Technical Report GIT-CC-94-21

Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs[1]

Weiming Gu, Greg Eisenhauer, Eileen Kraemer, Karsten Schwan
John Stasko, and Jeffrey Vetter

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Abstract – Falcon is a system for on-line monitoring and steering of large-scale parallel programs. The purpose of such interactive steering is to improve its performance or to affect its execution behavior. The Falcon system is composed of an application-specific on-line monitoring system, an interactive steering mechanism, and a graphical display system. In this paper, we present a framework of the Falcon system, its implementation, and evaluation of the system performance. A complex sample application – a molecular dynamics simulation program (MD) – is used to motivate the research as well as to evaluate the performance of the Falcon system.

1 Introduction

The high performance of current parallel supercomputers is permitting users to *interact* with their applications during program execution. Such interactive executions of large-scale parallel codes typically make use of multiple networked machines working in concert on behalf of a single user, as computational engines, display engines, input/output engines, etc. Our research explores the potential increases in performance and functionality gained by the *on-line* interaction of end users with their supercomputer applications. Specifically, we are investigating the *interactive steering* of parallel programs, which is defined as 'the on-line configuration of a program by algorithms or by human users, with the purpose of affecting the program's performance or execution behavior'. Interactive program steering does not involve simply the on-line or postmortem exploration of program trace or output data, as being investigated by researchers in program debugging[28, 18] or in computer graphics[6]. Instead, program steering targets the parallel code itself, and it can range from rapid changes made by on-line algorithms to the implementation of single program abstractions (e.g., a mutex lock [35]) to the user-directed improvement of or experimentation with high-level attributes of parallel codes (e.g., load balancing in a large-scale scientific code – see Section 2.2). In either case, program steering is based on the on-line capture of information about current program and configuration state [7, 31, 46, 40], and it assumes that human users and/or algorithms inspect, analyze, and manipulate such information when making and enacting steering decisions.

[1] This research was supported in part by NASA grant No. NAGW-3886 and with funding from Kendall Square Research Corporation.

Falcon is a system for the *on-line* monitoring and steering of threads-based parallel programs. This paper focusses on Falcon's contributions to program monitoring:

- *Application-specific monitoring* – in addition to providing default program information, Falcon permits users to capture and analyze application-specific program information, ranging from information about single program variables to program states defined by complex expressions involving several program components distributed across different processors of a single underlying parallel machine. These capabilities are especially useful for non-Computer Science end users, who wish to view, analyze, and steer their programs in terms of program attributes with which they are familiar (e.g., 'time step size', 'current energy', etc.).

- *Scalable, dynamically controlled monitoring performance* – by using concurrency and multiple mechanisms for capturing and analyzing monitoring information, the performance of the monitoring system itself can be scaled to different application needs, ranging from high-bandwidth and low-latency event-based monitoring to lower bandwidth sampling of accumulated values. Moreover, the resulting tradeoffs between monitoring latency, throughput, overhead, and accuracy may be varied dynamically, so that monitoring performance may be controlled and adjusted to suit the needs of individual applications and target machines. In addition, simple mechanisms are provided so that users can evaluate program perturbation due to monitoring.

- *On-line analysis, steering, and graphical display* – monitoring information captured with Falcon may be attached to arbitrary user-provided analysis code and subsequent (if desired) steering algorithms and/or graphical views. Analyses may employ statistical methods, boolean operators like those described in [40], or simply reorder the events being received, as described in Section 5.4. Graphical views may be displayed with multiple media or systems, currently including X windows, Motif, and the SGI Explorer environment. In addition, Falcon offers default on-line graphical animations of the performance of threads-based parallel programs. For such Motif-based displays, the Polka system for program animation provides users with easy-to-use tools for creating application-specific 2D animations of arbitrary program attributes[49].

- *Extension to multiple heterogeneous computing platforms* – an extension of Falcon addresses both single parallel computing platforms running threads programs as well as distributed computational engines using PVM as a software basis.

Falcon runs on several hardware platforms, including the Kendall Square Research KSR-1 and KSR-2 supercomputers, the GP1000 BBN Butterfly multiprocessor, the Sequent multiprocessor, SGI workstations, and SUN SPARCstations. Falcon is now in routine use at Georgia Tech by non-Computer Science end users, and it is available for public release for the KSR-1, KSR-2, SGI, and SUN SPARCstation platforms.

In the remainder of this paper, Section 2 presents the motivation for this research by examining the monitoring and steering needs of a sample parallel application, a molecular dynamics simulation (MD) used by physicists for exploring the statistical mechanics of complex liquids. Section 3 presents details of the implementation and performance of the Falcon system itself. The overall performance of the Falcon system as well as its performance with the MD code is evaluated in Section 4. Section 5 examines the nature and requirements of Falcon's graphical displays. Related research is described in Section 6. The final section presents conclusions and future research.

2 Monitoring and Steering a Parallel Code

Program monitoring and steering derive their value from their utilization in understanding and improving program behavior, and in permitting users to experiment with program characteristics that are not easily understood. Clearly, it will be hard to prove that promises of enhanced utility or performance of parallel applications can be fulfilled more easily by steered programs than by non-steered ones. However, it is

inevitable that program steering will be performed in the future, in part because scientists now have available to them the computational and network power for interactive execution of interesting physical simulations and the means for interactive data visualization or even for virtual reality interfaces to their programs. To further motivate our work, this section briefly describes a particular parallel code, its potential for utilizing program steering, and the required support for on-line monitoring.

2.1 The MD Application

MD is an interactive molecular dynamics simulation developed at Georgia Tech in cooperation with a group of physicists exploring the statistical mechanics of complex liquids [51, 8]. In this paper, the physical MD system being simulated contains 4800 particles representing an alkane film and 2700 particles in a crystalline base on which the film is layered. For each particle in the MD system, the basic simulation process takes the following steps: (1) obtain location information from its neighboring particles, (2) calculate forces asserted by particles in the same molecule (*intra-molecular forces*), (3) compute forces due to particles in other molecules (*inter-molecular forces*), (4) apply the calculated forces to yield new particle position, and (5) publish the particle's new position. The dominant computational requirement is calculating the inter-molecular forces between particles, and other important computations include finding the bond forces within the hydrocarbon chains, determining system-wide characteristics such as atomic temperature, and performing on-line data analysis and visualization.

The implementation of the MD application attains parallelism by domain decomposition. That is, the simulation system is divided into regions and the responsibility for computing forces on the particles in each region is assigned to a specific processor. In the case of MD, we can assume that the decomposition changes only slowly over time and that computations in different sub-domains are independent outside some cutoff radius. Inside this radius information must be exchanged between neighboring particles, so that different processes must communicate and synchronize between simulation steps. The resulting overheads are moderate for fairly coarse decompositions (e.g., 100-1000 particles per process), but unacceptable for finer grain decompositions (e.g., 10 particles per process).

2.2 Steering MD – Experimentation and Results

The on-line manipulation of parallel and distributed programs has been shown to result in performance improvement in many domains. Examples include the automatic configuration of small program fragments for maintaining real-time response in uniprocessor systems[32], the on-line adaptation of functional program components for realizing reliability versus performance tradeoffs in parallel and real-time applications [5, 14, 12], and the load balancing or program configuration for enhanced reliability in distributed systems[26, 43, 31].

The MD simulation offers opportunities for performance improvement through on-line interactions with end users and with algorithms, including:

- Decomposition geometries can be changed to respond to changes in physical systems. For example, a slab-based decomposition is useful for an initial system, but a pyramidal decomposition may be a better choice if a probe is lowered into the simulated physical system.
- The interactive modification of cutoff radius can improve solution speed by computing uninteresting time steps with some loss of fidelity, which typically requires the involvement of end users.
- The boundaries of spatial decompositions can be shifted for dynamic load balancing among multiple processes operating on different sub-domains, performed by end users or by a configuration algorithm.
- Global temperature calculations, which are expensive operations requiring a globally consistent state, can be replaced by less accurate local temperature control. On-line analysis can determine how often global computations must be performed based on the temperature stability of the system.

To demonstrate the utility of program steering, we next review some results of interactive MD steering applied to the problem of improving system load balance. In particular, we examine the behavior of the MD simulation when spatial domain of the physical system is decomposed vertically. In this situation, it is quite difficult to arrive at a suitable load balance when decomposing based on static information (such as counting the number of particles assigned to each process, etc.). This is because the complexity of MD computation depends not only on the number of particles assigned to each process, but also on particle distances (due to cutoff radius). Furthermore, the portions of the alkane film close to the substrate are denser than those on the top and therefore require more computation. In fact, fairly detailed modeling of the code's computation is required to determine a good vertical domain decomposition without experimentation, and there is no guarantee that an initial 'good' decomposition will not degrade over time due to particle movement or other changes in the physical system. As a result, it appears easier to simply monitor load balance over time and then steer the application code to adjust load balance (by adjusting domain boundaries) throughout the application's execution. In this paper, such steering is performed interactively by end users. Necessary algorithmic support will be developed in the future; it will enable users to interact with the application only when automated steering is not successful.

For interactive steering of MD, the Falcon system is used to monitor process loads on-line, the resulting trace information is analyzed, and workloads are displayed in bar graph form (see Figure 1). In addition, the MD code performs on-line visualization of particles and of current domain boundaries. The load balance view of Falcon and the MD system's data displays are depicted in Figures 1 and 2, respectively, for a sample simulation run with four domains on four processors. Associated with these displays is a textual user interface (also part of Falcon) that permits the user to change selected program attributes (in this case, shift individual domain boundaries) while the application is running.

The effects of dynamic steering when used to correct load imbalances can be quite dramatic, as shown in Figure 3. In this figure, several steering actions significantly improve program performance by successive adjustment of domain boundaries. These results are important for several reasons. First, they demonstrate that it is possible to improve program performance by use of on-line steering, rather than degrade performance due to steering and monitoring costs. Second, it should be apparent that user interactions with the code can be replaced or assisted by on-line steering algorithms, in effect giving users the ability to migrate their experiences and experimental knowledge into their application codes, without requiring extensive program changes. Third, and more broadly, these results indicate the potential of on-line steering for helping end users experiment with and understand the behavior of complex scientific codes.

2.3 The Requirements of Steering

While the steering of MD code by adjustment of domain boundaries as presented in Section 2.2 is straightforward, important to our work are the future opportunities presented by on-line steering and monitoring. Toward this end, our group is now experimenting with interactive parallel programs in several domains, including (1) the interactive simulation of complex systems used in conjunction with some physical system, for on-line diagnosis of problems or for trying out certain fault containment strategies[13] (e.g., telecommunication systems), and (2) the on-line experimentation with scientific or engineering applications. For example, we are developing an interactive global atmospheric modeling code, where scientists can easily experiment with alternative values for atmospheric quantities to adjust model runs in accordance with actual measured atmospheric data obtained from satellite observations (e.g., concentrations of certain pollutants or strengths and directions of wind fields). Similarly, we are using on-line steering to give users the ability to interact with their large-scale optimization codes, to direct program searches out of local minima, to detect and correct searches possibly leading to infeasible solutions, etc.

To realize on-line program steering, several assumptions must be made, some of which may be removed or ameliorated by our future work. First, program steering requires that application builders must write their code such that steering is possible. Second, users must provide the program and performance information necessary for making steering decisions. Third, it is imperative that such information can be obtained with the latency required by the desired rate of steering. Concerning the first requirement, in the MD code,

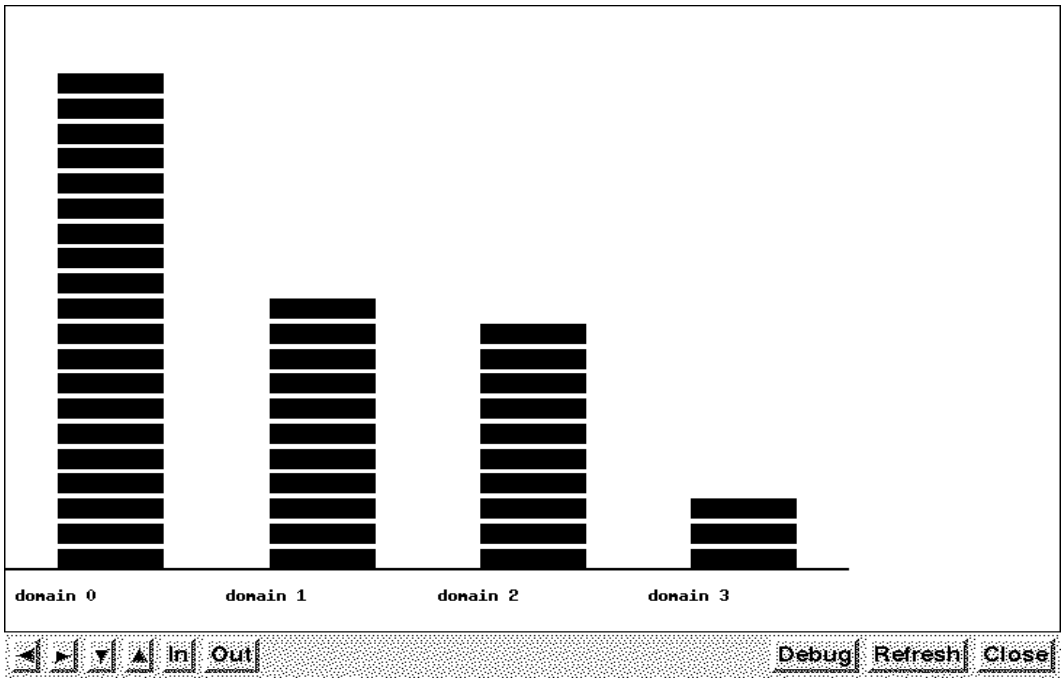
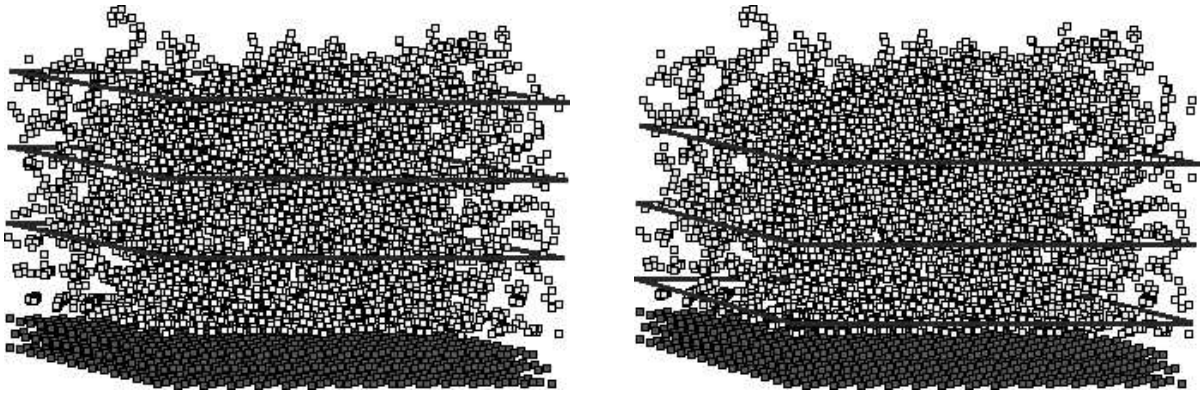


Figure 1: The load balance view of MD.



(a) Initial Decomposition

(b) Balanced Decomposition

Figure 2: Initial and balanced decompositions of the steered system. The horizontal frames mark the boundaries between processor domains. The dark particles are the fixed substrate while the lighter particles are the alkane chains.

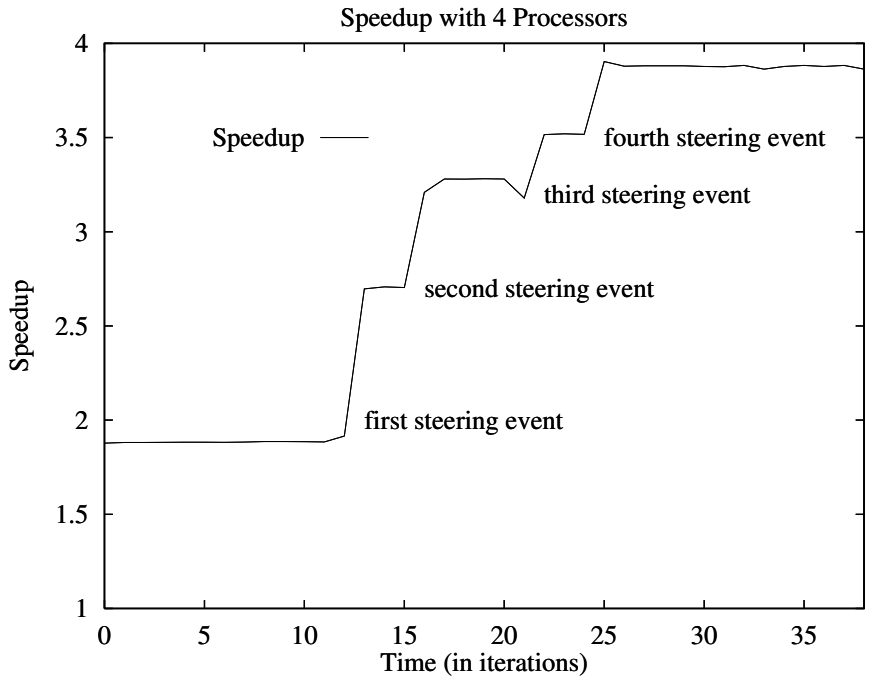


Figure 3: The effect of steering on performance over time with 4 processors.

domains are represented such that their boundaries are easily shifted to make steering for improved workload balance possible. In general, however, programs can be made steerable only by requiring end users to write them accordingly, by requiring substantial compiler support[46], or by requiring that the programming language offer stronger mechanisms of abstraction than those existing in parallel Fortran or in the Cthreads library used in our work (e.g., the object model [5, 11, 26, 14]). We are currently designing higher level language primitives for definition of steering actions and for inclusion of such actions with application code. At this time, however, Falcon relies on user-directed inclusion of *actuators* with the application code. These actuators are then stored into a runtime library which serves as a catalogue of names as well as an interface to the Falcon's on-line monitoring mechanism (see Section 3.4 for a description and brief evaluation of the steering library).

One of the primary concerns of this paper is the second requirement for on-line steering: the on-line provision, analysis, and display of information to users about current program behavior and performance, at rates suitable for program steering. Examples of such information used in graphical displays include the on-line data visualizations depicting molecular distributions in MD, the associated current values of domain boundaries (see Figure 2), and performance information about threads depicted in graphical views like the thread life-time view shown in Figure 13. Examples of such information used by on-line steering algorithms include lock contention values, which are used by on-line configuration algorithms to adjust individual mutex locks (see [35]) based on changes in a program's locking pattern.

A third requirement of on-line steering is that steering is effective only if it can be performed at a rate higher than the rate of program change. In the case of load balancing by dynamic domain shifting in MD, human users can detect load imbalances and shift domain boundaries faster than the rate of occurrence of significant particle movements (which require several minutes for moderate size physical simulations on our KSR-2 machine). However, when steering is used to dynamically adjust lock waiting strategies, changes in locking patterns must be detected and reacted upon in every few milliseconds[35]. As a result, any on-line monitoring support for program steering must permit users to realize suitable tradeoffs in the bandwidth versus latency of monitoring.

In response to the requirements listed above, Falcon gives users the ability to control instrumentation by permitting them to explicitly include program-specific *sensors* of different types into their application codes. A sensor definition language generates sensor implementations for target C and Fortran programs, and runtime-configurable monitoring libraries capture, analyze, and store/forward or display sensor outputs as desired by users. In addition, Falcon offers efficient system I/O (for data visualizations) and underlying communications across computer networks (for all remote mechanisms).

The description and evaluation of on-line monitoring in Falcon is the primary focus of this paper. However, to demonstrate the usability of Falcon, we also briefly describe and evaluate Falcon's interfaces to program animation and graphical data rendering tools.

3 The Design and Implementation of Falcon

3.1 Design Goals

Past work in program monitoring has focussed on helping programmers understand the correctness or performance of their parallel codes[33, 41], on minimizing or correcting for program perturbation due to monitoring[30], on reducing the amounts of monitoring or trace information captured for parallel or distributed program debugging[40], and on the effective replay[28] or long-term storage[47] of monitoring information.

Falcon has three important attributes. First, Falcon supports the *application-specific monitoring/steering, analysis, and display* of program information, so that users can capture, process, and understand and steer exactly the program attributes relevant to steering or to the specific performance problems being diagnosed or investigated. That steering requires application-specific program information is clearly demonstrated by the MD application steered in Section 2.2, where program variables capturing domain boundaries are adjusted based on monitoring output describing workload in terms of durations of molecular computations across different domains. Section 4 will also demonstrate that such specialization of monitoring to capture only specific program attributes can also significantly improve monitoring system performance and scalability compared to standard tools like GProf or compared to the default monitoring performed by Falcon.

Second, the primary focus of Falcon is to *reduce or at least control monitoring latency* throughout the execution of a parallel program, while maintaining acceptable monitoring workload imposed on the underlying parallel machine. Dynamic control of monitoring overhead is important because the effectiveness of program steering can depend on the delay between the time at which a program event happens and the time at which the event is noted and acted upon. In addition, excessive monitoring overheads not only offset performance gains achieved by steering, but also alter the order of occurances of program events. Finally, for scalability to large-scale parallel machines and programs, the Falcon system is configurable in its offered total performance and associated resource usage.

A third attribute of Falcon is its support for *scalable monitoring*, by varying the resources consumed by its runtime system in accordance with machine size and program needs. In Section 4, we show that Falcon can be used to monitor programs of any size running on our 64-node KSR multiprocessor, such that monitoring overheads and latencies can be adjusted in conjunction with program and machine size.

3.2 System Design

Falcon is constructed as a toolkit that collectively supports the on-line program monitoring and steering of parallel and distributed programs. There are four major conceptual components, as shown in Figure 4: (1) monitoring specification and instrumentation, which consists of a low-level *sensor specification language*, higher level *view specification constructs*, and an instrumentation tool, (2) runtime libraries for information capture, collection, filtering, and analysis, (3) mechanisms for program steering, and (4) a graphical user interface and several graphical displays of program behavior and performance information.

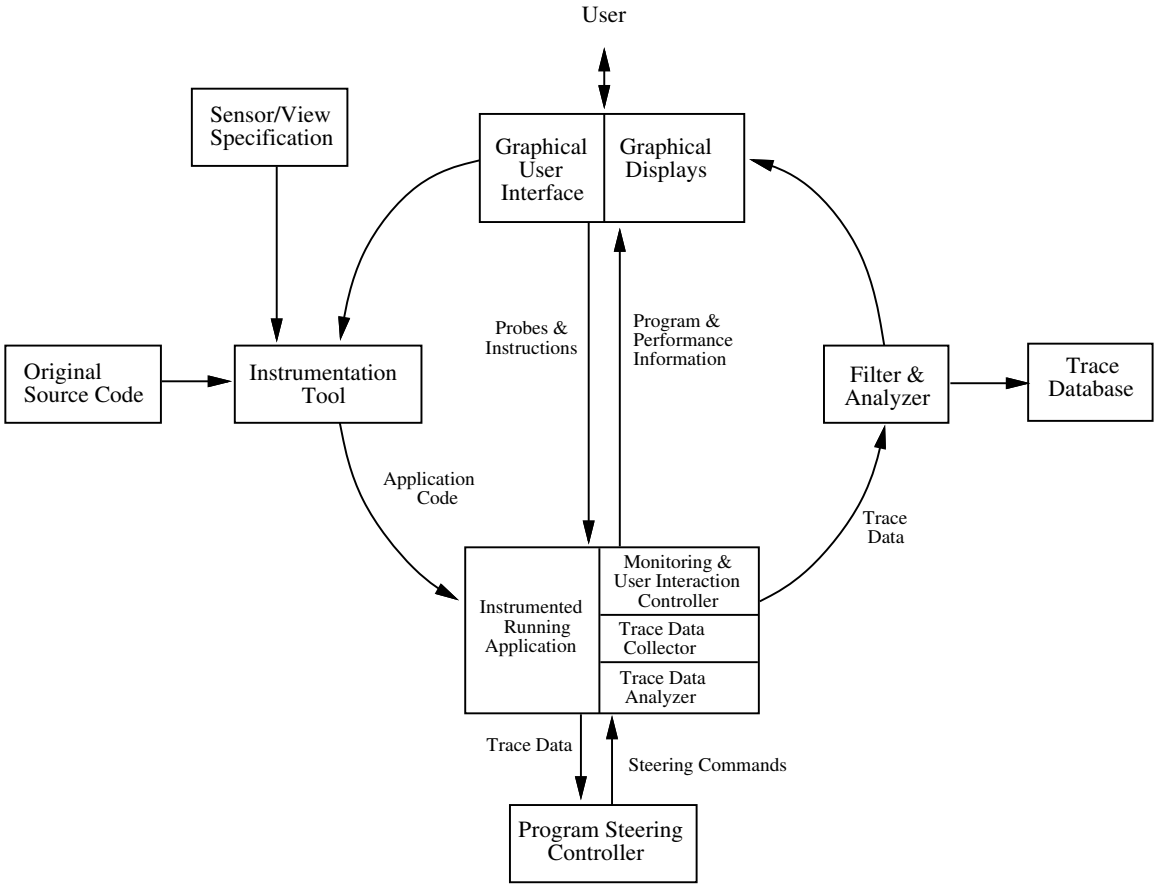


Figure 4: Overall architecture of Falcon.

The following steps are taken when using Falcon. First, the application code is instrumented with the sensors and probes generated from sensor and view specifications. Toward this end, monitoring specifications allow users to expose specific program attributes to be monitored and based on which steering may be performed. User programs and/or Falcon's user interface or analysis/steering algorithms directly interact with the runtime system in order to gain access to information about runtime-created sensor and actuator instances. When the application is running, program and performance information of interest to the user and to steering algorithms is captured by the inserted sensors and probes, and is collected and partially analyzed by Falcon's runtime monitoring facilities. These facilities essentially consist of monitoring data output queues attaching the user program being monitored to a variable number of additional components performing steering and low-level processing of monitoring output (discussed in detail in Section 3.3 below). Partially processed monitoring information is then fed to steering mechanisms for effecting on-line changes to the program or to its execution environment; or it is fed to the central monitor and graphical displays for further analysis and for display to end users. Trace information can also be stored in a trace data base for postmortem analyses.

The monitoring, steering, and user interaction 'controllers', as part of the Falcon runtime system, activate and deactivate sensors, execute probes or collect information generated by sampling sensors, maintain a directory of program steering attributes, and also react to commands received from the monitor's user interface. For performance, these controllers are physically divided into several *local monitoring controllers* and a *steering controller* residing on the monitored program's machine so that they are able to rapidly interact with the program. In contrast, the *central monitoring and steering controller* is typically located on

a front end workstation or on a processor providing user interface functionality.

Falcon uses the Polka system for the construction and use of graphical displays of program information[49]. Several performance or functional views (e.g., the aforementioned bargraphs and thread visualizations) have been built with this tool. However, in order to attain the speeds required for on-line data visualization and to take advantage of other performance display tools, Falcon also interfaces to custom displays and to systems for the creation of high-quality 3D visualizations of program output data, like the SGI Explorer tools.

3.3 System Implementation

Falcon's implementation relies on a Mach-compatible Cthreads library[36] available on several hardware platforms, including the Kendall Square Research KSR-1 and KSR-2 supercomputer, the GP1000 BBN Butterfly multiprocessor, the Sequent multiprocessor, and uni- and multi-processor SGI and SUN SPARC workstations. Figure 5 depicts the system's implementation. It is discussed next in the context of the

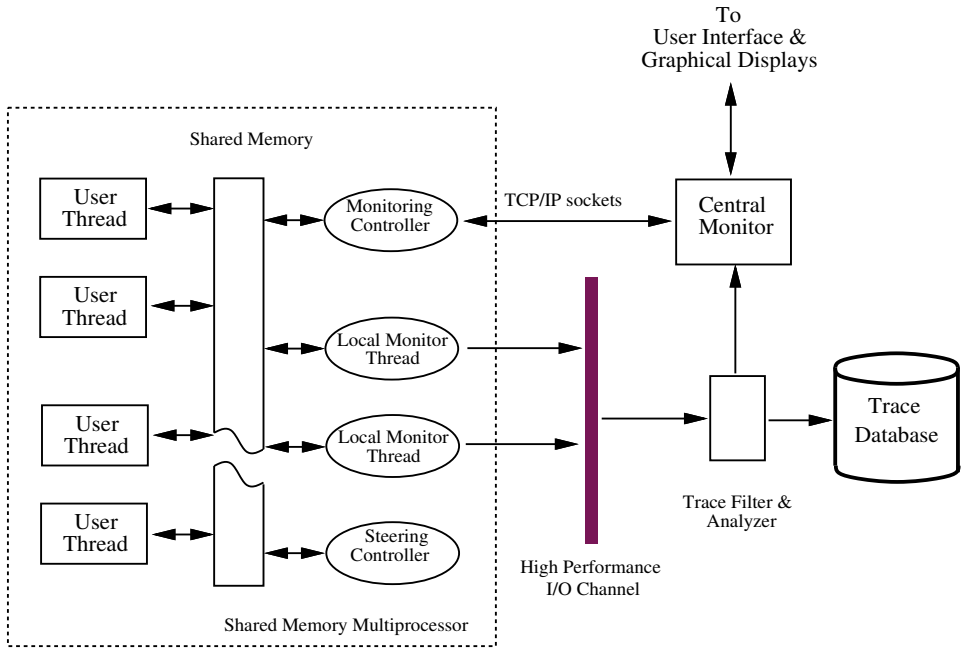


Figure 5: Implementation of the monitoring mechanism with Cthreads.

basic contributions of Falcon to the monitoring literature: (1) low monitoring latency and varied monitoring performance, also resulting in system scalability, (2) the ability to control monitoring overheads, and (3) the ability to perform application-specific monitoring and on-line analyses useful for steering algorithms and graphical displays.

Application-specific monitoring – sensors and sensor types. Using a simple specification language, programmers may define application-specific *sensors* for capturing (a) the program and performance behaviors to be monitored and (b) the program attributes based on which steering may be performed. The specification of a tracing sensor is shown in Figure 6. It simply describes the structure of the application data to be contained in the trace record generated by this sensor. From this declaration is generated the sensor subroutine shown in Figure 7. The body of this subroutine generates entries for an event data structure, then writes that structure into a buffer. A local monitoring thread later retrieves this structure from the buffer. Each sensor's code body is also surrounded by an `if` statement, so that the sensor can be turned on or off during program execution (ie., the monitoring system itself may be dynamically steered).

```
sensor work_load {
        attributes {
                int     domain_num;
                double  work_load;
        }
};
```

Figure 6: Specification of sensor work_load.

```
int
user_sensor_work_load(int process_num, double work_load)
{
    if (sensor_switch_flag(SENSOR_NUMBER_WORK_LOAD) == ON) {
        sensor_type_work_load  data;
        data.type = SENSOR_NUMBER_WORK_LOAD;
        data.perturbation = 0;
        data.timestamp = cthread_timestamp();
        data.thread = cthread_self();
        data.process_num = process_num;
        data.work_load = work_load;

        while (write_buffer(get_buffer(cthread_self()), &data,
                    sizeof(sensor_type_work_load)) == FAILED) {
            data.perturbation = cthread_timestamp() - data.timestamp;
        }
    }
}
```

Figure 7: Generated code of sensor work_load.

Figure 6 shows the specification of the tracing sensor that monitors the workload of each domain partition in MD, and Figure 7 depicts the generated sensor code. There are four *implicit fields* for any event record that describe the event's sensor type, timestamp, thread id, and perturbation. The purpose of the *perturbation field* is to record the additional time spent by the sensor waiting on a full monitoring buffer, if any. This 'buffer full' information is important for generating comprehensible execution time displays. A more detailed explanation of this problem appears with the discussion of Figure 13 in Section 5.3.

It is important to realize that each single sensor specification generates an event type; but its corresponding sensor code may be inerted to many different places within a single parallel program. Moreover, since new threads can be forked during an application's execution time, sensor instances are dynamic. The monitoring system identifies such dynamically created sensors using a combination of thread identifier and sensor type. In addition, users may explicitly register individual *instrumentation objects*, which correspond to specific calls to sensor code made by the target program. Such registration gives the monitoring system the ability to control (e.g., turn on or off) single invocations of sensor code instead of controlling all instances of a certain type of sensor as a whole.

Controlling monitoring overheads – sensor types and sensor control. The monitoring overheads experienced with sensor invocations may be controlled by use of different sensor types: sampling sensors, tracing sensors, or extended sensors. A *sampling sensor* simply writes its output into a structure located in shared memory periodically accessed by the monitor's runtime components also resident on the parallel machine, called *local monitoring threads*. A *tracing sensor* generates timestamped event records that may be used immediately for program steering or stored for postmortem analysis. In either case, trace records are stored in *trace queues* from which they are removed by local monitoring threads. Last, an *extended sensor* performs simple analyses before producing output data, so that some data filtering or processing required

for steering may be performed prior to output data generation. It is evident that sampling sensors inflict less overhead on the target application's execution than tracing and extended sensors. However, as shown in Section 4, the more detailed information collected by tracing sensors may be required for diagnosis of certain performance problems in parallel codes. Furthermore, the combined use of all three sensor types may enable users to balance low monitoring latency against accuracy requirements concerning the program information required for program steering.

Monitoring overheads may be controlled during each program run by direct interaction of user programs and/or Falcon's user interface and/or analysis/steering algorithms with the monitor's runtime system. First, sensors can be turned on or off during the application's execution[47]. Second, sensors can dynamically adjust their own behavior to continously control overall monitoring overhead and latency. For example, a tracing sensor that monitors a constantly accessed mutex lock can reduce its tracing rate to every five mutex lock accesses, thereby improving monitoring perturbation at the cost of reducing trace accuracy. In this paper, we use such dynamic sensor configuration for *selective monitoring* of a parallel program, where during a single program run, different monitoring methods are employed at different points in time. This is attained by enabling or disabling specific sensors, by switching from sampling to tracing sensors, and by changing the behavior of individual sensors (e.g., sensor sampling rates). Experimentation described in Section 4 will demonstrate the utility of selective monitoring with the MD code.

Controlling monitoring overheads – concurrent monitoring and steering. As depicted in Figure 5, local monitoring and steering threads perform trace data collection, processing, and steering concurrently and asynchronoulsy with the target application's execution. Local monitors and steering controllers typically execute on the target program's machine; but they may run concurrently on different processors, using a buffer-based mechanism for communication between application and monitoring threads.

An alternative approach performs all monitoring activities, including trace data capture, collection, and analyses, in the user's code. One problem with this approach is that the target application's execution is interrupted whenever a monitoring event is generated and processed, and the lengths of such interruptions are arbitrary and unpredictable if complicated on-line trace analyses are used. In contrast, the only direct program perturbation caused by Falcon is the execution of embedded sensors and the insertion of trace records into monitoring buffers. Such perturbation is generally predictable (results on the KSR-2 are presented in Section 4), and its effects on the correctness of timing information can be eliminated using straightforward techniques for perturbation analysis [30].

Falcon's runtime system itself may be configured (steered) in several ways, including disabling or enabling sets of sensors, varying activation rates, etc. One such on-line variation explored in detail in this paper is changing the number of local monitoring threads and communication buffers to configure the system for parallel programs and machines of different sizes. Such changes permit selection of suitable monitoring performance for specific monitoring and steering tasks, and they may be used to adapt the monitoring system to dynamic changes in workload imposed by the target application. For example, when heavy monitoring is detected by a simple monitor-monitor mechanism, new local monitors may be forked. Similarly, when bursty monitoring traffic is expected with moderate requirements on monitoring latency, then buffer sizes may be increased to accommodate the expected heavy monitoring load. Such parallelization and configuration of monitoring activities is achieved by partitioning user threads into groups, each of which is assigned to one local monitor. When a new application thread is forked, it is added to the local monitor with the least amount of work.

On-line analysis and display. Monitoring information partially processed by local monitors can be fed to Falcon's steering mechanism to effect on-line changes to the program and its execution environment. It can be sent to Falcon's central monitor for further analysis and for display of program behavior and application performance to end users. It can be stored in a trace data base for postmortem analysis. The central monitor, user interface, graphical displays, and trace database may reside on a different machine to reduce interference from monitoring activities to the target application's execution, and to capitalize on efficient graphics hardware and libraries existing on modern workstations. Section 5 describes some on-line analysis typically required for the on-line display of monitoring information: the need to reorder information produced by Falcon prior to its presentation to users. Falcon's interfaces to systems for the creation of high-quality

3D visualizations of program output data are out of scope of this paper. For the MD application, custom visualizations were constructed in order to gain the speeds required for on-line data viewing and steering.

3.4 On-line Steering Mechanisms

As described in Sections 1 and 2, program steering requires functionality in addition to that being offered by Falcon's monitoring components. Falcon's on-line steering component is a natural extension of its monitoring facilities. Similar to local and central monitors, steering is performed by a *steering server* on the target machine and a *steering client* providing user interface and control facilities. The steering server is typically created as a separate execution thread to which local monitors forward only those monitoring events that are of interest to steering activities. Such events tend to be a small proportion of the total number of monitoring events, in part because simple event analysis and filtering is done by local monitors rather than by the steering server. Steering decisions, then, are made based on specific attributes of those events, by human users (interactively) or by steering algorithms.

Falcon's steering system permits users to implement on-line control systems that operate on and in conjunction with the programs being steered. As a result, the primary task of each steering server is to read incoming monitoring events and then 'decide' what actions to take, based on previously encoded decision routines and actions, both of which are stored on a steering event database which is part of the server. This database contains entries for each type of steering event, where each event may either perform some actual steering action on the parallel program or simply note the occurrence of some monitoring event for future use in steering or for inspection by users from the client's user interface. Accordingly, the secondary task of each steering server is to interact with the remote steering client. The steering client is used to enable/disable particular steering actions, display and update the contents of the steering event database, and input direct steering commands from end users to the server. The steering client is not addressed by the target (on the parallel machine) performance measurements shown below. Its functionality and performance are discussed in more detail elsewhere.

At the lowest level of abstraction, a steering action that modifies an application is either a *probe* or an *actuator*. A probe updates a specific program *attribute* asynchronously to the program's execution. These attributes are defined by application programmers in an object-oriented fashion, where each specific program abstraction can define one or multiple attributes and then export methods for operating on these attributes. The steering event database lists all steerable objects and their program attributes. Also, actions are stored in the database with each event type. Actions are defined methods able to operate on the attributes of these objects. A complete object-oriented framework for defining and operating on program attributes is defined in [38]. The definition and dynamic adjustment of operating system level attributes is described in [35]. For purposes of this paper, the reader should assume that such attributes correspond to specific program variables (ie., to specific locations in the program's data). The steering server uses probes to update such variables at any time it chooses, and it uses actuators to have the program's execution threads enact certain steering actions on its behalf. Such actuators may also execute additional functions to ensure that modifications of program state do not violate program correctness criteria[5].

The performance of steering is assessed in Section 4.5 below.

4 System Evaluation

To understand the performance of the Falcon monitoring system, we evaluate its implementation on a Kendall Square Research KSR-2 parallel machine[2]. This machine has 64 processors interconnected by two rings. The KSR-2 supercomputer is a NUMA (non-uniform memory access) shared memory, cache-only

[2] The 64 node KSR-2 machine at Georgia Institute of Technology was upgraded from a 64 node KSR-1 during our experimentations. Therefore, some of the results presented in this paper are obtained on the KSR-1 machine, while others are obtained on the KSR-2. Programs running on the KSR-2 are roughly twice as fast as those running on a KSR-1 due to differences in machine clock speeds.

architecture with an interconnection network that consists of hierarchically interconnected rings, each of which can support up to 32 nodes. Each node consists of a 64-bit processor, 32 MBytes of main memory used as a local cache, a higher performance 0.5 Mbyte sub-cache, and a ring interface. CPU clock speed is 20 MHz on the KSR-1 and 40 MHz on the KSR-2, with a peak performance of 20 and 40 Mflops per node for KSR-1 and KSR-2, respectively. Access to non-local memory results in the corresponding cache line being migrated to the local cache, so that future accesses to that memory element are relatively cheaper. The parallel programming model implemented by KSR's OSF Unix operating system is one of kernel-level threads which offer constructs for thread fork, thread synchronization and shared memory between threads. This kernel-level thread facility is called Pthreads. Falcon itself employs Cthreads, a user-level threads facility that is built on top of Pthreads.

In the remainder of this section, we first evaluate the basic performance of Falcon's monitoring mechanisms, including measurements of the average costs of tracing sensors and of minimal and expected monitoring latencies. Next, using the MD code, we evaluate Falcon's ability to control monitoring overheads and to scale to different performance requirements. The overheads incurred by individual elements of the runtime steering library are evaluated last.

4.1 Sensor Performance

The perturbation, latency, and throughput of sensors depend on three factors: (1) the size of the event data structure, (2) the cost of event transmission and buffering from sensors to local monitors, and (3) sensor type. A tracing sensor generating a 'large' event containing many user-defined and implicit attributes will execute longer than one generating a 'small' event. Event transmission and buffering costs are affected by a variety of factors, including the number of event queues and local monitor threads, and the actual event processing demands placed on local monitors. Factor (1) is evaluated in Table 1, which depicts the basic costs of executing a sensor modulo its size, where basic costs include: (a) accessing the sensor switch flag, (b) computing the values of sensor attributes, and (c) writing the generated sensor record into an event queue. The table displays measured execution times on a KSR-2 machine.

Event record length	32 bytes	64 bytes	128 bytes
Cost (microseconds)	6.8	7.9	9.6

Table 1: Average cost of generating a sensor record on the KSR-2.

The results in Table 1 indicate that the direct program perturbation caused by inserted sensors should be acceptable for many applications for moderate amounts and rates of monitoring. Specifically, if an application can tolerate from 5% to 10% perturbation, then Falcon's monitoring mechanism can produce monitoring events at a rate from 7,500 to 15,000 events per second on the application's critical execution path. Given these costs, total perturbation of a parallel program can be derived as the cumulative cost of generating all of the sensor records in the program's critical path. A more complex perturbation model is required when considering side effects of such direct program perturbation[30].

The dominant factor in sensor execution is the cost of accessing the buffer shared between application and monitoring threads. The use of multiple monitoring buffers (one per user thread) in Falcon reduces the contention of buffer access by user and monitoring threads, so that the effective cost of buffer access is the cost of copying a sensor record to the buffer. This latter cost depends on the size of the sensor record, as clearly evident from the measurements in Table 1. It should be noted that these costs do not include perturbation that might be caused by bottlenecks in the processing and transmission of the events (which would result in delays in obtaining buffer space). However, such worst case perturbation may be avoided by making dynamic monitoring adjustments provided by Falcon's runtime monitoring mechanisms, such as turning off non-critical sensors, reducing a sensor's tracing rate, forking new local monitoring threads, etc.

4.2 Monitoring Latency and Perturbation

Monitoring latency is defined as the elapsed time between the time of sensor record generation and the time of sensor record receipt and (minimal) processing by a local monitoring thread. Low latency implies that steering algorithms can rapidly react to changes in a user program's current state[37]. Monitoring latency includes the cost of writing a sensor record to a monitoring buffer, the waiting time in the buffer, and the cost of reading the sensor record from the monitoring buffer. While the reading and writing times can be predicted based only on sensor size, the event waiting time in the monitoring buffer depends on the rate at which monitoring events can be processed by local monitors.

Buffer size	Record length		
(bytes)	32 bytes	64 bytes	128 bytes
256	69	73	87
1,024	68	71	84
4,096	68	70	83
16,384	69	73	85

Table 2: Minimum monitoring latency (in microseconds) on the KSR-2.

Buffer size	Record length		
(bytes)	32 bytes	64 bytes	128 bytes
256	164	181	242
1,024	201	264	294
4,096	211	277	498
16,384	256	347	556

Table 3: Latency at moderate monitoring rates (in microseconds) on the KSR-2.

Tables 2 and 3 depict the results of two experiments with a synthetic workload generator instrumented to generate sensor records of size 32 bytes at varying rates, using a single local monitoring thread. In Table 2, monitoring latency is evaluated under low loads, resulting in an approximate lower bound on latency. Results vary with event record sizes, but demonstrate the independence of monitoring latency on the size of the monitoring buffer at low loads. Table 3 uses higher monitoring loads[3] and experimentally demonstrates the expected result that larger monitoring buffers reduce program perturbation, but also increase monitoring latency for buffered events. Specifically, latency is not affected by buffer size at low rates, but increases with increasing buffer sizes even at moderate monitoring rates. This would indicate the use of smaller buffers. However, program perturbation can be larger with small buffers since programs must wait until buffer space is available when attempting to produce an event. Figure 8 demonstrates that the maximum event processing rate of a single local monitoring thread is about 40,000 to 45,000 events per second on the KSR-2 (assuming no significant processing of events in the local monitoring thread). However, monitoring latency remains acceptable when the monitoring rate is less than this saturation point.

The bottleneck due to limitations on the processing ability of single local monitors can be remedied by use of parallelism. Figure 9 shows that monitoring delay is reduced when multiple local monitors are used

[3] The measurements in Table 3 use a monitoring rate of approximately 40,000 events per second, which almost saturates the

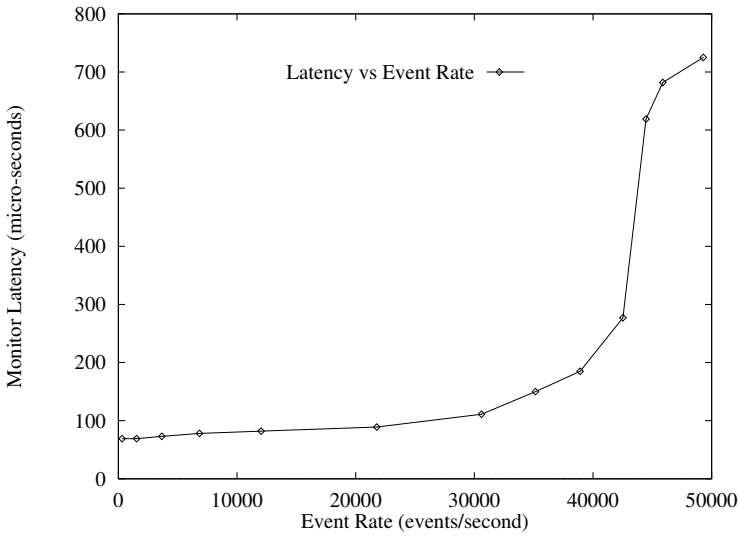


Figure 8: Monitoring latency versus event rate on the KSR-2.

to monitor the MD application. In this experiment, all procedure calls to the Cthreads library are traced. As MD runs on more processors, the frequency of calls to the Cthreads library increases, resulting in higher event rates. It is evident from the results shown in Figure 9 that additional local monitors are effective in reducing monitoring delay when this delay exceeds some threshold (around 200 microseconds for the MD code on the KSR-2). Below this threshold, the additional overheads associated with multiple vs. single local monitoring threads prevents their effectiveness.

In general, the measurements shown in Tables 2 and 3 and in Figures 8 and 9 demonstrate that there exists no general means of attaining both low monitoring latency and perturbation at arbitrary rates of monitoring (other than using additional hardware support). The approach taken by Falcon toward addressing this problem is simply to permit the configuration of the monitoring system itself (buffer sizes, number of trace buffers, number of local monitoring threads, and attachment of monitoring threads to buffers – monitoring load distribution) to offer the performance characteristics desired by the application program. Such configuration can be performed dynamically in a fashion similar to on-line program steering, where the saturation points for local monitors may be used as triggers for configuring the monitoring system itself.

4.3 Monitoring the MD Code

This section demonstrates the overall performance and utility of Falcon's monitoring mechanisms, again using the MD application. Measurements in this section are taken on a 64-node Kendall Square Research KSR-1 machine. The specific MD simulation used in these measurements uses a cylindrical domain decomposition; MD performance and speedups with different decompositions are evaluated in detail elsewhere[9].

Table 4 depicts the results of four different sets of MD runs, normed against a run of MD without monitoring. These experiments compare the performance and perturbation when using Falcon for five different cases: (1) when no monitoring performed (Original MD), (2) when tracing only MD calls to the underlying Cthreads package (Dft Mon Only), (3) when tracing Cthreads events as well as sampling (using sampling sensors) the 10 most frequently called procedures in MD (Dft Mon & Sampling), (4) when using the Unix GProf profiler existing on the KSR-1 machine (MD with Gprof), and (5) when tracing Cthreads events as well as the 10 most frequently called procedures in MD (Tracing All Mon Events). The table and figures list computation times and speedups with different numbers of processors. These measurements do not consider

single local monitoring thread used in the experiment.

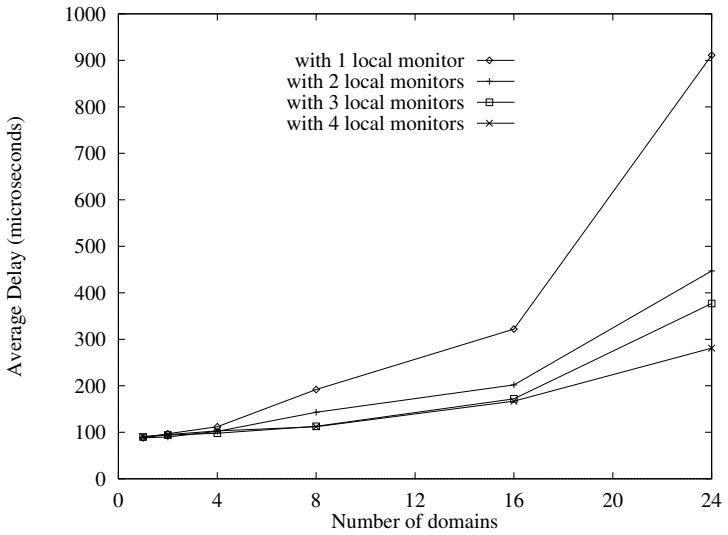


Figure 9: Monitoring latency with multiple local monitors on the KSR-2. (Each domain of particles is assigned to one processor.)

Number of Processors	Execution Time of Each Iteration (seconds) & Monitoring Overhead				
	Original MD	Dft Mon Only	Dft Mon & Sampling	Tracing All Mon Events	MD with Gprof
1	8.19	8.19(< 1%)	9.61(17%)	114.60(1299%)	22.53(175%)
4	2.65	2.65(< 1%)	3.21(21%)	59.30(2140%)	7.29(175%)
9	1.45	1.45(< 0%)	1.72(19%)	65.33(4406%)	4.28(195%)
16	0.62	0.63(1%)	0.73(17%)	54.29(8628%)	1.71(175%)
25	0.30	0.31(2%)	0.35(16%)	41.56(13776%)	0.82(173%)
36	0.19	0.20(4%)	0.23(16%)	33.65(17245%)	0.54(195%)

Table 4: Average execution time and perturbation of each iteration of MD with different amounts of monitoring or profiling on KSR-1.

the costs of either forwarding trace events to a some front end workstation or storing them in a trace data base, since those costs are not dependent on Falcon's design decisions but rather on the performance of the networking code and/or file system implementation of the KSR-1 machine. Specifically, measurements with trace events essentially 'throw away' events at the level of local monitors, whereas the measurements with sampling sensors actually use local monitors to retrieve and evaluate sampling sensor values stored in shared memory on the KSR-1 machine.

The MD application's performance with different amounts of monitoring or profiling is depicted in Figure 10, and the resulting program perturbation due to monitoring is shown in terms of speedup degradation in Figure 11. Evaluations of each experiment are presented next. The first experiment (Dft Mon Only – default monitoring) measures the overhead of monitoring when Falcon traces all calls to the underlying Cthreads package. Specifically, this is the amount of monitoring required for the thread life-time view

[4]Super-linear speedups are due to the KSR-1's ALLCACHE memory architecture. When MD runs on a large number of processors, it can load all of its code and data into the fast sub-caches or local caches associated with these processors, while

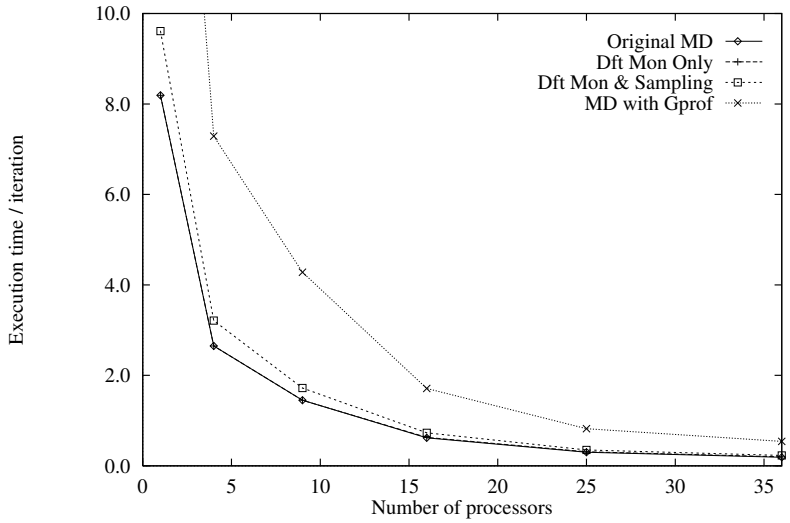


Figure 10: Comparing average execution time of each iteration of MD on the KSR-1.

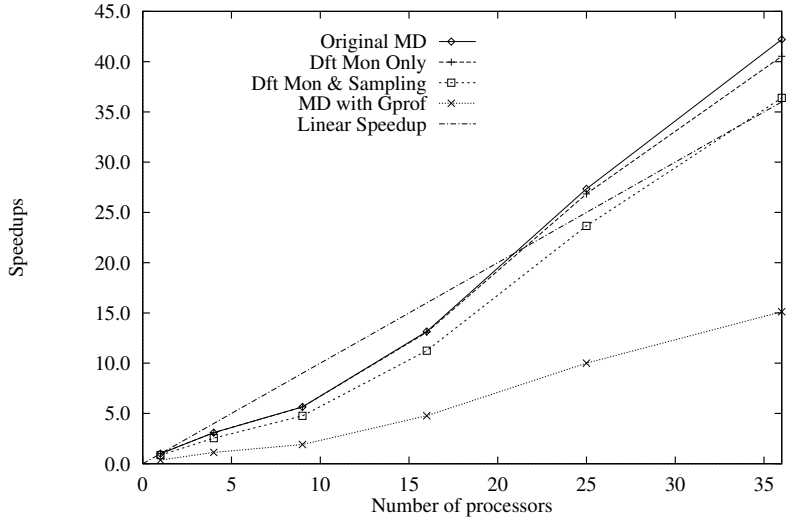


Figure 11: Comparing speedups of MD on the KSR-1[4].

shown in Figure 13 in Section 5.2. The monitoring information being collected includes the runtime activities associated with each thread (such as thread_fork, thread_join and thread_detach events), synchronization calls, and all other information displayed in the thread life-time view. It is apparent from Figures 10 and 11 that default monitoring does not noticably perturb the execution of MD. However, monitoring overheads increase slightly with an increasing number of processors, which is caused by an increasing number of events (more user threads imply more cthreads calls, and hence more monitoring events) generated during a shorter execution time and beginning to saturate the available local monitoring threads. The creation of additional local monitors can remedy this problem.

The second experiment compares the overhead of Falcon monitoring with that of commonly used program profiling tools, namely, with Gprof. The KSR implementation of Gprof used in these measurements has been optimized to take advantage of the machine's memory architecture in several ways, including replicating counters on each processor to avoid remote accesses. To compare fairly, we exclude the time spent on writing the results to file from the presented Gprof execution times. Using Falcon, we monitor the 10 most frequently called procedures in MD. These calls constitute about 90% of all procedure calls made in the program. Each procedure is monitored by a sampling sensor, which increments a counter for each procedure call being monitored. Counter values are sampled each millisecond by local monitoring threads. The result of this experiment is the addition of 20% to MD's total execution time. In comparison, with Gprof, the execution time of MD is increased by approximately 180%. Similar advantages of Falcon to other profiling tools are demonstrated when using Prof. Experimental results not reported in detail here show that Prof's overhead is approximately 130% [15]. The results described above are not surprising, since profiling tools typically maintain large amounts of compiler-derived information about a parallel program's attributes. In comparison, Falcon only maintains the specific information required for taking certain program measurements.

While the first two experiments clearly demonstrate the importance of monitoring only the program attributes of interest to the user, the third experiment shows that it is also important to adjust or select the techniques being used for information capture. In this experiment, tracing sensors are used in place of sampling sensors for monitoring the 10 most frequently called procedures in MD, which results in a very significant increment of monitoring overheads. The excessive performance penalties arising from this 'misuse' of tracing sensors are primarily due to the direct perturbation caused by monitoring tens of millions of procedures calls and are exacerbated by the saturation of the single local monitoring thread being used in the experiment. The resulting (lack of) performance clearly demonstrates two points. First, since tracing sensors are too expensive for procedure profiling, any monitoring system must offer a variety of mechanisms for information capture and analysis, including both sampling and tracing sensors. Second, since tracing can help users gain an in-depth understanding of code functionality and performance (see Sections 4.4 and 5), users should be able to both control the rates at which tracing is performed and the specific attributes of the application that are captured via tracing. We call the user's ability to focus monitoring on specific system attributes *selective monitoring*. It is explained in more detail in the next section.

In general, the experiments with MD presented in this section demonstrate that the multiple monitoring mechanisms (e.g., tracing vs. sampling sensors) supported by Falcon can be employed such that monitoring overheads remain moderate for realistic parallel application programs.

4.4 An Example of Selective Monitoring Using Falcon

In this experiment, the MD code's most computationally intensive component is monitored using Falcon's sampling and tracing sensors. Both types of sensors are needed since programmers require both summary (e.g., total number of invocations) and sequencing or dependency information (e.g., 'b' was done after 'a' occurred) to understand and evaluate code performance. Such dynamically selective monitoring is useful since programmers can focus on different phenomena at different times during the performance evaluation process. The specific purpose of the selective monitoring demonstrated in this section is to understand the

it cannot do so when running on a single processor.

effectiveness of certain, commonly used 'short cuts' which are intended to eliminate or reduce unnecessary computations in codes like MD.

The dominant computation of each domain thread in the MD code is the calculation of the pair forces between particles, subject to distance constraints expressed with a cut-off radius. This calculation is implemented with a four-level, nested loop organized as follows (pseudocode is shown below):

```
for (each molecule mol_1 in my domain) do
    for (each molecule mol_2 in domains within cut_off_radius) do
        if (within_cutoff_radius(mol_1, mol_2)) then continue;
        for (each particle part_1 in molecule mol_1) do
            if (within_cutoff_radius(part_1, mol_2)) then continue;
            for (each particle part_2 in molecule mol_2) do
                if (within_cutoff_radius(part_1, part_2)) then continue;
                calculate_pair_forces(part_1, part_2);
            end for
        end for
    end for
end for
```

The inner three levels of this loop check the distances between molecules and particles to eliminate all particles outside the cut-off-radius. When the distance between two molecules is checked, three dimensional bounding boxes are used for each molecule. Each molecule's bounding box includes all of its particles. The minimum distance between two molecules is defined as the distance between their bounding boxes' closest points, whereas the minimum distance between a particle and a molecule is the distance from the particle to the molecule's bounding box' closest point.

The question to be answered with selective monitoring is whether the additional costs arising from the use of bounding boxes is justified by the saved costs in terms of the resulting reduction in the total number of pair force calculations. More specifically, does the reduction is total number of pair force calculations justify the additional computation time consumed by bounding box calculations? A simple selective monitoring mechanism is used to answer this question, by dynamically monitoring the performance of this four-level loop. Specifically, a sampling sensor is first used to monitor the hit ratios of the distance checks at all levels. When a hit ratio at some loop level falls below some threshold, say 10%, a tracing sensor monitoring this loop level is activated to obtain more detailed information. The intent is to correlate the low hit ratio with specific properties of domains or even of particular molecules. Specifically, for each 'hit' distance check at the 2nd level loop, we trace the distances between particles and molecules at the 3rd level loop. The motivation is to understand the relationships of distances between molecules' bounding boxes and with distances between specific particles of a molecule with the bounding boxes of other molecules. In other words, what is the effectiveness of the second level distance check?

The performance of such dynamically selective monitoring is presented in Table 5. In these measurements, we use a MD data set that contains 300 molecules with 16 particles each. This relatively small system is then monitored by insertion of sampling and tracing sensors at one, two, three, or all levels of the nested loop (the outermost level is numbered zero, while the innermost three). Tracing at all levels results in overheads that are somewhat unacceptable, especially when the same tracing is performed for larger systems. This is apparent from the increases in monitoring overheads experienced when tracing at all levels for increasing system sizes (e.g., 9 vs. 16 domains). On the other hand, when tracing only at lower levels (e.g., levels 1 or 2), overheads are less than 1% for smaller systems and no more than 5% for larger systems, and sampling overheads remain small for all system sizes.

These results indicate that selective monitoring is quite effective, even when applied to this highest frequency set of loops in the MD program's execution. Furthermore, the strategy of sampling execution and only initiating tracing when some problem (e.g., a low hit ratio) is experienced should result in composite monitoring overheads that approximate the sampling overheads experienced with Falcon for long system

No. of domains	Execution Time of each MD time step (seconds) & Monitoring Overhead					
	No Monitoring	Sampling Hit-Ratio	Tracing at Level 1	Tracing at Level 2	Tracing at Level 3	Tracing at All levels
4	1.28	1.28(< 1%)	1.28(< 1%)	1.34(5%)	1.38(8%)	1.46(14%)
9	0.703	0.706(< 1%)	0.708(< 1%)	0.734(4%)	0.742(5%)	0.794(13%)
16	0.301	0.301(< 1%)	0.304(1%)	0.316(5%)	0.323(7%)	0.356(18%)
25	0.147	0.147(< 1%)	0.149(1%)	0.155(5%)	0.158(7%)	0.188(28%)

Table 5: Performance of selective monitoring of the MD's main computation component on the KSR-2.

runs. In conclusion, the on-line 'steering' of Falcon's monitoring mechanisms themselves can be used to control runtime monitoring overheads.

4.5 Performance of On-line Steering

As outlined in Section 3.4, the steering component of Falcon operates in conjunction with its monitoring components, by receiving and processing selected monitoring events, then controlling the application's execution based on such runtime state information. This section presents low-level measurements that highlight the basic performance and operation of program steering when viewed as a low-level control system. Therefore, tor these measurements, networking is disabled and, hence, no remote operations are performed with the steering client. Three processors are used, one running an application thread, a second running a single local monitoring thread, and a third running the steering server. Algorithmic steering is used in order the evaluate the basic costs of observing some interesting program state via the (1) local monitor and (2) steering server, (3) making a simple decision based on that observation, and (4) enacting that decision by taking a steering action. These costs are evaluated in the first experiment, which measures the latency of actions (1)-(4) for a lightly loaded system:

Measurement	microseconds
Avg. Latency	610
Min. Latency	224
Max. Latency	4483

Table 6: Latency for closed-loop steering.

Table 6 describes the closed-loop latency for steering under the following conditions: a total of 100,000 sensor events are generated by the application program, they are received by local monitors, and they are then forwarded to the steering server without any additional filtering or processing. (4) The steering server performs a simple action in response to each event's receipt. This action consists of a write to a variable in the application program. (1)-(4) are performed for an application program that repeatedly performs the following tasks. First, it generates a monitoring event using a Falcon sensor. Second, the program waits on some pre-specified memory location that will be asynchronously updated by the steering server. Third, the steering server receives the event from the local monitor, reads the event type, accesses its database of steering events to determine the actions required for this event type, and then uses a probe to enact this action. The probe essentially changes the value of the memory location on which the program is waiting.

For these measurements, the database only contains a moderate number of different steering event types and their respective actions.

The results depicted in Table 6 demonstrate an average latency of 610 microseconds for algorithmic program steering using Falcon. This implies that program steering can be performed using Falcon at rates approximating the execution times of the set of inner loops in programs like MD. However, it is not possible to use Falcon's current mechanisms to perform steering of program abstractions accessed with high frequencies, like the adaptable locks described in [35]. Such high-rate and low-latency steering must be performed by local monitors themselves, possibly using custom implementations of sampling sensors. Two surprising results depicted in the table are (1) the high maximum latency for servicing a steering event (4,483 microseconds), which is due to mismatches in the scheduling of application, monitoring, and steering threads, and (2) the low minimum latency of 224 microseconds for steering, which is comprised mainly of the costs of event transmission from the application, to local monitor, to steering thread, respectively (recall that monitoring latency is approximately 70 microseconds).

The second experiment evaluates more complex steering actions, by forcing the steering thread to take multiple actions for each received steering event. Specifically, Table 7 depicts the latencies of steering when for each received steering event, the steering server takes some variable number (1, 10, 100) of actions involving both probes and actuators. The purpose of this experiment is to determine the incremental costs of steering.

Complex action	Number	Microseconds	Gain
Probe write	1	643	-
Probe write	10	2930	4.6
Probe write	100	15418	23.9
Actuator	1	627	-
Actuator	10	1207	1.9
Actuator	100	7870	12.6

Table 7: Average closed-loop latency with complex actions.

First, consider the costs of probe-based steering. For each probe, different memory locations have to be accessed. In this experiment, worst case costs are evaluated by forcing the steering server to access its database once for each received event. As the complexity of the action increases, the execution time required by the steering server to execute this action increases. For a complex action, the server must execute this action before accepting any other events from the monitor. As seen from Table 7, the basic probe write costs 643 microseconds. These costs increase by a factor of 4.6 for a complex action that requires 10 probe writes (2,930 microseconds), and they increase by a factor of 23.9 for very complex actions (to 15,418 microseconds for 100 probe writes). These measurements indicate that the steering server's construction is sensible in that it permits the basic costs of steering to be amortized over the costs of increasingly complex actions.

The second portion of Table 7 addresses actuator costs. These measurements are interesting in their demonstration of scalability for actuators versus probes in terms of the resulting costs arising for steering servers. Specifically, 100 actuator activations do not correspond to 100 executions of actuator code by the steering server. Instead, the server simply enables the actuator once (for 100 executions), and then relies on the user program to execute steering actions using the enabled actuator. As expected, actuator-based steering costs do not depend on the number of steering actions taken; they depend only on the number of times actuators are enabled or disabled! However, in order for the steering server to program the actuators, the server must write to a buffer shared by the server and the application. A lock must prevent simultaneous access to this region. Unfortunately, this lock is a point of contention between the server and the application

and, as such, the performance of the server is somewhat degraded.

From these measurements and in accordance with earlier results presented in this paper and in [2] addressing the time required for analyzing monitoring output, it should be apparent that the steering component of Falcon is sufficiently fast to (1) keep up with fairly high rates of monitoring and (2) steer programs at rates and with overheads enabling medium grain on-line program configuration[5] and application steering.

5 The On-line Presentation of Monitoring Information

The process of on-line user interaction with a target application includes (1) obtaining application-specific information through monitoring mechanisms, (2) displaying this information to the user, and (3) controlling program execution based on (2). Steps (1) and (3) have been discussed in the previous sections. This section presents Falcon's methods for presenting monitoring information to end users.

5.1 Falcon's On-line Display System: An Overview

Graphical displays have been shown useful in presenting data structures [39], algorithms [48], runtime program behaviors [29], and performance information[17, 41] to human users. However, most current work deals primarily with off-line graphical and animated presentations of program and performance information. Falcon's specific goals concerning the presentation of information to end users are to evaluate: (1) how on-line displays of program information can help users understand a target program's performance and runtime behavior, and (2) how users can use such an understanding to steer their parallel codes. The resulting necessary attributes of graphical displays used for program steering include:

- *Application-specific displays* – program information should be presented to end users in familiar terms, that is, by reference to abstractions in their programs rather than by reference to machine or operating system details with which they may not be familiar.
- *Behavior-preserving displays* – program monitoring and information display cause program perturbation and they exhibit a lag between the time of information generation and its display to end users. Programmers and end users must be made aware of both monitoring perturbation and information delay and should, potentially, be able to control them when performing program steering.

The Falcon information display system offers functionality addressing both attributes. First, the central monitor in Falcon is able to 'attach' any number of event streams from local monitors to its input ports, and 'route' events to any number of analysis packages and subsequent displays through its output ports. Such attachments to either input or output ports may be changed during program execution, if needed. As a result, event streams may be subjected to multiple analysis packages and then displayed by any method of display chosen by end users. Attachments are created and dissolved by commands to the central monitor, and alternative displays may be associated with event streams by use of class hierarchies within the display process. Figure 12 demonstrates the use of three alternative display methods for a sample event stream. The first method, built on the X window system and the Athena and Pablo [41] widget sets, uses the work_load events for the display of the application's load balance information. The second method applies statistical methods to analyze events from the application and then presents the resulting summary information to end users in a textual format. The third method, built on the Polka animation system and the Motif widget set, uses specific events from the on-line event stream to animate the program's behavior at the threads level (also see Figure 13).

In this paper, we focus on 2D graphical displays of program behavior or performance, which have been shown useful for on-line steering in previous sections, where graphically displayed load balance information is used to direct the execution of the MD code (e.g., see Figure 1). Our future work is combining event streams from the monitoring system with program output typically generated via file system calls, so that users can

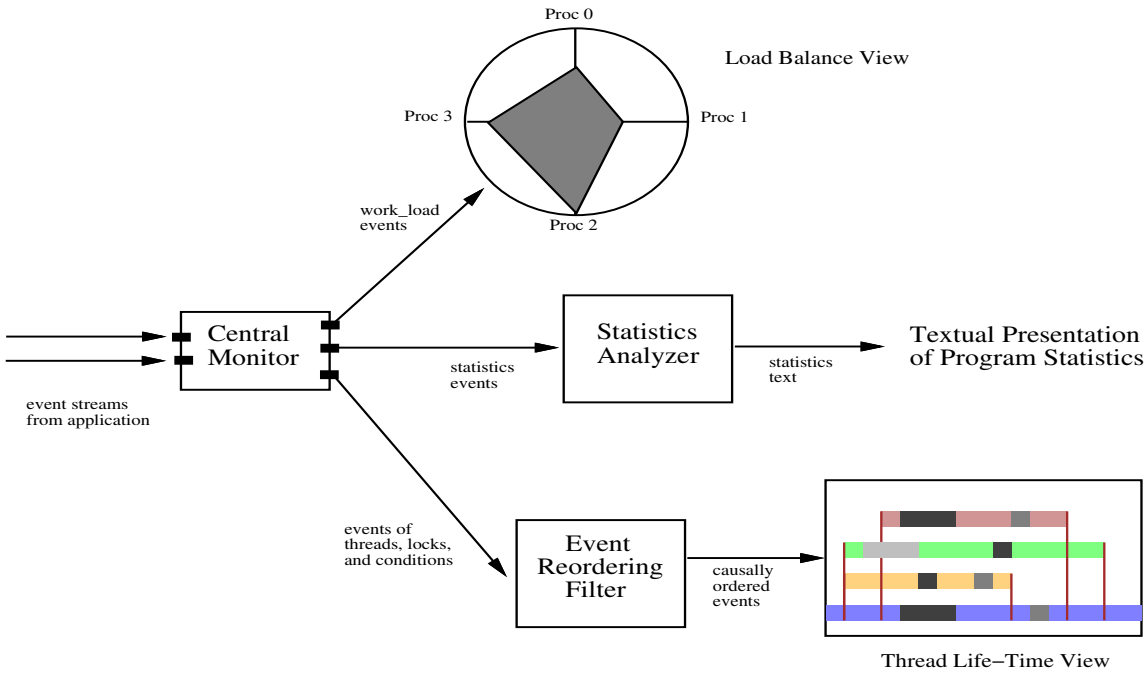


Figure 12: A sample on-line display system for an application.

understand and direct program execution in terms of individual program variables (e.g., 'energy levels' or 'molecular positions' in the MD code). Toward this end, we are now developing and integrating into Falcon interactive 3D data visualization tools. These tools are being applied to a large-scale atmospheric modeling application.

Falcon attempts to preserve the original behavior of the parallel program when displaying program performance. Two issues arise: (1) monitoring can perturb program execution, and (2) the monitoring system's method of event collection via buffers does not preserve the actual time ordering of events being produced and displayed. Specifically, since monitoring events are first buffered on the parallel machine and local monitoring threads are not perfectly synchronized, events received by the central monitor and ultimately, by analysis and display packages are not guaranteed to be in order. For off-line monitoring, event files can be sorted. For on-line monitoring, event reordering must be performed on-line and with suitable efficiency. Furthermore, in order to preserve the behavior of the original program when presenting such information to users, reordering must be performed so that the causal order of events exhibited by the executing program is preserved and enforced.

The remainder of this section describes how Falcon displays address both program perturbation and monitoring delay, by providing on-line perturbation information displayed as *perturbation events*, and by reordering and displaying monitoring events according to known information about a program's causal execution order. In both cases, the system-level default information about threads available in Falcon is utilized.

5.2 The Thread Life-Time View: Performance of Threaded Programs

The graphical *thread life-time* view described next is one contribution of the Falcon project toward understanding the dynamic behavior of threads-based parallel programs. Available with Cthreads programs running on SGI and SPARC workstations and on KSR machines, this view uses the default sensors embedded in Cthreads. The view is implemented with the Polka animation library [49]. Since Polka provides a variety of graphical objects, animation primitives, and user interface facilities, the program defining the thread

life-time view only consists of roughly 200 lines of application-level Polka code. Polka runtime libraries provide a flexible animation scheduling policy, permit different temporal mappings of program events to their animations, and therefore facilitates the construction of on-line displays. Polka is described and evaluated in detail in [49].

The thread life-time view shows the different states of threads over time. The state information depicted in the view includes thread execution time, blocking time, waiting time in ready queues, the identifiers of conditions or mutex locks on which threads are blocked, and thread identifiers. From this information, users can easily discern the time a thread spends doing useful computation versus waiting for other threads, the degree to which different threads' executions are synchronized, processor utilization, and other useful program and performance information. Figure 13 shows a snapshot of the MD program's execution on four processors (ie., molecules are partitioned into four domains) on the KSR-1. When a new thread is forked,

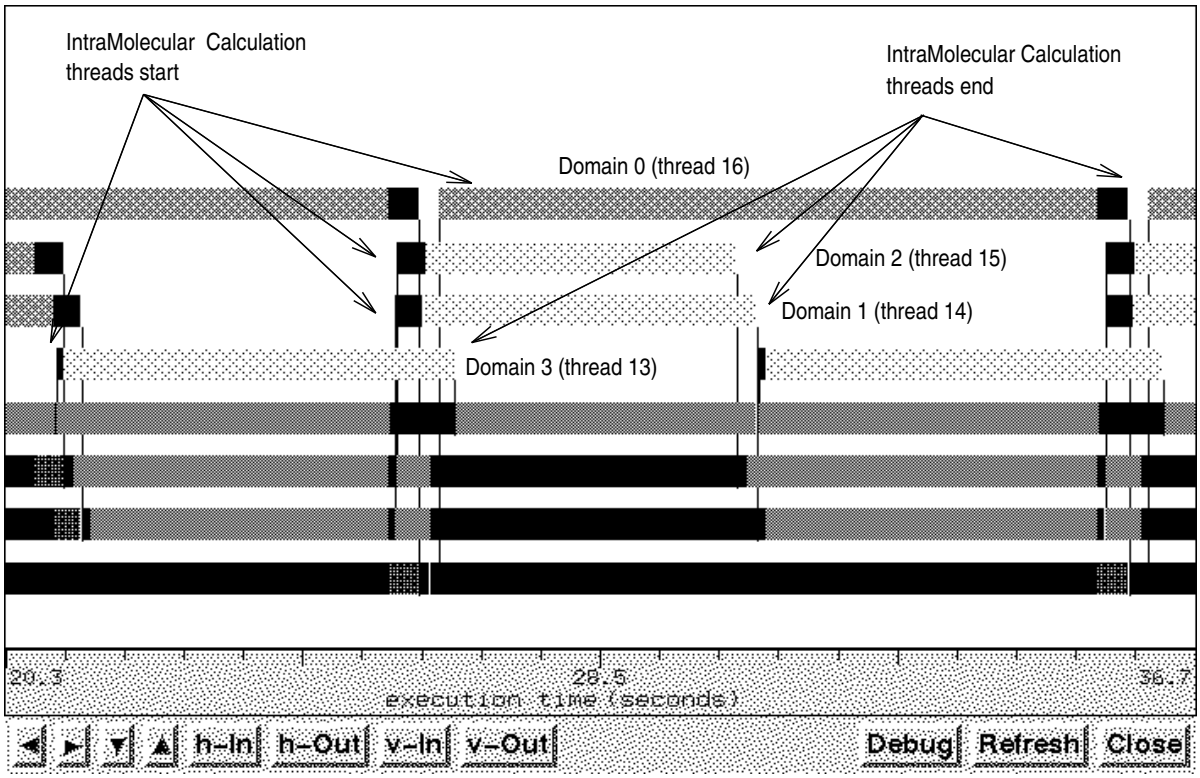


Figure 13: An annotated thread life-time view of MD. The actual display uses colors to represent different threads and different thread states.

a narrow horizontal bar is created to represent the new thread's life-time, and a vertical line is drawn from the parent thread to the child thread at the time of the fork event. A narrow bar terminates when the thread, which the narrow bar represents, joins another thread after it exits or when a detached thread calls thread_exit. In the case of thread_join, another vertical line is drawn from the caller thread to the thread it is joining. The resulting empty space in the display can be reused for depiction of a new thread, if there is any. Since the color display has to be rendered into monochrome for this presentation, some annotations are added to compensate for lost information. Specifically, the thread life-time view uses different colors and patterns to represent thread states. In Figure 13, the solid black pattern represents a thread in a running state, while the dark gray pattern represents a thread waiting for a condition. The lightly dotted pattern indicates that a thread has called thread_exit and is waiting to join to another thread. The heavily dotted pattern indicates that a thread is in a processor's ready queue; it is waiting for another thread using the processor to complete its execution.

The bottom four bars in Figure 13 represent four threads each computing properties of the molecules in their respective domains (numbered 0, 1, 2, and 3 from bottom to top). Each such 'domain' thread forks a second 'helper' thread in every iteration. These 'helper' threads are shown as short bars above the 'domain' threads (they may not be in the same order as the 'domain' threads). They calculate intramolecular forces while domain threads wait for information from neighboring domains. At the end of each iteration, domain threads perform neighbor-to-neighbor synchronization. As apparent from the figure, the intramolecular calculations of domain 3 proceed and perform useful computations while the domain thread is waiting for completion of the computations of neighboring domains' threads. However, domain thread 3 experiences significant wait time since its domain computation is finished quickly, whereupon it must wait for completion of neighboring threads' computation (it needs their data from the current iteration before starting the next iteration). Therefore, it is also clear from this view that work load is imbalanced: domain 3 has little work to do, while domain 0 is almost always busy. This illustrates a problem with the slab-based domain decomposition strategy used in this run of the MD program: Domain 0 is responsible for additional molecules in the substrate on which the liquid being modeled is layered; this substrate is much denser than the liquid and therefore, contains many more molecules.

5.3 Perturbation Events

The thread life-time view shown in Figure 13 can help users understand program performance problems only if the thread running and waiting time shown is due to application's execution and synchronization rather than monitoring perturbation caused by executing extra code and additional synchronization between application threads and monitoring threads. Section 4.1 shows that the basic perturbation due to sensor code execution is quite small, provided that monitoring buffers are sufficiently large and local monitoring threads can process events fast enough to keep monitoring buffers from being completely full. Furthermore, the direct perturbation due to the execution of a sensor is easily predicted from the sensor's type. As a result, such perturbation can be removed from the event trace by application of simple perturbation analysis. However, if local monitoring threads cannot keep up with the rate of event generation, then monitoring buffers will eventually become full, and application threads will have to wait for some time until events have been removed from such full buffers. Since such waiting time caused by filled monitoring buffers is not due to the program's code or data, the resulting thread life-time view can be incomprehensible or misleading to end users.

Figure 14 depicts a potentially misleading thread life-time view constructed with an event trace captured from a large-scale atmospheric modeling code. The problems in this view are due to an unsuitable configuration of the monitoring system (a single local monitoring thread), which is quickly overwhelmed by events from a large number of computational threads. Without perturbation events, it would appear to programmers that their computational threads execute for different amounts of time. This is misleading since in this program, each of the computational threads have the same amount of work. In fact, when first using Falcon, one of the atmospheric code's implementors spent several hours chasing a non-existing load imbalance indicated by the life-time view (without perturbation events). A more precise inspection of the view in Figure 14 shows pure black bars that represent 'worker' threads, each responsible for a partition of the computation space. The three bars below each 'worker' thread are 'helper' threads, which are employed by the 'worker' thread to help calculate separate terms in its computation. The iterative algorithm performs barrier synchronization after threads finish their work and before the next iteration starts. The figure indicates that the third 'helper' thread of the top 'worker' thread and the first 'helper' thread of the bottom 'worker' thread waited a very long time for mutex locks. In addition, the second and third 'helper' threads of the bottom 'worker' thread have significantly longer computation times than other 'helper' threads. These superficial observations imply that the program has unbalanced work loads and improper synchronizations. However, the improved threads life-time view with perturbation events presents a different picture.

Figure 15 shows the same execution of the atmospheric modeling code, with special perturbation events depicting the total blocking times experienced by its threads on the monitor's event buffers. It is clear from this 'correct' view that the 'helper' threads have balanced work loads, but that their execution times are extended due to monitoring perturbation experienced by the 'helper' threads of the bottom 'worker'.

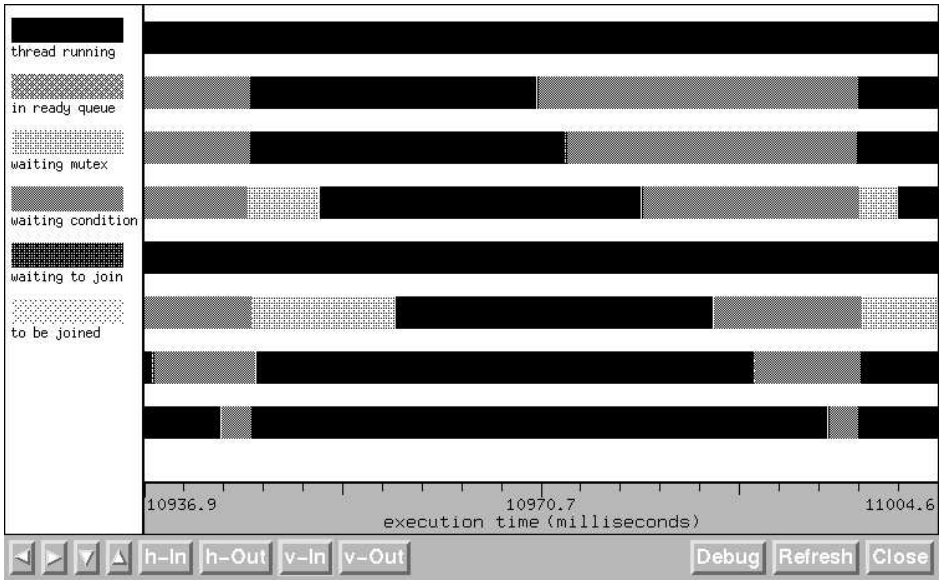


Figure 14: A thread life-time view that shows perturbation events.

Specifically, the extremely long blocking times for mutex locks apparently experienced by the third 'helper' thread for the top 'worker' and the first 'helper' thread for the bottom 'worker' (shown in Figure 14) are not due to mutex locking. They are due to additional thread waiting times experienced when writing mutex_end_lock events to the monitoring buffers. From this example, it should be evident that perturbation events help users understand the monitoring system's contribution to total thread execution and wait times. Our current work is generalizing this straightforward notion of perturbation events to apply more sophisticated sequential perturbation analyses (e.g., see [30]). Another type of monitoring perturbation, causing misordered event streams, is discussed next.

5.4 On-line Event Reordering

Event orderings and program animation. Displays like the thread life-time view of Figure 13 can provide users with insights into program progress and correctness. However, the perturbation example described above already demonstrates that graphical views can be quite misleading and confusing if the information being displayed does not correspond to the program's actual execution. This section focusses on another issue with on-line graphical views, namely, on the fact that the graphical animation order determined by the receipt and display of monitoring events does not correspond to the actual or causal order in which program events occur! Such misorderings can both confuse users and more critically, cause failures of the animation itself. For example, causal ordering would require that the thread_fork event creating a thread precede any event executed by the new thread. A display that shows a child running before it has been forked by its parent does not make any sense. Furthermore, suppose that the first event for this child thread is a condition_wait event. In the thread life-time view of Figure 13, this event is represented by a change in the color and fill pattern of that thread's horizontal bar. However, if the thread_fork event has not been received by the display system, the horizontal bar does not yet exist. When the display system attempts to perform a color-change action on this non-existent object, it crashes. Some of these crashes could be avoided by adding a layer of error-checking code to the display system, but this adds execution overhead, makes displays more difficult to design, and still leaves the viewer with displays that may not be useful.

The out-of-order events that cause problems for the display system *cannot* have occurred in the program's execution, since they would violate causal event orderings determined by program and language semantics.

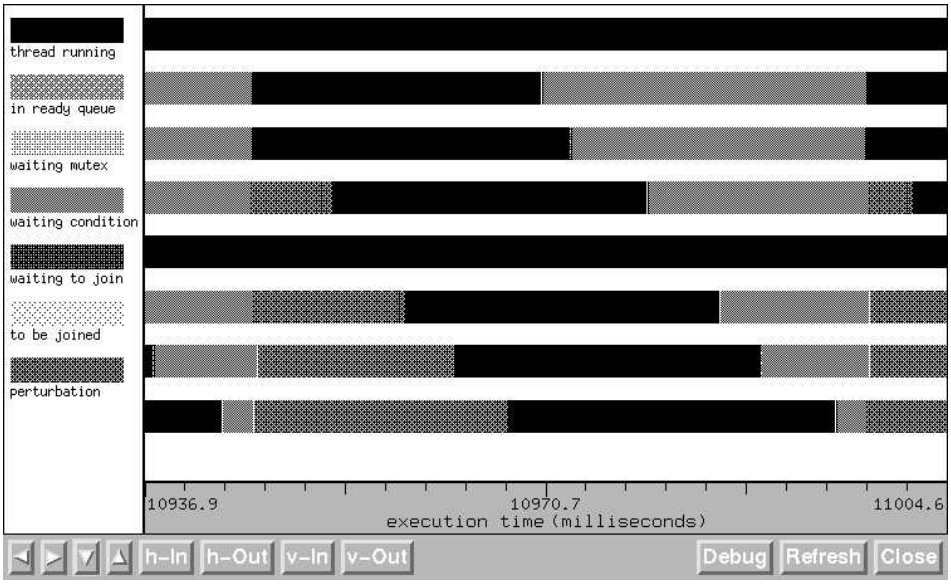


Figure 15: A thread life-time view that shows perturbation events.

Instead, misorderings existing in the event stream are due to the buffering and processing methods employed in the monitoring system. Specifically, high monitoring performance (i.e., low perturbation) requires that events for each thread be buffered until a local monitor is ready to process them. Furthermore, different local monitors send events to the central monitor at their own speeds, in part because the number of events to be processed and the processing requirements of individual events may differ among local monitors. As a result, while the event stream reaching the display system is in-order with respect to each individual thread (recall that each thread uses only a single event buffer), it may be out of order with respect to thread events from different threads.

On-line event reordering. The diagnosis and correction of out-of-order events is a common problem in parallel and distributed monitoring systems. Existing systems (e.g., ParaGraph[17] and SIEVE[42]) rely on a sort by timestamp value to impose a total order on all events stored in event files. The on-line nature of the Falcon monitoring system precludes using such a solution, and sorting by timestamp order does not entirely eliminate the problem of out-of-order events[4]. In addition, coarse clock granularities and poor clock synchronization among different processors may lead to event timestamps that do not accurately reflect the actual order of program execution. For example, if the system clock changes only every 10 milliseconds, and if two events occur within this time frame, then the ordering of these two events cannot be determined within this period. A more realistic concern on the KSR supercomputer used in our work is poor clock synchronization, where one processor's clock can be sufficiently ahead of another processor's clock so that the elapsed time between a thread fork and the first event executed by the child appears to be negative. This problem is exacerbated when threads are allowed to migrate across processors, something we avoid in Cthreads but is permitted in the Pthreads parallel programming library on the KSR machine.

Ordering rules. The previous discussion of out-of-order events makes apparent that the use of timestamps is not sufficient for determining and enforcing suitable, global event orderings. Falcon addresses this issue by employing an *ordering filter* between the central monitor and the display system (see Figure 12). This filter ensures that the event stream reaching the display system adheres to a pre-specified, known causal ordering among thread events. This ordering filter has knowledge of all execution threads, mutex locks, and conditions identified occurring in the event stream. The algorithm employed by the filter follows a "minimum-intervention policy". Namely, it examines each event in the stream arriving from the monitoring system, checks the applicable ordering rules for this event type, and if no rules are violated, forwards the

event to the display system. If a rule violation is indicated, the event is held back until the rules are satisfied.

As an example, consider the ordering rule for a mutex lock event. Actually, a mutex lock is recorded as two separate events - the mutex_begin_lock event indicating that a thread has attempted to obtain the lock, and the mutex_end_lock event indicating that a thread has succeeded in obtaining the lock. The following ordering rule is observed by the filter for a mutex_end_lock:

```
mutex_end_lock t m n <- ((thread_init t || thread_fork pt t) &&
                         (mutex_init m || mutex_alloc m) &&
                         (mutex_unlock m n-1) )
```

This rule may be translated as: "The mutex_end_lock event with parameters t, m, and n, may be passed on to the display system if thread t has been initialized or forked by a parent thread, mutex variable m has been initialized or allocated, and the mutex_unlock event for variable m, sequence number n - *1* has already been passed on to the display system." Accordingly, the parameters associated with the event mutex_end_lock are t, the id of the thread attempting to obtain the lock, m, the id of the mutex variable, and n, the sequence number indicating the number of successful lock attempts on this particular mutex variable. Among these parameters, the most interesting parameter is n, since it required an unforeseen augmentation of the Falcon system and since it enables the efficient implementation of on-line event ordering discussed below.

The rule applied to a mutex lock is one of many rules implemented by the reordering filter (see Appendix A for a complete listing of these rules). Moreover, even for this single rule, with each of its expressions is associated another set of ordering rules that must be met. The rules appearing in Appendix A are written to reflect the logic of the current filtering code. Our future work is addressing the automatic generation of filtering code from formal rule specifications like the one shown above.

Implementation of on-line reordering. Figure 16 outlines the implementation of the event reordering filter. The event stream at the left arriving from the central monitor is only partly ordered with respect to

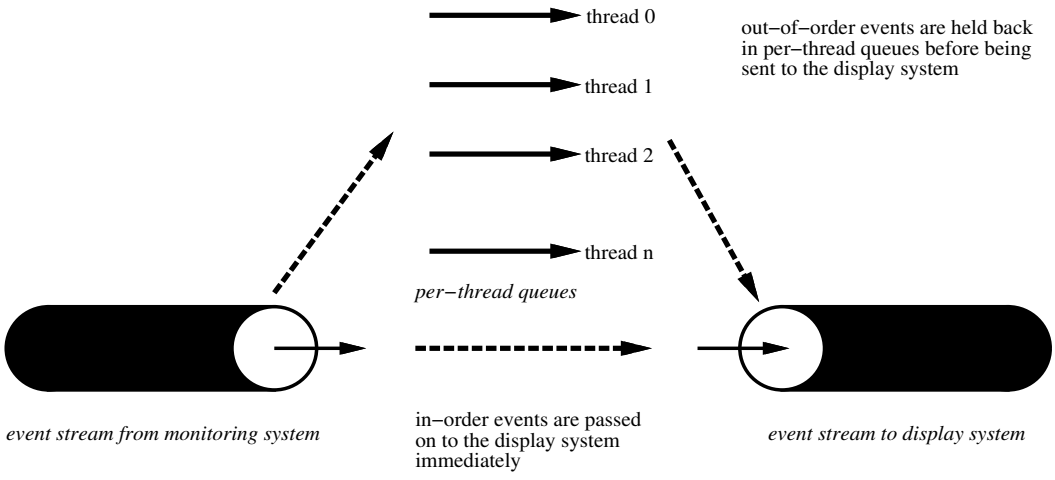


Figure 16: Architecture of the on-line trace reordering filter.

each thread_id. The event stream forwarded to the display system shown at the right hand side of the figure is ordered according to the specified ordering rules. To attain this ordering, the filter maintains an ordered queue for all events with the same thread_id encountered in the event stream, shown at the center of the figure. This queue only contains events that are not ready to be processed (that do not yet satisfy the rules), whereas other events are immediately forwarded to the display system. The ordering filter then continues to examine new events, checking the head of each active queue in every round to see if it is now possible to

place the event in the stream going to the display system. Note that these queues are not activated until a thread_init event (in the case of the program's initial thread), or a thread_fork event (all other threads) is processed for that thread_id. Processing of the queue is turned "off" again when a thread_exit is encountered. Straightforward generalizations of this code would entail dynamic queue creation and deletion at some cost in runtime performance.

Additional data structures in the ordering filter are assigned to mutex_ids and condition_ids, each of which is represented by a data structure that keeps track of the sequence numbers associated with this abstraction that have been processed thus far. An example of these data structures is shown in Figure 17, where threads are waiting on both condition variables. These data structures are dynamically allocated as the events are

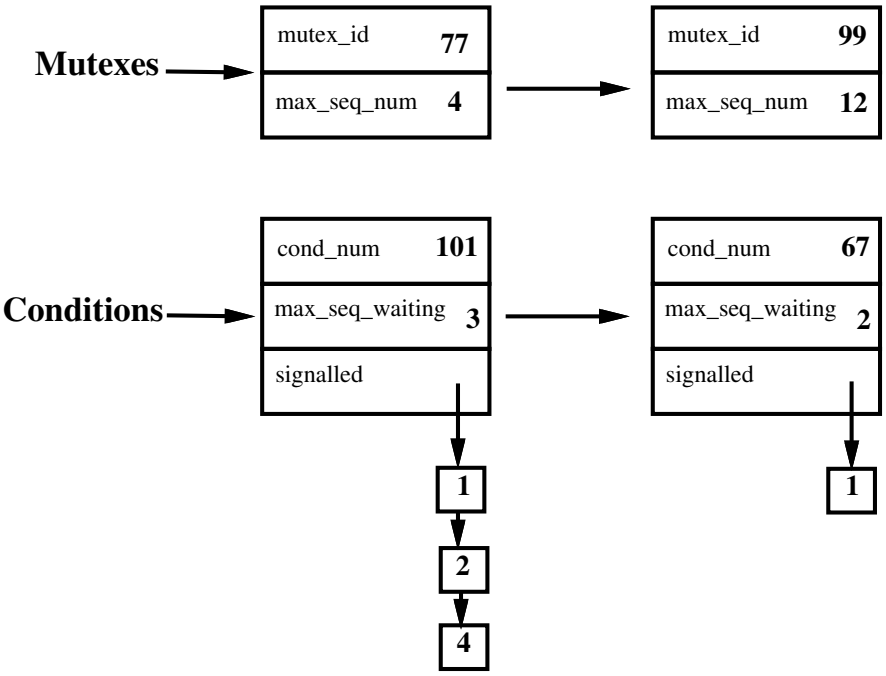


Figure 17: Detail.

observed in the stream. A mutex_init or mutex_alloc event causes data structure allocation for this mutex_id, and the sequence number for the mutex is initialized to 0. No event associated with this mutex_id may be processed until after the mutex_init or mutex_alloc events have occurred. In addition, mutex_end_lock and mutex_unlock events have a sequence number, and are required to be processed in sequence number order. Similar data structures exist for events concerning condition variables, again requiring that condition_init or condition_alloc events precede each condition's use and using sequence numbers initialized to 0. Specifically, a condition_end_wait event for sequence number n must be preceded by a condition_signal on sequence number n or a condition_broadcast on a range of sequence numbers containing n. In turn, a condition_signal on n must be preceded by a condition_begin_wait on n. A condition_broadcast on *n1 ... n2* must be preceded by a condition_begin_wait on *n2*. The condition_begin_wait on sequence number n must be preceded by condition_begin_wait on sequence number *n-1*.

Evaluation. Meaningful performance numbers for the efficiency of the ordering filter are difficult to obtain. Because online monitoring *requires* the ordering filter to prevent display crashes, it is not possible to compare the appearance and execution of the display with the ordering filter versus without the ordering filter. Instead, we have attempted to evaluate the effects of the ordering filter on the appearance and speed of the displays under three offline conditions. The degree to which the events are misordered may also have an effect on the delay or "drag" that the reordering filter may impose on the display. Accordingly, we have produced trace

files with varying degrees of misordering and have developed a metric to describe the degree of misordering in an event stream or file. Traces were collected from four executions of the MD application. For each run the buffer size of the local monitor was varied in order to produce trace files with varying ratios of out-of-order events. The use of large buffers should produce more out-of-order events in the trace file (but less perturbation in the program), and smaller buffers should cause fewer out-of-order events (but more perturbation in the program).

As a measure of the misordering of the events, we calculated a *hold-back ratio*. Recall that the reordering code will temporarily hold back any event that violates causal ordering. If a misordered event is held back for multiple times, it will be counted for as many times. In our experiments, the hold-back ratios for the four trace files range from 0.60 (9,020 events held back in a file of 14,970 records) for the smallest size local buffer to 2.81 (40,903 events held back in a file of 14,552 records) for the larger local monitor buffer. The results clearly confirms the hypothesis that smaller event buffers cause more out-of-order events and larger buffers causes less out-of-order events.

For each trace file, we run a sorting program to produce another version of the trace file with all event records totally ordered by their timestamps. The thread life-time display code is then executed, observed, and timed for each of four trace files under the following three conditions: (1) the thread life-time view reading directly from the sorted file, (2) the reordering filter reading from the *sorted* trace file, passing event records to the thread life-time display through a socket, (3) and the reordering filter reading from the *original* trace file, passing event records to the thread life-time display through a socket.

Not surprisingly, the total running time of the display under the second condition exceeds the first in every case, ranging from a 3% increase to a 6% increase in display time. We attribute this delay primarily to CPU contention between the display and reordering code (they run on the same machine). However, the running times of the thread life-time display under the second and the third conditions are not significantly different. The degree of misordering does not significantly affect display execution time simply because the reordering code is much faster, from 10 to 30 times, than the display code itself, which relies on relatively more expensive X-windows call to show events to end users. In other words, the reordering filter is sufficiently fast to supply the display code with a steady stream of events.

6 Related Research

Interactive program steering. The concept of steering can be found in many interactive scientific visualization and animation applications which allow users to directly manipulate the objects to be visualized or animated [22, 21]. For example, in a wind tunnel simulation, users can interactively change shapes and boundaries of objects in the wind tunnel in order to see the effects on the air flow. Research has also addressed the provision of programming models and environments to support the interactive steering of scientific visualization. In [22], DYNA3D and AVS (Application Visualization System from AVS Inc.) are combined with customized interactive steering code to produce a time-accurate, unsteady finite-element simulation. The VASE system [21] offers tools that create and manage collections of steerable Fortran codes.

The idea of steering has also been used in parallel and distributed programming to dynamically change program states or execution environment for improving program performance or reliability [5, 35, 8]. Early work in this research area focusses on the dynamic tuning of parallel applications in order to adapt them to different execution environments [44, 45]. Recent experiments demonstrate that changes to specific program states or program components, such as locks [35] and problem partition boundaries [8], can significantly improve overall performance. Our research interests are to provide a mechanism for programmers easily take advantage of this dynamic tuning capability as well as supporting the on-line capture of program and performance information necessary for efficient program steering. While we can base some of our work on past research on the monitoring of parallel and distributed programs for correctness and/or performance debugging, on-line and dynamic monitoring are relatively new topics[40]. We refer the reader to [16] for a brief survey of current research on interactive steering and on-line monitoring.

Program monitoring. Past work in monitoring of parallel and distributed programs focuses on performance understanding and debugging. These performance monitoring systems (e.g. Miller's IPS[34] and IPS-2[33], Reed's Pablo[41]) provides programmers with execution information about their parallel codes, and leads their attention to those program components on which most execution time is spent. A variety of performance metrics, such as normalized processor time[1], execution time on the critical execution path [33], etc., are employed to describe the program's runtime performance. One limitation of these performance metrics is the difficulty to relate measured performance numbers to specific program details. Instead, most such research measures program execution times at the procedure level. However, program steering can depend on program information derived from specific program variables or statements, such as the analyses of the workloads of each domain when steering the MD application.

Some recent work has addressed application-specific program monitoring[47, 40]. In these systems, users can explicitly specify what variables or program states to monitor using specification languages [40, 23], some of which are based on the Entity-Relational model[47]. The W^3 search model described in [20] addresses this problem in a different fashion: performance data is collected using hooks either inserted by the compiler or by programmers; based on this data, potential performance bottlenecks are identified and resources causing these bottlenecks are found and then, corrected by application programmers.

Data and perturbation analysis. Monitoring information may be refined with trace data analysis techniques, such as the Critical Path Analysis and Phase Behavior Analysis described in [33], often in an off-line manner. More sophisticated analysis techniques may be used to reduce and correct perturbation to the measured program performance due to monitoring [30]. In addition, performance data may be subjected to various statistical filtering techniques prior to its display to users. All such techniques may be applied to Falcon's monitoring data, as well.

A number of systems have addressed the problem of "out-of-order" events, events that violate causality. These events violate the "happened-before" relationship described in [27] and [10]. Post-mortem display systems such as ParaGraph[17] and SIEVE[42] may sort the trace files by timestamp. Instant Replay[28], Makbilan[52], TraceViewer[19], the Animation Choreographer[25], and Xab[3] have all used a causality graph as on ordering tool for the post-mortem display of the execution of parallel programs. These methods are not effective for run-time performance display because they rely on fully available trace files that may be sorted prior to their display. In contrast, Xab[3], a tool for monitoring PVM programs, uses a timestamp adjustment approach. Each processor calculates time as the sum of its local clock and an "offset" value. This offset value is adjusted whenever a process a message with a a later timestamp than the receiving process's current time. However, it was found that lower-level changes to PVM were required to eliminate some "out-of-order" events. These changes are in part analogous to the Cthread-based support provided for on-line event reordering in the Falcon system.

On-line program steering utilizes current and past efforts concerning the efficient linkage of multiple supercomputer engines, which is being addressed by several Gigabit testbeds efforts in the United States. Systems like PVM[50] and Express offer software support for constructing large-scale distributed and parallel codes.

7 Conclusions and Future Work

The Falcon monitoring system enables programmers to capture and view precisely the program attributes of interest to them. Such monitoring may be performed on-line (during the program's execution) with low latency and more importantly, with dynamically controlled monitoring overheads. To attain such controls, Falcon's monitoring mechanisms themselves may be configured on-line to realize suitable tradeoffs in monitoring latency, overhead, and perturbation.

Falcon performs program monitoring on-line, namely, monitoring information is captured, analyzed, and stored or displayed during the target program's execution. This permits programmers to view their long-running parallel codes interactively, and then steer their execution into more appropriate data domains or

simply, to play 'what if' games with alternative parameter settings. Toward this end, Falcon also offers an integrated library for interactive program steering, as well as support for the on-line provision of monitoring information both to algorithms controlling program configuration and to graphical displays based on which users can perform program steering.

This paper demonstrates the utility and potentials of on-line program steering and monitoring with a large-scale parallel application program, a molecular dynamics simulation used by physicists to study the interfacial properties of lubricants. Additional measurements are based on an atmospheric modeling code used by scientists to study global atmospheric phenomena. When Falcon is used with these programs, it becomes apparent that programmers should be permitted to perform monitoring and steering at multiple levels of abstraction within a single parallel program, ranging from inspecting and steering individual program variables to steering at the threads or process level. The evaluation of Falcon's performance with these applications also demonstrates the importance of supporting multiple degrees of granularity (and accompanying overheads) with which monitoring may be performed. Detailed performance studies on a 64-node KSR shared memory multiprocessor show how changes in the methods of capturing program information can result in distinct differences in monitoring performance. In other publications, we also also demonstrate some limitations on applying Falcon's functionality, notably when using it for the steering of individual operating system abstractions used by parallel programs (e.g., mutex locks[35]). To support the monitoring and steering rates required for such fine grain program control, monitoring mechanisms must be customized. Our future work will address how such customized mechanisms may be used in conjunction with the remainder of the Falcon system. In addition, future work is addressing the monitoring of object-oriented, parallel programs, including the provision of default monitoring views and performance displays[38].

The MD and atmospheric modeling codes as well as the Falcon system are implemented and evaluated on a 64-node KSR shared memory supercomputer. However, the Falcon system is available on several shared memory platforms, including SGI and SUN Sparc parallel workstations. A version of Falcon currently being completed also works with PVM across networked execution platforms. Similar portability is attained for the graphical displays used with Falcon. Notably, the Polka animation library can be executed on any Unix platform on which Motif is available [49]. The Falcon system has been in routine use at the Georgia Institute of Technology by non-Computer Science end users. Its low-level mechanisms are available via the Internet since early Summer 1994. A version of Falcon offering on-line user interfaces for monitoring and monitor control will be released in 1995.

Current extensions of Falcon not only address additional platforms (e.g., an IBM SP machine now available at Georgia Tech and the monitoring of PVM programs running Cthreads, C, or Fortran programs), but also concern several essential additions to its functionality. First, currently, users can insert into their code simple tracing or sampling sensors, where sensor outputs are forwarded to and then analyzed by the local and central monitors. We are now generalizing the notion of sensors to permit programmers to specify higher level 'views' of monitoring data like those described in [24, 40, 47]. Such views will be implemented with library support resident in both local and central monitors. Second, we are developing notions of composite and extended sensors that can perform moderate amounts of data filtering and combining before tracing or sampling information is actually forwarded to local and central monitors. Such filtering is particularly important in networked environments, where strong constraints exist on the available bandwidths and latencies connecting application programs to local and central monitors.

An important component of our future research is the use of Falcon with very large-scale parallel programs, either using thousands of execution threads or exhibiting high rates of monitoring traffic. For these applications, it will be imperative that monitoring mechanisms are dynamically controllable and configurable. Namely, it must be possible for users to focus their monitoring on specific program components, to alter such monitoring dynamically, and to process monitoring data with dynamically enabled filtering or analysis algorithms. Moreover, such changes must be performed so that monitoring overheads are experienced primarily by the program components being inspected. Dynamic control of monitoring is also important for the efficient on-line steering of parallel programs of moderate size. Specifically. program steering requires that monitoring overheads are controlled continuously, so that end users or algorithms can perform steering actions in a timely fashion.

On-line control of monitoring performance will be performed in Falcon by affecting the rates of data collection by individual or sets of sensors, the degrees of parallelism used by local monitors, and the amounts of filtering done by local monitors prior to information transfers to central monitors. In addition, we are developing on-line control algorithms that permit Falcon's use with real-time applications.

Longer term research with Falcon addresses the integration of higher level support for program steering, including graphical steering interfaces, and the embedding of Falcon's functionality into a programming environment supporting the process of developing, tuning, and steering threads-based parallel programs, called LOOM. In addition, Falcon will be a basis for the development of distributed laboratories in which scientists can inspect, control, and interact on-line with virtual or physical instruments (typically represented by programs) spread across physically distributed machines. The specific example being constructed by our group is a laboratory for atmospheric modeling research, where multiple models use input data received from satellites, share and correlate their outputs, and generate inputs to on-line visualizations. Moreover, model outputs (e.g., data visualizations), on-line performance information, and model execution control may be performed by multiple scientists collaborating across physically distributed machines.

Acknowledgements. We thank Niru Mallavarupu for contributing to early implementations of Falcon components. Thomas Kindler is responsible for the parallel implementation of the atmospheric modeling code.

References

[1] Thomas E. Anderson and Edward D. Lazowska. Quartz: A tool for tuning parallel program performance. In *Proc. of the 1990 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 115–125, Boston, May 1990.

[2] Peter Bates. Debugging heterogeneous distributed systems using event-based models of behavior. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 11–22, Madison, Wisconsin, May 1988.

[3] Adam Beguelin, Jack Dongarra, Al Geist, and Vaidy Sunderam. Visualization and debugging in a heterogeneous environment. *Computer*, 26(6):88–95, June 1993.

[4] Adam Beguelin and Erik Seligman. Causality-preserving timestamps in distributed programs. Technical Report CMU-CS-93-167, Carnegie Mellon University, Pittsburgh, PA, June 1993.

[5] Thomas E. Bihari and Karsten Schwan. Dynamic adaptation of real-time software. *ACM Transactions on Computer Systems*, 9(2):143–174, May 1991.

[6] Gretchen P. Brown, Richard T. Carling, Christopher F. Herot, David A. Kramlich, and Paul Souza. Program visualization: Graphical support for software development. *IEEE Computer*, 18(8):27–35, August 1985.

[7] Bernd Bruegge. A portable platform for distributed event environments. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 184–193, Santa Cruz, California, May 20-21 1991. ACM Press. ACM SIGPLAN NOTICES 26(12), December 1991.

[8] Greg Eisenhauer, Weiming Gu, Karsten Schwan, and Niru Mallavarupu. Falcon – toward interactive parallel programs: The on-line steering of a molecular dynamics application. In *Proceedings of The Third International Symposium on High-Performance Distributed Computing (HPDC-3)*, pages 26–34, San Francisco, CA, August 1994. IEEE, IEEE Computer Society.

[9] Greg Eisenhauer and Karsten Schwan. Md - a flexible framework for high-speed parallel molecular dynamics. In Adrian Tentner, editor, *High Performance Computing - 1994*, pages 70–75, P.O. Box 17900, San Diego, CA 92177, April 1994. Society for Computer Simulation, Society for Computer Simulation. Proceedings of the 1994 SCS Simulation Multiconference.

[10] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, August 1991.

[11] Ahmed Gheith, Bodhi Mukherjee, Dilma Silva, and Karsten Schwan. Ktk: Kernel support for configurable objects and invocations. In *Proceedings of the Second International Workshop on Configurable Distributed Systems*, pages 92–103, Pittsburgh, Pennsylvania, March 1994. The IEEE Computer Society Press.

[12] Ahmed Gheith and Karsten Schwan. Chaos-arc – kernel support for multi-weight objects, invocations, and atomicity in real-time applications. *ACM Transactions on Computer Systems*, 11(1):33–72, April 1993.

[13] Kaushik Ghosh, Kiran Panesar, Richard M. Fujimoto, and Karsten Schwan. PORTS: A parallel, optimistic, real-time simulator. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, Edinburgh, July 1994. College of Computing, Georgia Institute of Technology. to appear.

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

[15] Weiming Gu, Greg Eisenhauer, Eileen Kraemer, Karsten Schwan, John Stasko, Jeffrey Vetter, and Nirupama Mallavarupu. Falcon: On-line monitoring and steering of large-scale parallel programs. In *Proceedings of FRONTIERS'95*, February 1995. To appear. Also available as Technical Report GIT-CC-94-21, College of Computing, Georgia Institute of Technology.

[16] Weiming Gu, Jeffrey Vetter, and Karsten Schwan. An annotated bibliography of interactive program steering. *ACM SIGPLAN Notices*, 29(9):140–148, September 1994.

[17] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, September 1991.

[18] David P. Helmbold, Charles E. McDowell, and Jian-Zhong Wang. Determining possible event orders by analyzing sequential traces. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):827–840, July 1993.

[19] David P. Helmbold, Charlie E. McDowell, and Jian-ZhongWang. Traceviewer: A graphical browser for trace analysis. Technical Report UCSC-CRL-90-59, Univ. of California at Santa Cruz, Santa Cruz, CA, October 1990.

[20] Jeffrey K. Hollingsworth and Barton P. Miller. Dynamic control of performance monitoring on large scale parallel systems. In *Proceedings of the 7th ACM International Conference on Supercomputing*, pages 185–194, Tokyo, Japan, July 1993.

[21] David Jablonowski, John Bruner, Brian Bliss, and Robert Haber. VASE: The visualization and application steering environment. In *Proceedings of Supercomputing'93*, pages 560–569, November 1993.

[22] David Kerlick and Elisabeth Kirby. Towards interactive steering, visualization and animation of unsteady finite element simulations. In *Proceedings of Visualization'93*, 1993.

[23] Carol Kilpatrick, Karsten Schwan, and David Ogle. Using languages for describing capture, analysis, and display of performance information for parallel and distributed applications. In *International Conference on Computer Languages '90, New Orleans*, pages 180–189. IEEE, March 1990.

[24] Carol E. Kilpatrick and Karsten Schwan. ChaosMON – application-specific monitoring and display of performance information for parallel and distributed systems. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 57–67, Santa Cruz, California, May 20-21 1991. ACM Press. ACM SIGPLAN NOTICES 26(12), December 1991.

[25] Eileen Kraemer and John T. Stasko. Toward flexible control of the temporal mapping from concurrent program events to animations. In *Proceedings Eighth International Parallel Processing Symposium*, pages 902–908, 1994.

[26] Jeff Kramer and Jeff Magee. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, SE-11(4):424–436, April 1985.

[27] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communication of the Association for Computing Machinery*, 21(7):558–565, July 1978.

[28] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–481, April 1987.

[29] Allen D. Malony, David H. Hammerslag, and David J. Jablonowski. Traceview: A trace visualization. *IEEE Software*, pages 19–28, September 1991.

[30] Allen D. Malony, Daniel A. Reed, and Harry A. G. Wijshoff. Performance measurement intrusion and perturbation analysis. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):433–450, July 1992.

[31] Keith Marzullo and Mark Wood. Making real-time reactive systems reliable. *ACM Operating Systems Review*, 25(1):45–48, January 1991.

[32] Henry Massalin and Calton Pu. Threads and input/output in the synthesis kernel. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 191–201. SIGOPS, Assoc. Comput. Mach., December 1989.

[33] Barton P. Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski. IPS-2: The second generation of a parallel program measurement system. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):206–217, April 1990.

[34] Barton P. Miller and Cui-Qing Yang. IPS: An interactive and automatic performance measurement tool for parallel and distributed programs. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 482–489, Berlin, West Germany, September 1987. IEEE.

[35] Bodhi Mukherjee and Karsten Schwan. Experiments with a configurable lock for multiprocessors. In *Proceedings of the International Conference on Parallel Processing, Michigan*, pages 205–208. IEEE, Aug. 1993.

[36] Bodhisattwa Mukherjee. A portable and reconfigurable threads package. In *Proceedings of Sun User Group Technical Conference*, pages 101–112, June 1991.

[37] Bodhisattwa Mukherjee and Karsten Schwan. Improving performance by use of adaptive objects: Experimentation with a configurable multiprocessor thread package. In *Proc. of Second International Symposium on High Performance Distributed Computing (HPDC-2)*, pages 59–66, July 1993. Also TR# GIT-CC-93/17.

[38] Bodhisattwa Mukherjee, Dilma Silva, Karsten Schwan, and Ahmed Gheith. Ktk: kernel support for configurable objects and invocations. *Distributed Systems Engineering Journal*. Expected to be out early 95.

[39] Brad A. Myers. INCENSE: A system for displaying data structures. *Computer Graphics*, 17(3):113, July 1983.

[40] D.M. Ogle, K. Schwan, and R. Snodgrass. Application-dependent dynamic monitoring of distributed and parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):762–778, July 1993.

[41] Daniel A. Reed, Ruth A. Aydt, Roger J. Noe, Keith A. Shields, and Bradley W. Schwartz. *An Overview of the Pablo Performance Analysis Environment*. Department of Computer Science, University of Illinois, 1304 West Springfield Avenue, Urbana, Illinois 61801, November 1992.

[42] Sekhar R. Sarukkai and Dennis Gannon. Parallel program visualization using SIEVE.1. In *International Conference on Supercomputing*. ACM, July 1992.

[43] Karsten Schwan, Prabha Gopinath, and Win Bo. CHAOS – kernel support for objects in the real-time domain. *IEEE Transactions on Computers*, C-36(8):904–916, July 1987.

[44] Karsten Schwan and Anita K. Jones. Flexible software development for multiple computer systems. *IEEE Transactions on Software Engineering*, SE-12(3):385–401, March 1986.

[45] Karsten Schwan, Rajiv Ramnath, Sridhar Vasudevan, and Dave Ogle. A system for parallel programming. In *9th International Conference on Software Engineering, Monterey, CA*, pages 270–282. IEEE, ACM, March 1987. Awarded best paper.

[46] Karsten Schwan, Rajiv Ramnath, Sridhar Vasudevan, and David Ogle. A language and system for the construction and timing of parallel programs. *IEEE Transactions on Software Engineering*, 14(4):455–471, April 1988.

[47] Richard Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems*, 6(2):157–196, May 1988.

[48] John T. Stasko. TANGO: A framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39, September 1990.

[49] John T. Stasko and Eileen Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, June 1993.

[50] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.

[51] T. K. Xia, Jian Ouyang, M. W. Ribarsky, and Uzi Landman. Interfacial alkane films. *Physical Review Letters*, 69(13):1967–1970, 28 September 1992.

[52] Dror Zernik and Larry Rudolph. Animating work and time for debugging parallel programs – foundation and experience. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 46–56, Santa Cruz, California, May 20-21 1991. ACM Press. ACM SIGPLAN NOTICES 26(12), December 1991.

Appendix A: A Complete List of Cthreads Events Reordering Rules

Here is the terminology used in describing the ordering rules:

```
<- = "is allowable if preceded by"
t  = thread number
c  = condition number
m  = mutex number
n  = sequences number
pt = parent thread
ct = child thread
jt = join_to thread number
bn = beginning sequence number
en = ending sequence number
p  = processor number
x  = don't care value
```

The ordering rules for all events from default monitoring of Cthreads programs are listed below. A brief explanation for each rule is provided. For each mutex number `m` and condition number `c`, it is initially set to 0.

```
thread_init t          <- ();
```
This is the initial event for thread `t`. All prior events pertaining to this thread are ignored. An internal buffer is created for this thread number, and it is turned "on".

```
thread_fork pt ct      <- ( (thread_init pt) && ((pt == 0) && (thread_fork t pt)) );
```
The parent thread must be "on" for this event to be processed. An internal buffer is created for the child thread and it is turned "on". It is required that the parent thread is initialized before this event.

```
thread_exit t          <- ( (thread_init t) && (thread_fork pt t) );
```
The internal buffer is de-allocated and the thread is turned "off". Any succeeding events recorded by this thread are ignored.

```
thread_begin_join t jt <- (thread_init t);
```

```
thread_end_join t jt   <- ( (thread_init t) && (thread_exit jt) );
```
The `thread_exit jt` event for the join_to thread `jt` must have occurred before this event.

```
thread_detach t        <- (thread_init t);
```

```
thread_yield t         <- (thread_init t);
```

```
thread_set_name t      <- (thread_init t);
```

```
mutex_init t m         <- ( (thread_init t)
                            && !( (mutex_init x m) || (mutex_alloc x m) ) );
```
No prior `mutex_init x m` or `mutex_alloc x m` event may have occurred.

```
mutex_alloc t m        <- ( (thread_init t)
                            && !( (mutex_init m) || (mutex_alloc m) ) );
```
No prior `mutex_init x m` or `mutex_alloc x m` event may have occurred.

```
mutex_begin_lock t m n <- ( (thread_init t)
                            && ( (mutex_init x m) || (mutex_alloc x m) ) );
```
A `mutex_init x m` or `mutex_alloc x m` must precede this event.

```
mutex_end_lock t m n   <- ( (thread_init t)
                           && ( (mutex_init x m) || (mutex_alloc x m) )
                           && (mutex_end_lock x m n-1) )
```

A `mutex_init x m` or `mutex_alloc m` must precede this event. The `mutex_end_lock m, n-1` must have occurred. The initial value of this term is `mutex_end_lock x m 0`, which is always true.

```
mutex_unlock t m n     <- ( (thread_init t)
                           && ( (mutex_init x m) || (mutex_alloc x m) )
                           && !(mutex_end_lock x m n+1) );
```

A `mutex_init x m` or `mutex_alloc x m` must precede this event. The `mutex_end_lock m, n+1` may not have occurred.

```
mutex_free  t m       <-  (thread_init t);

mutex_clear t m       <-  (thread_init t);

mutex_set_name t m    <-  (thread_init t);

condition_alloc t c   <-  ( (thread_init t)
                           && !(condition_init x c || condition_alloc x c) );
```

No prior `condition_init x c` or `condition_alloc x c` event may have occurred before this one.

```
condition_init t c    <- ( (thread_init t)
                           && !( (condition_init c) || (condition_alloc c) ) );
```

No prior `condition_init x c` or `condition_alloc x c` event may have occurred before this one.

```
condition_free t c   <- (thread_init t);

condition_clear t c  <- (thread_init t);

condition_begin_wait t c n m <- ( (thread_init t)
                                  && ((condition_alloc x c) || (condition_init x c))
                                  && (condition_begin_wait t c n-1 m) );
```

A `condition_init x c` or `condition_alloc x c` event must have occurred, and so do the preceding `condition_begin_wait t c n-1 m` event.