in the value "true."

For the first sample view implementation plan, the perturbation per notification will be K^2 times two sensor executions plus four message sends (since the resident monitor resides on the same processor as the application process, at least for a LAN environment), where $1/K$ is the selectivity of each subexpression (`QueueManager[`i`].queueSize > 24`) and the AND operator multiplies the selectivities. Here, potentially many sensor executions will occur between each notification (depending on the selectivity), perhaps resulting in a large perturbation per notification. For the second plan, the perturbation per notification will be K times the sensor execution, which has a cost of one initial message plus three message sends for the probe. Since $K > 1$ (probably $K \gg 1$), the second view implementation plan is preferred, how strongly depends on K, which is not known. In fact, the second view implementation plan would be the one selected.

To summarize, we propose a four-step process to generate code from a view specification. First, all feasible view implementation plans are generated. This is done either by simply enumerating all possible plans and then eliminating those that do not pass fairly simple correctness checks, or by applying the checks during the enumeration to avoid enumerating entire groups of incorrect plans. The second step filters out those plans that do not meet latency constraints. This step employs an analytical model that estimates the per event record latency. Inaccuracies in the model may eliminate some plans that meet these constraints. On the other hand, all remaining plans will be correct. The third step partitions the remaining plans into collections of plans that generate approximately the same number of event records. The most efficient plan in each collection is selected, based on a per event record analytical model of the CPU overhead. This model is quite accurate. In the fourth and final step, one plan is selected from those that remain based on an informal analysis that takes into account both the per event record perturbation and the number of event records generated. Inaccuracies in this analysis are much higher than in the previous steps. However, the inaccuracy will manifest itself in less efficient data collection, rather than incorrect data collection. Also, at this point, only a few plans are being considered; the vast majority of initially generated plans having been eliminated by application of more accurate analyses.

C. Customizing Plan Generation

While the full four-step process presented in the previous section may be automated, it can be be simplified significantly for particular hardware and software configurations. In this section, we present the plan simplifications used for the three hardware configurations on which the monitor has been implemented.

Local area networks: The first configuration is a local area network containing Sun-3 machines and a Pyramid communicating over an Ethernet (see Fig. 1). As discussed in Section IV-B, communication in such an environment is very expensive compared with processing time. Hence, for this configuration, we apply the following heuristic: push analyses to the lowest level where they may be performed, thereby reducing communication as much as possible. This decision is motivated by the experimental results presented next, and is justified elsewhere [45], [25]. In particular, this heuristic can be shown to minimize perturbation and latency simultaneously for this configuration with all but artificially complex view specifications. This heuristic ensures that analyses of monitoring information possible within the same address space in which the required sensors are located will be performed locally. A resident monitor performs the analysis that requires event records collected from different processes on its node, and the central monitor performs the analysis that requires event records from multiple machines.

The experimental results regarding the perturbation experienced in the distributed implementation of the monitoring system described next rely on a distributed workload generator. In the experiment below, the generator's configuration consists of two event generator processes, both of which are colocated on a single Sun workstation. A resident monitor is also located on the workstation, but the central monitor resides on a different workstation on the same subnet. Each event generator process generates up to 5000 randomly drawn events. At each event time, the process may elect to generate or not generate on actual event, where the generated event is an assignment of the value 0 or 1 to a local variable mapped to a monitoring *attribute* in the process (`attribute1` and `attribute2`, respectively, for each of the generator processes). The global event to the evaluated by the monitoring system is

`attribute1 > attribute2` ?

The global event's frequency of change for each program run is not known due to the randomness of the individual event generators.

In the measurements below, generator processes are first run without monitoring and then the event of interest is analyzed with extended sensors, by the resident monitor, or by the central monitor, respectively, each time measuring the resulting program perturbation. The table below depicts the measurement results (in these measurements, the timing method used was determined to have a 1% maximum error):

	Elapsed time (s)	Perturbation
Unmonitored	185	–
Central	246	61%
Local	233	48%
Sensor	226	41%

In all cases above, the actual overhead reported here is dominated by the use of Unix communication primitives. Thus, the exact amounts of the reported overhead percentages is not relevant. Instead, observe the differences in the amounts reported above. Specifically, the entry "Unmonitored" depicts the total time in seconds for the unmonitored execution of two generating processes located on the same machine. The entry "Central" assumes the generation of event records by generator processes each time an actual assignment to `attribute1` or `attribute2` is performed. Those event records are then sent to the nonlocal central monitor (via the resident monitor), which compares the values of the respective attributes. Compared to the measurements in row "Central," it is apparent that a comparison of attribute values using a resident monitor on the generators' workstation is preferable to central monitoring (see the entry labeled "Local"). This result holds despite the additional cost of context switching caused by the execution of the resident monitor on the generator processes' workstation.

Next, consider a different global event, which permits the system to use extended sensors for the event's analysis. If the global event is

`attribute1 = 1 OR attribute2 = 1` ?

then an extended sensor will generate events for the resident monitor only if its attribute's respective value is 1, thereby reducing the total number of event records generated by the process from 5000 to 2500 total (on the average). This sensor-based analysis results in the lowest perturbation reported in the table above (see the entry "Sensor"). Further reductions in perturbation may be achieved in several ways, including the use of shared memory among user processes and the resident monitor to share monitoring information [66], [65], the use of threads versus processes for representation of resident monitors (thereby reducing context switch cost), or the delivery of monitoring information across additional communication links among workstations, much like with the monitoring hardware additions in the Intel Paragorn machine.

In conclusion, the measurements reported in the table are a simple illustration of the heuristic mentioned above: in the network environment, analysis should be moved as close to collection as possible. Note that this observation holds in computer networks, multicomputers [7] and multiprocessors [45], [25], as long as the communication costs significantly outweigh the costs of the analysis being performed. We conjecture that this result will also hold for the monitoring hardware provided with the new Intel Paragorn multicomputer, since its communication bandwidths are significantly less that the computational power of the Intel 860 used as a communication co-processor.

A real-time multiprocessor: An implementation of the monitor on a real-time multiprocessor system exhibits differences in several basic system parameters and therefore, dictates the use of different heuristics. This multiprocessor was composed of seven nodes, each containing an Intel 8086 processor, which is somewhat slower than the Motorola 68020 processors in our Sun-3 workstations.

First, in this system, the relative cost of sending messages within and among different processors is lower than in Unix. Specifically, the GEM real-time operating system executing on this multiprocessor provides message sending primitives that can transmit small messages (i.e., event records) within 1 ms, compared with 4 to 10 ms between somewhat faster Sun-3 workstations. Second, this message communication overhead is roughly equivalent to the overhead of process switching in GEM (also approximately 1 ms). Third, the bandwidth of the bus connecting different multiprocessor nodes is quite high and generally underutilized (measurements demonstrate bus utilization of less than 20% for the real-time robotics applications running on this machine). Fourth, the multiprocessor's link to the monitoring system's user interface (on an attached Sun workstation) has comparatively low bandwidth and high latency compared to the intra-multiprocessor links. As a result, for this hardware configuration, we dedicated a single processor to the execution of a single resident monitor. Sensors and extended sensors send event records to this resident monitor at a cost of roughly 1 ms per event record. The resident monitor performs all analyses not done by extended sensors and it also performs those analyses done by the central monitor in the distributed system (in order to reduce the bandwidth of communication from resident to central monitor). The resident monitor communicates with the central monitor executing on a Sun-3 workstation and providing a graphical interface to the user.

Commercial, small-scale multiprocessors: A similar monitoring architecture was adopted for an Encore Multimax multiprocessor, which could be used for execution of selected components of a parallel/distributed program mapped to a set of Sun workstations and the Encore Multimax [10]. Here, a single Unix process acting as a resident monitor is responsible for all application processes executing on the Encore machine. This resident monitor sends event records to the central monitor executing on a Sun workstation, which may also communicate with resident monitors located on other Sun workstations. On the multiprocessor, monitoring overhead is reduced further by use of shared memory for the storage of event records [65], [10]. Sensors and extended sensors may write event records directly into shared memory accessible to the resident monitor. When using NSC 032 processors with a shared memory access time of 10 μs, a pointer to an event record can be generated and queued in 144 μs, then retrieved from the queue in 250 μs. The actual record can be generated and read in approximately 900 μs, and a sensor is turned on or off in 350 μs. These measurements imply that a single resident monitor may fully utilize its processor if all other processors on the ten-node Encore Multimax generate events at the fastest possible rate. Similar results should hold for the NSC 332 or 432-based Encore machines now in use. However, as with the real-time multiprocessor, excessive communication with the central monitor will result in low utilization of the dedicated Encore node. We have observed similar results on a 32-node GP-1000 BBN Butterfly multiprocessor with another version of the monitoring system [26].

Summary of results: To summarize, it appears that both the configuration of the monitoring system in terms of resident and central monitors and the selection of appropriate monitoring plans using probes and sensors, depend on the characteristics of the underlying hardware and on application characteristics or requirements stated with the attribute and view languages. It would be interesting to consider the automatic derivation of such application requirements from information supplied by the programming environment or by the adaptation controller.

V. Monitoring Programs for Dynamic Adaptation

This section describes a program monitoring and adaptation example that highlights some of the design and implementation issues in distributed, dynamic monitoring. This example uses the Issos parallel programming environment (see Fig. 2 and [55], [61], [62], [49]). The sample parallel (and distributed) program is a game.[5] This game shares one aspect with many parallel and distributed programs, including parallel branch-and-bound applications [58], parallel MultiLisp programs [52], and others. Namely, the game is subject to problems with workload balancing, since the program dynamically generates and consumes units of work that cannot be predicted statically (prior to program execution).

The game consists of ships moving on a sea. The description of the two-dimensional sea is partitioned into sections, with a section manager process responsible for each section. Ship manager processes are responsible for handling requests dealing with ships, such as moving and firing. All requests are placed into a single, logically centralized queue, maintained by a queue manager process. Ship managers take and process requests from this queue. The game is driven from a script, with multiple user processes reading this script and issuing requests to the queue manager.

This distributed application illustrates several aspects of the monitoring system, including:

- the operation of the monitor's distributed components;
- the interaction between the monitoring system and other Issos tools;
- the tradeoffs regarding the use of the monitor's various collection mechanisms and the tradeoffs regarding the distribution of information analysis; and
- the tradeoffs between tracing and sampling of program execution.

[5]The applications with which the monitor and the Issos system have been tested include the parallel quicksort program, a robotics application, and a synthetic workload generator.

Dynamic monitoring—basic requirements: The usefulness of dynamic monitoring is demonstrated using a small version of the game, consisting of a user and a ship manager process executing on the Pyramid, and a queue manager process executing on a Sun workstation (see [56], [5], [6], [19] for studies of useful algorithms for dynamic program adaptation based on monitoring data). The monitoring system's components are the central monitor, the PCS, and the AC executing on the Pyramid and the resident monitor executing on the Sun. This example demonstrates the dynamic, joint operation of the central and resident monitors with the AC and PCS. The purpose of this cooperation is to balance the rates of request generation by the user process and request processing by the ship manager. The monitoring statement

```
VIEW DEF Queue_size (Queuesize:
 Queue_manager.Queuesize)
   ACTIVE WHEN (Queue_manager.Queuesize
    >= 15);
   CORRECT TO WITHIN 25 MEC;
   NOTIFY 119 OF seventh@cis.ohio-
    state.edu
    WITHIN 875 MS;
END VIEW DEF
```

instructs the monitor to notify the AC when the size of the request queue maintained by the queue manager process (the attribute `QueueSize` of the program component `Queue-Manager`) exceeds the statically specified threshold "15," whereupon the AC causes the PCS to create a second ship manager process, then includes this process into the game on the Pyramid. The desired result of this adaptation is an increase in request processing and therefore, a decrease in the size of the request queue.

The monitor's collection and analysis mechanisms are exercised as follows. For data collection, a traced, extended sensor is embedded into the queue manager's code. This sensor computes the queue's current size from the number of executions of queue element additions and deletions, and it notifies the resident monitor of each change in queue size. The resident monitor checks the current size of the queue against its threshold specified by the adaptation controller, in this case "15." It notifies the central monitor only when the event "threshold exceeded" occurs, as shown in Fig. 3. The sensor is turned on and off by the central and resident monitors in response to commands received from the AC.

The distribution of analysis and collection is straightforward. The analysis required for notification of the central monitor and of the AC regarding the event "threshold exceeded" is performed within the user's code and the resident monitor. As a result, the number of event records to be transferred from the resident to the central monitor is reduced by a factor of roughly fifty, thereby significantly reducing the network message traffic generated by monitoring. Specifically, two local messages and one network message are required to turn on the extended sensor (unless it is initialized to "on"): from AC to central monitor, from central monitor to resident monitor, and from resident monitor to user process. During game execution, the extended sensor generates approximately

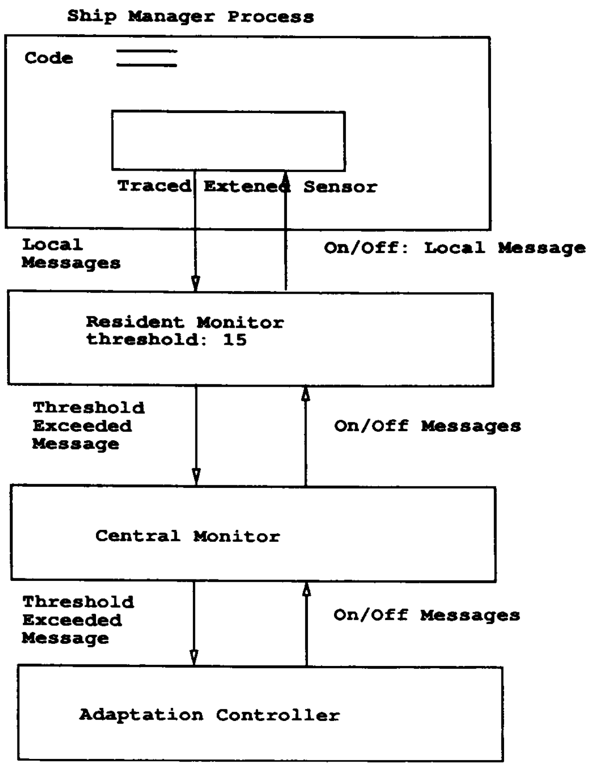


Fig. 3. Monitoring with extended sensors and resident/central monitors.

fifty event records, each recording the addition or deletion of a queue element; these records are sent to the resident monitor as local messages. One message is sent by the resident monitor to notify the central monitor of the event "threshold exceeded." These messages are identified in Fig. 3, as well.

This example shows that the analysis of monitoring information must be distributed and parallelized across the central and resident monitors and the user processes being monitored. The analysis of monitoring information by resident monitors is essential in order to reduce the message traffic within the monitoring system and to reduce the workload imposed on the central monitor. Some analysis may also be shifted to the extended sensor itself. For example, a significant improvement in monitoring performance for this example is gained when the event "threshold exceeded" is computed within the extended sensor itself, so that only a single event record must be transferred from the user program to the resident monitor.

Dynamic monitoring—run-time changes: To demonstrate the system's dynamic variability regarding collection and analysis, and to indicate some tradeoffs between tracing and sampling, we continue monitoring after the addition of a second ship manager, and observe the performance effects of this adaptation. When doing this, the size of the request queue remains stable for some time after the second ship manager is added. However, due to the lack of actual parallelism in the execution of multiple ship managers on the Pyramid, a balance of request generation and processing is not achieved. To be notified of this imbalance, the AC dynamically changes the analysis performed by the resident monitor. In this case, it sets a new value for the queue threshold used by the resident monitor (e.g., from "15" to "20") immediately after addition of the second ship manager. Upon being notified of the event "threshold of 20 exceeded," the AC then slows down request generation by the user process by increasing the amount of time it waits between issuing two consecutive commands from its script. The result of this action is a balance of request generation and processing.

Once this balance is achieved, continued monitoring by tracing queue size using the extended sensor discussed above would be inefficient. As a result, the AC turns this sensor off when it has not been notified of a "threshold exceeded" event for more than 1 min. However, since external conditions, such as changes in Pyramid or Sun loads due to the activities of other users, may change over time, the AC periodically polls the monitor for the queue's size. This polling is achieved by means of a probe.

The additional costs of monitoring in this example derive from two messages due to the AC's dynamic change of the queue threshold to be used for its notification: one local message from AC to central monitor and one message from central to resident monitor, and three messages due to its dynamic deactivation of the sensor: one from the AC to central monitor, one from central to resident monitor, and one from the resident monitor to the user program. The cost of probing after the desired balance has been achieved is small. Each probe consists of one local message from AC to central monitor, one probe request across the network from central to resident monitor, one message from user process to resident monitor reporting the probe value, one return message from resident monitor to central monitor, and one local return message from central monitor to AC.

To summarize, this example suggests that probes are an important element of any dynamic monitoring system that must be able to operate with variable overheads at different times during a program's execution. Furthermore, the use of resident monitors for analysis is essential when such analysis must be changed dynamically, as in the case of the change of threshold values for the extended sensor. A change of the actual user program if such analysis were performed within the extended sensor embedded in this program would be difficult; it would require the full functionality of the dynamic program adaptation systems described in [6], [19]. Finally, run-time program changes are generally not possible if the monitoring system cannot guarantee worst case delays between the occurrence of an event and its collection, analysis, and reporting.

Dynamic monitoring—centralized analysis: It is clear from the simple examples above that a centralized monitor is needed for making dynamic changes in what is being monitored and in how such monitoring is being performed. In addition, certain monitoring queries must be analyzed centrally. For example, if the centralized queue manager process is eliminated from the game, an imbalance of 1) request generation by multiple user processes executing on different network nodes and 2) request processing by multiple, distributed ship managers can be observed only if the generation and processing of requests are monitored locally for each process and then analyzed centrally. Since such central analysis requires that each resident monitor inform the central monitor of each occurrence of request generation and processing, the high cost of such sampling suggests that a less accurate means of information collection should be used in this case. For instance, the AC might sample the sizes of secondary message queues (e.g., UNIX message queues) using probes and make adaptation decisions based on this information.

Summary: In this section, alternative methods for implementation of monitoring have been shown possible based on simple monitoring specifications and on simple changes to those specifications. In addition, it has been shown that users (programs like the AC and human users via a user interface not presented in this paper) can interact with the monitoring system to effect changes in what is being monitored during program execution (e.g., a dynamic change of a queue threshold value from "15" to "20" determined by the AC and enacted by central and resident monitors). The change in the application effected by the AC in response to received monitoring information consisted of "throttling" one of the application's processes. In general, many useful changes can be made to parallel and distributed application programs during their execution, including dynamic process or object migration [8] and the on-line creation of additional processes or the deletion of superfluous processes [6], [28]. Not all such changes are easily performed in the prototype of the Issos system, in part due to restrictions regarding object naming, but those restrictions are not relevant to the evaluation of the basic ideas driving the monitoring system described in this paper.

VI. Related Research

Data collection techniques were at the center of attention in early work on monitoring, including *profiling* in a variety of programming languages (e.g., [13], [54], [73], [75]), which involves collecting execution counts or performing timing at the procedure, statement or instruction level, using sampling or tracing. There have been few advances since the early 1960's when these techniques were first introduced [48], [47]. Techniques using special hardware have been more innovative. Since additional logic imposes no overhead on the computation, capabilities such as event counters, comparators, histogram generators and combinational and sequential logic on events [74], and even reconfiguration based on monitoring can be provided.

Early monitors of multiprocessors and distributed systems emphasized performance evaluation issues and were, in general, confined to the techniques mentioned above [1], [44], [72]. As a result, rudimentary measurement and timing tools are embedded within commercially available multiprocessors, such as the Sequent and Encore machines.

In contrast to early work, we are not assuming hardware support is available for program monitoring. Instead, this research concerns a more integrated approach to monitoring, thereby attempting to facilitate the use of low-level information collection tools. This approach is shared by other recent efforts. For example, interesting user interfaces for some basic performance measurement tools are integrated into the front ends of the BBN Butterfly multiprocessor [11]. A unified set of facilities for monitoring a packet radio network was developed at U.C.L.A. [70]. Similarly, Gertner's system facilitates the monitoring of a distributed system at the message passing level, focusing on message traffic, which is also the case for most of the systems described below.

Another unified set of facilities has been developed by Malony and Reed, who specifically address perturbation and the analysis and display of performance information. They consider both analysis and display of the ''constituent levels'' of a parallel system, i.e., hardware design, system software design, algorithm design and application design. Two ways their approach differs from ours are 1) they identify a minimum set of events that should be captured by signals to a hardware monitor, operating system calls or flags to a compiler preprocessor and 2) they do not build on a uniform information model.

The Computer Network Monitoring System (CNMS) uses a combination of hardware and software to monitor a geographically distributed network [42]. The Jade monitoring system provides a set of tools to observe and control message traffic in a distributed system. This monitor separates the collection and detection of information from the analysis and display of information. However, in contrast to our approach, it does not have access to semantic information about the distributed programs being observed. Message monitoring is also the focus of IDD, which uses a single supervisor process for message analysis, thereby making it a potential bottleneck. IDD does support filtering as a final step by use of a time logic, similar to Snodgrass' temporal query language. IDD also has some graphical capabilities, but does not address the simultaneous and graphical support of the specification of what is to be monitored, the actions to be taken with collected information, and the display of monitoring data. Furthermore, none of the systems mentioned in this paragraph address the monitoring of arbitrary program attributes, as done by the Issos monitor.

Interestingly, some very recent work is now considering hardware support for the dynamic monitoring of arbitrary attributes. The event-based TMP [20] and the dynamic monitoring system described in [68] both assume the use of specialized co-processors for the collection and analysis of dynamic program information. TMP differs from our work in that it is based on the lower level notion of events rather than the program state-based approach offered by the Issos monitor. The work described in [68] solely focuses on the necessary hardware support for the capture of arbitrary program attributes.

Our research also shares some attributes with recent work on distributed or parallel program debugging, which is surveyed in [39]. Such debuggers typically perform a more intrusive form of monitoring. Bates [3] implements dynamic analysis using primitive and high-level *events*, where some *filtering* [65] is performed for high-level events in order to reduce the amounts of information presented to users. However, in contrast to our approach, Bates performs filtering after collection has been completed. In addition, Bates does not allow the association of display information with event descriptions, so that it is hard to customize graphical displays of monitoring information.

Some research on distributed debugging [41] is related to our work in that it is interested in posing higher level questions about a distributed program's execution state. Recent work by such researchers addresses parallel systems, as well. In the IPS-2 system, multiple levels of abstraction concerning the target parallel hardware and application programs are identified. In addition, IPS-2 offers program analysis tools for user assistance in identification of performance-critical program components. Our system would associate such tools with the user interface, again using the information model as the basis for description of the required information. In contrast to our work, IPS-2 can automate the association of instrumentation with a target parallel program by requiring sensors to be associated with procedure calls.

However, research in parallel or distributed debugging typically focuses on asking questions after a program run, and in some instances, programs are instrumented only with respect to interprocess communication [41] (IPS-2 instruments at the procedure level). A program debugger for the BBN Butterfly [31] shares our approach of collecting only the information that is necessary for analysis and presentation. However, this system is more concerned with the replay of parallel programs than with dynamic monitoring. Also, it displays all program information being collected rather than specific views of such information. We conjecture that such an approach does not scale to large parallel machines or to distributed systems due to the excessively large amounts of monitoring information required for nonselective replay. Instead, the collection, analysis, and display of monitoring information should be based on some formalism like the E-R model able to easily manipulate large amounts of information. We share this approach with recent research by Gannon at Indiana [53]. In contrast, recent work by Casavant at Iowa concerns construction of a framework based on event-based formalisms for the automatic generation of application-dependent debuggers much like the Gandalf system is able to generate language-dependent syntax-directed editors [43].

Finally, research addressing the usefulness of dynamic program adaptation is reported elsewhere (see [34], [6] for a definition of the problem and [28], [5], [19], [33], [38] for interesting specific examples or algorithms).

VII. Conclusions and Future Work

This paper describes a system for the specification and the dynamic collection and analysis of program and operating system information in concurrent systems. The monitoring system is itself parallelized and distributed; it consists of resident monitors on each network node, which collect and analyze information local to that node, and a logically centralized monitor, which presents a user interface and correlates and stores distributed information, as necessary. The system's novel attributes include 1) its multiplicity of information collection mechanisms: sensors, extended sensors, and probes, and 2) its use for dynamic or static adaptation of concurrent application programs. The utility of the system is demonstrated with a workload generator program and with the adaptation of a sample parallel (and distributed) program.

A major contribution of this research is a demonstration that an entity-relationship (E-R) model may be used for 1) the description of concurrent software and distributed or parallel hardware, 2) the specification of program views and attributes for monitoring, and 3) the determination of distributed analysis and collection to be performed for the specified views. This paper has focused on using the model for monitoring; other papers discuss information specification and display [27], [26], [62], [60], [59], [19], [45], [25]. Entities, relationships, and their attributes are specified in the program construction system when the parallel application is designed and implemented. Later, views are specified on entities and relationships to describe the desired monitoring information, to be used, for example, for adaptation. The low level distributed collection and analysis mechanisms can then be generated automatically from these high level specifications. The model's utility has been demonstrated in our own research on a wide variety of multicomputers, including a local area network [45], [25], [62], several kinds of multiprocessors, a hypercube multicomputer.

To achieve high performance, monitoring information cannot be collected and analyzed separately, as also shown by recent research regarding the hardware assistance for the monitoring of arbitrary program attributes [68], [20]. In our prototype systems, this is evident from the gains in efficiency attained by the combined analysis and collection performed in extended sensors or resident monitors, where analysis can be pushed to the lowest level at which it may be performed, thereby reducing interprocess and interprocessor communication. Thus, while the specification of what to monitor should be done using some information model describing a concurrent program or system, this specification must then be used to generate efficient information collection and analysis using a diverse set of mechanisms (in the systems presented here, probes, sensors, extended sensors, code in resident monitors, and code in central monitors).

An interesting, resulting limitation of our system is that it achieves high performance by assuming that users specify monitoring queries statically (in contrast to research on post-execution analysis [30]), so that the collection and analysis required for answering such queries may be optimized and compiled "into" the application and monitoring system. This implies that users cannot dynamically ask the system to collect and analyze totally different program information. Furthermore, significant extensions of the system are necessary to deal with objects whose dynamic creation cannot be anticipated by the monitoring system at the time of program compilation.

Our current research continues to focus on dynamic program monitoring, using re-implementations of the monitoring system on a range of parallel machines, including a network of Sun workstations, a BBN Butterfly, a Sequent Symmetry, and a Kendall Square Research multiprocessor. Our current research is concentrating on the graphical display of monitoring information [27], based on the E-R information model, including the use of tools for the generation of program animations [67].

In addition, we are beginning to understand the use of on-line program information for the interactive steering of large-scale scientific applications running on parallel machines.

ACKNOWLEDGMENT

In addition to the authors, R. Ramnath and S. Vasudevan were the other principal designers of the Issos adaptation system, prototypical parts of which were also developed by T. Bihari, M. Choudhary, S. Sarkar, J. Hollingsworth, and W. K. Varadhan. J. Gawkowski was partially responsible for a graphical user interface constructed for the monitor. P. Gopinath and M. Mandal designed, implemented, and measured the multiprocessor implementation of the monitoring system. D. Doerner, F. A. Fisher, E. MacHardy, S. Neuman, and S. Duncan of the University of North Carolina implemented the initial prototype of the Unix-based, resident monitor.

REFERENCES

[1] M.D. Abrams and S. Treu, "A methodology for interactive computer service measurement," *Commun. ACM,* vol. 20, no. 12, pp. 936–944, Dec. 1977.

[2] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe, "The eden system: A technical review," *IEEE Trans. Software Eng.,* vol. SE-11, no. 1, pp. 43–58, Jan. 1985.

[3] P. Bates, "Debugging heterogeneous distributed systems using event-based models of behavior," in *Proc. Workshop Parallel and Distributed Debugging,* Univ. of Wisconsin, Madison, WI, May 1988.

[4] B. Bershad, E. Lazowska, H. Levy, and D. Wagner, "An open environment for building parallel programming systems," in *Proc. ACM SIGPLAN Parallel Programming: Experience with Applications, Languages and Systems, SIGPLAN Notices,* ACM SIGPLAN, Sept. 1988, pp. 1–9.

[5] T. Bihari and K. Schwan "A comparison of four adaptation algorithms for increasing the reliability of real-time software," in *Proc. Ninth Real-Time Syst. Symp.,* Huntsville, AL, IEEE, Dec. 1988, pp. 232–241.

[6] ——, "Dynamic adaptation of real-time software," *ACM Trans. Comput. Syst.,* vol. 9, no. 2, pp. 143–174, May 1991. Older version available from the Dep. Comput. Inform. Sci., Ohio State Univ., OSU-CISRC-5/88-TR, newer version available from College of Comput., Georgia Inst. of Technol., Atlanta GA, GTRC-TR-90/67.

[7] W. Bo "Topologies–Distributed abstract objects in multicomputers," Ph.D. dissertation, Dep. Comput. Inform. Sci., Ohio State Univ., Sept. 1989.

[8] J. S. Chase, F. G. Amador, E. Lazowska, H. Levy, and R. Littlefield, "The amber system: Parallel programming on a network of multiprocessors," in *Twelfth ACM Symp. Operat. Syst. Principles, SIGOPS Notices,* ACM SIGOPS, Dec. 1989, pp. 147–158.

[9] P. P.-S. Chen, "The entity-relationship model—Toward a unified view of data," *ACM Trans. Database Syst.,* vol. 1, no. 1, pp. 9–36, Mar. 1976.

[10] M. Choudhary, "Multi-weight objects and invocations for networks and multiprocessors; A run-time system for COOL," Ph.D. dissertation, M.Sc thesis, Dep. Comput. Inform. Sci., Ohio State Univ., Apr. 1988.

[11] W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken, and T. Blackadar, "Performance measurements on a 128-node butterfly parallel processor," in *Proc. 1985 Int. Conf. Parallel Processing,* Aug. 1985, pp. 531–540.

[12] A. K. Jones, E. F. Gehringer, and Z. Z. Segall, "The cm* testbed," *IEEE Comput. Mag.,* vol. 15, no. 10, pp. 40–53, Oct. 1982.

[13] J.˜Fitch, "Profiling a large program," *Software—Practice and Experience,* vol. 7, pp. 511–518, 1977.

[14] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, *Solving Problems On Concurrent Processors.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

[15] E. F. Gehringer, D. P. Siewiorek, and Z. Segall, *Parallel Processing: The Cm* Experience.* Bedford, MA: Digital Press, Digital Equipment Corp., 1987.

[16] I. Gertner, "Performance evaluation of communicating processes," Ph.D. dissertation, Univ. of Rochester, May 1980.

[17] P. Gopinath, R. Ramnath, and K. Schwan, "Entity-Relationship datase support for real-time applications," in *Proc. PARBASE-90,* ACM, IEEE, and Euromicro, 1990. Also available as Philips Technical Note TN-89-135.

[18] ——, "Database design for real-time systems," *J. Syst. Software,* vol. 17, no. 2), Feb. 1992.

[19] P. Gopinath and K. Schwan, "Chaos: Why one cannot have only an operating system for real-time applications," *SIGOPS Notices,* pp. 106–125, July 1989. Also available as Philips Technical Note TN-89-006.

[20] D. Haban and D. Wybranietz, "A hybrid monitor for behavior and performance analysis of distributed systems," *IEEE Trans. Software Eng.,* vol. 16, no. 2, pp. 197–211, Feb. 1990.

[21] P. Harter, D. Heimbigner, and R. King, "Idd: An interactive distributed debugger," in *Proc. 5th Int. Conf. Distributed Syst.,* IEEE, Mar. 1985.

[22] M. T. Heath and J. A. Etheridge, "Visualizing the performance of parallel programs," *IEEE Software,* vol. 8, no. 5, pp. 29–39, Sept. 1991.

[23] S. E. Hudson and R. King, "The cactis project: Database support for software environments," *IEEE Trans. Software Eng.,* vol. 14, no. 6, pp. 705–719, June 1988.

[24] J. Joyce, G. Lomow, K. Slind, and B. Unger, "Monitoring distributed systems," *ACM Trans. Compu. Syst.,* vol. 5, no. 2, pp. 121–150, May 1987.

[25] M. J. Kaelbling and D. M. Ogle, "Minimizing monitoring costs: Choosing between tracing and sampling," in *Proc. 23rd Int. Hawaii Conf. Syst. Sci., Vol. I,* Jan. 1990, pp. 314–320.

[26] C. Kilpatrick and K. Schwan, "Chaosmon—Application-specific monitoring and display of performance information for parallel and distributed systems," in *Proc. ACM Workshop Parallel and Distributed Debugging,* pp. 57–67. ACM SIGPLAN Notices, vol. 26, no. 12, May 1991.

[27] C. Kilpatrick, K. Schwan, and D. Ogle, "Using languages for describing capture, analysis, and display of performance information for parallel and distributed applications," in *Proc. Int. Conf. Comput. Languages '90,* New Orleans, LA, IEEE, Mar. 1990, pp. 180–189.

[28] J. Kramer and J. MaGee, "Dynamic configuration for distributed systems," *IEEE Trans. Software Eng.,* vol. SE-11, no. 4, pp. 424–436, Apr. 1985.

[29] L. Lamport, "Synchronizing time servers," Tech. Rep. 18, Digital Systems Research Center, Palo Alto, CA, June 1987.

[30] R. LeBlanc and A. Robbins, "Event-driven monitoring of distributed programs," in *Proc. 5th Int. Conf. Distributed Syst.,* IEEE, Mar. 1985.

[31] T. Leblanc and J. Mellor-Crummey, "Debugging parallel programs with instant replay," *IEEE Trans. Comput.,* vol. C-36, Apr. 1988.

[32] T. J. LeBlanc and S. A. Friedberg, "Hierarchical process composition in distributed operating systems," in *Proc. 5th Int. Conf. Distributed Comput. Syst.,* Denver, CO, IEEE, ACM May 1985, pp. 26–34.

[33] K.-J. Lin, S. Natarajan, and J. W. S. Liu, "Imprecise results: Utilizing partial computations in real-time systems," in *Proc. IEEE Real-Time Syst. Symp.,* 1987, pp. 210–217.

[34] J. Magee and J. Kramer, "Dynamic configuration for distributed real-time systems," in *Proc. Int. Conf. Parallel Processing,* IEEE, ACM, Aug. 1983, pp. 277–288.

[35] A. D. Malony, D. Reed, J. W. Arendt, R. A. Aydt, D. Grabas, and B. K. Totty, "An integrated performance data collection, analysis, and visualization system," Tech. Rep. TTR11, Univ. of Illinois at Urbana–Champaign, Mar. 1989.

[36] A. D. Malony, D. A. Reed, and H. A. G. Wijshoff, "Performance measurement intrusion and perturbation analysis," *IEEE Trans. Parallel Distributed Syst.,* vol. 3, no. 4, pp. 433–450, July 1992.

[37] D. C. Marinescu, J. E. Lumpp, T. L. Casavant, and H. J. Siegel, "Models for monitoring and debugging tools for parallel and distributed software," *J. Parallel Distributed Comput.,* vol. 9, no. 2, pp. 171–184, June 1990.

[38] K. Marzullo and M. Wood , "Making real-time systems reactive," *ACM Operat. Syst. Rev.,* vol. 25, no. 1, Jan. 1991.

[39] C .E. McDowell and D. P. Helmbold, "Debugging concurrent programs," *ACM Comput. Surveys,* vol. 21, no. 4, pp. 593–623, Dec. 1989.

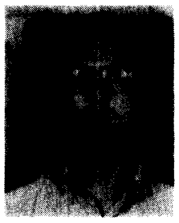
[40] B. P. Miller, M. Clark, J. Hollingsworth, S. Kirstead, S.-S. Lim, and T. Torzewski, "Ips-2: The second generation of a parallel program measurement system," *IEEE Trans. Parallel Distributed Syst.,* vol. 1, no. 2, pp. 206–217, Apr. 1990.

[41] B. P. Miller and C. Q. Yang, "Ips: An interactive and automatic performance measurement tool for parallel and distributed programs," in *Proc. Int. Conf. Distributed Comput. Syst.,* Berlin, Germany, Sept. 1987.

[42] D. E. Morgan, W. Banks, D. P. Goodspeed, and R. Kolanko, "A computer network monitoring system," *IEEE Trans. Software Eng.,* vol. SE-1, no. 3, Sept. 1975.

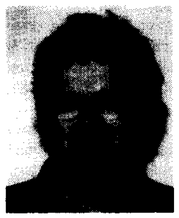
[43] D. Notkin, "The gandalf project," *J. Syst. Software,* vol. 5, no. 2, pp.

91–106, May 1985.
[44] G. J. Nutt, "A survey of remote monitors," Tech. Rep. 500-542, National Bureau of Standards, Jan. 1979.
[45] D. Ogle, "The real-time monitoring of distributed and parallel systems," Ph.D. dissertation, Dep. Comput. Inform. Sci., Ohio State Univ., Aug. 1988.
[46] D. M. Ogle, P. Gopinath, and K. Schwan, "Tool integraton in distributed programming and execution environments—Representing and using monitored information," in *Proc. IEEE Workshop Experimental Distributed Syst.*, Huntsville, AL, IEEE, 1990, pp. 83–90.
[47] B. Plattner, "Real-time execution monitoring," *IEEE Trans. Software Eng.*, vol. SE-10, no. 6, pp. 756–764, Nov. 1984.
[48] B. Plattner and J. Nievergelt, "Monitoring program execution: A survey," *IEEE Comput. Mag.*, vol. 16, no. 11, pp. 76–93, Nov. 1981.
[49] R. Ramnath, "ICE—An environment for constructing and tuning parallel programs," Ph.D. dissertation, Dep. Comput. Inform. Sci., Ohio State Univ., Aug. 1988.
[50] S. P. Reiss, "Pecan: Program development systems that support multiple views," *IEEE Trans. Software Eng.*, pp. 276–284, Mar. 1985.
[51] ——, "Connecting tools using message passing in the field environment," *IEEE Software*, vol. 7, no. 4, pp. 57–67, July 1990.
[52] R. H. Halstead, Jr., "Multilisp: A language for concurrent symbolic computation," *ACM Trans. Programming Languages and Syst.*, vol. 7, no. 4, pp. 501–538, Oct. 1985.
[53] S. R. Sarukkai and D. Gannon, "Parallel program visualization using sieve.1," in *Proc. Int. Conf. Supercomput.*, ACM, July 1992.
[54] E. Satterthwaite, "Debugging tools for high level languages," *Software—Practice and Experience*, vol. 2, pp. 197–217, 1972.
[55] K. Schwan, R. Ramnath, S. Sarkar, and S. Vasudevan, "Cool—Language constructs for constructing and tuning parallel programs," in *Proc. Int. Conf. Comput. Languages*, Miami Beach, FL, IEEE, Oct. 1986, pp. 90–103.
[56] K. Schwan, T. E. Bihari, and B. Blake, "Adaptive, reliable software for distributed and parallel, real-time systems," in *Proc. Sixth Symp. Reliability in Distributed Software*, Williamsburg, VA, IEEE, Mar. 1987, pp. 32–44.
[57] K. Schwan, T. Bihari, B. W. Weide, and G. Taulbee, "High-performance operating system primitives for robotics and real-time control systems," *ACM Trans. Computer Syst.*, vol. 5, no. 3, pp. 189–231, Aug. 1987.
[58] K. Schwan, B. Blake, W. Bo, and J. Gawkowski, "Global data and control in multicomputers: Operating system primitives and experimentation with a parallel branch-and-bound algorithm," in *Concurrency: Practice and Experience*. New York: Wiley, Dec. 1989, pp. 191–218.
[59] K. Schwan and A. K. Jones, "Specifying resource allocation for the cm* multiprocessor," *IEEE Software*, vol. 3, no. 3, pp. 60–70, May 1984.
[60] K. Schwan and J. Matthews, "Graphical views of parallel programs," *ACM SIGSOFT Notices*, 1986.
[61] K. Schwan, R. Ramnath, S. Vasudevan, and D. Ogle, "A system for parallel programming," in *Proc. 9th Int. Conf. Software Eng.*, Monterey, CA, IEEE, pp. 270–282, ACM, Mar. 1987. Awarded best paper.
[62] ——, "A language and system for parallel programming," *IEEE Trans. Software Eng.*, vol. 14, no. 4, pp. 455–471, Apr. 1988.
[63] Z. Segall, A. Singh, R. Snodgrass, A. Jones, and D. Siewiorek, "An integrated instrumentation enivronment for multiprocessors," *IEEE Trans. Comput.*, vol. C-32, no. 1, Jan. 1983.
[64] R. Snodgrass, "The temporal query language tquel," *ACM Trans. Database Syst.*, vol. 12, no. 2, pp. 247–298, May 1987.
[65] ——, "A relational approach to monitoring complex systems," *ACM Trans. Comput. Syst.*, vol. 6, no. 2, pp. 157–196, May 1988.
[66] D. C. Sowell and K. Schwan, "Supporting parallel programming environments with shared persistent data structures," in *Proc. Int. Workshop Unix-Based Software Development Environments*, Dallas, TX, Usenix, Jan. 1991.
[67] J. T. Stasko, "TANGO: A framework and system for algorithm animation," *IEEE Comput. Mag.*, vol. 23, no. 9, pp. 27–39, Sept. 1990.
[68] J. J. P. Tasi, K.-Y. Fang, and H.-Y. Chen, "A noninvasive architecture to monitor real-time distributed systems," *IEEE Software*, vol. 23, no. 3, pp. 11–23, Mar. 1990.
[69] R. N. Taylor, F. C. Belz, L. A. Clarke, L. Osterweil, R. W. Selby, J. C. Wileden, A. L. Wolf, and M. Young, "Foundations for the arcadia environment architecture," in *Proc. ACM SIGSOFT '88: Third Symp. Software Development Environments, SIGPlan Notices, 24/2,* Feb. 1989, pp. 1–13.
[70] D. A. Tobagi, S. E. Lieberson, and L. Kleinrock, "On measurement facilities in packet radio systems," in *Proc. Nat. Comput. Conf.*, AFIPS, 1976, pp. 589–596.
[71] J. D. Ullman *Principles of Database Systems.* Rockville, MD: Computer Science Press, 1989.
[72] D. Vrsalovic, D. P. Siewiorek, Z. Z. Segall, and E. F. Gehringer, "Performance prediction for multiprocessor systems," in *Proc. Int. Conf. Parallel Processing,* IEEE, ACM, Aug. 1984, pp. 139–147.
[73] W. M. Waite, "A sampling for applications programs," *Software—Practice and Experience,* vol. 3, pp. 75–79, 1973.
[74] W. A. Wulf, R. Levin, and P. Harbison, *Hydra/C.mmp: An Experimental Computer System.* New York: McGraw-Hill, 1981.
[75] G. Yuval, "Gathering run-time statistics without black magic," *Software–Practice and Experience,* pp. 105–108, 1975.



David M. Ogle received the B.S., M.S., and Ph.D. degrees in computer and information science from the Ohio State University in 1981, 1984, and 1988, respectively.

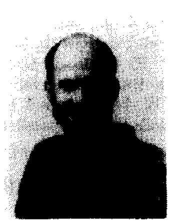
He joined the IBM Corporation, Research Triangle, Park, NC, in 1989 as a member of the Experimental Systems Department. He is actively involved in ad-tech projects in the areas of distributed systems and multiprotocol networking.



Karsten Schwan received the M.Sc. and Ph.D. degrees in computer science from Carnegie-Mellon University.

He is an Associate Professor in the College of Computing, Georgia Institute of Technology. His research addresses operating systems and programming support for parallel machines, including OS kernel structures and tools for program monitoring, tuning, and configuration. While at Ohio State University, he established the Parallel, Real-time Systems (PARTS) Laboratory conducting research focusing on real-time operating systems for parallel machines. At Georgia Institute of Technology, he is also co-director of the High Performance and Parallel Computing Experimentation Laboratory (HPPCEL) providing a 32-node Kendall Square supercomputer.

Dr. Schwan is an associate editor of *Concurrenty—Practice and Experience*.



Richard Snodgrass (S'79–M'81–SM'87) received the Ph.D. degree from Carnegie-Mellon University in 1982.

He joined the University of Arizona in 1989. His research interests include temporal databases and programming environments. He directed the design and implementation of the Scorpion Meta Software Development Environment, described in his book, *The Interface Description Language: Definition and Use* (Rockville, MD: Computer Science Press).

He is an associate editor of the *ACM Transactions on Database Systems* and is on the editorial board of the *International Journal of Computer and Software Engineering*. He will chair the program committee for the 1994 SIGMOD Conference.