Submitted to the IEEE TPDS – November 1994
Revised September 1995

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

Weiming Gu, Greg Eisenhauer, Karsten Schwan, Jeffrey Vetter

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Abstract – Advances in high performance computing, communications, and user interfaces are enabling developers to construct increasingly interactive high performance applications. The *Falcon* system presented in this paper supports such interactivity by providing runtime libraries, tools, and user interfaces that jointly permit the on-line monitoring and steering of large-scale parallel codes. The principal aspects of Falcon described in this paper are its abstractions and tools for capture and analysis of *application-specific* program information, performed *on-line*, with *controlled latencies* and scalable to parallel machines of substantial size. In addition, Falcon provides support for the on-line graphical display of monitoring information, and it allow programs to be steered during their execution, by human users or algorithmically. Falcon also promotes the use of on-line monitoring for purposes other than performance debugging and it permits developers and end users to experiment with alternative program configurations, to play 'what if' games with selected program attributes, and to improve program performance by reacting to runtime changes in program behavior. This paper presents our basic research motivation, outlines the Falcon system's functionality, and includes by a detailed evaluation of its performance characteristics in light of its principal contributions. Falcon's functionality and performance evaluation are driven by our experiences with large-scale parallel applications being developed with end users in physics and in atmospheric sciences. The sample application highlighted in this paper is a molecular dynamics simulation program (MD) used by physicists to study the statistical mechanics of liquids.

Index Terms – Parallel processing, program steering, program adaptation, on-line monitoring, instrumentation, trace analysis, perturbation, performance evaluation, performance displays, molecular dynamics simulation.

---

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

1 Introduction

Recent advances in high performance computing and communications are permitting large-scale applications to make simultaneous use of multiple networked parallel machines and workstations as their computational, graphical display, and input/output engines. Increased computational performance, more mature system software, and higher network bandwidth are also allowing end users to execute applications interactively rather than in the traditional batch execution mode. Our research contributes to interactive high performance computing by providing tools for the class of program interactions termed *program steering*. Specifically, the Falcon system supports *interactive program steering*, or the on-line configuration of a program by human users, with the purpose of affecting the program's execution behavior. Falcon also supports program steering performed by on-line algorithms, typically called *program adaptation*[5, 6].

Program steering is useful to end users and developers alike because it permits them to explore alternative computational methods, data domains, and parameter settings. Program steering, either in conjunction with or solely performed by on-line algorithms, can also be used to improve program performance by reacting to dynamic changes in the program's computational or input/output behaviors[7]. In the long term, we expect program steering technologies to used in the broader context of *distributed laboratories*, where multiple and physically distributed end users can collaborate with each other as if they were co-located in a single laboratory setting. In this distributed setting, the users can solve complex scientific or engineering problems by jointly experimenting with multiple coupled and distributed scientific simulations (*i.e.*, 'instruments'), all of which may be monitored and steered by the users and with algorithms. Such steering may be used for many purposes, including playing 'what if' games by varying input sets, or improving simulation results by adjusting program parameters, or performing both such actions in order to validate simulation results in reference to observational data.

The *Falcon* system is our first step toward realizing the general vision of distributed laboratories and interactive distributed programs. Falcon is a set of tools that jointly support three tasks. The first task is the on-line capture, collection, and analysis of the application-level program and performance information required for program steering and for display to end users. The second task is the analysis, manipulation, and inspection of such on-line information, by human users and/or programs, based on which decisions concerning program steering may be made. The third task is the support of steering decisions and actions, which typically result in on-line changes to the program's execution. These changes may range in complexity from modifications of a few selected application parameters to concerted changes of complex run-time program states. The time-scale of these changes can vary from rapid modifications to the implementation of single program abstractions (*e.g.*, a single mutex lock in a parallel code[40]) to the occasional modification of program attributes by end users (*e.g.*, load balancing in a large-scale scientific code as described in Section 2.2

below).

The Falcon system's development is part of an ongoing research effort involving the development of large-scale scientific and engineering applications[9, 26], of abstractions and tools for program development and evaluation, and of mechanisms for increasing such programs' utility by making them interactively accessible to end users. The research contributions described in this paper focus on the monitoring component of Falcon:

- *Application-specific monitoring* – in addition to providing default program information, we permit users to capture and analyze application-specific program information, ranging from information about single program variables to program states defined by complex expressions involving multiple program components. These capabilities are especially useful for non-computer science end users, who wish to view, analyze, and steer their applications in terms 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 to scale to target machines of differing sizes.

- *On-line analysis* of captured program information based on which the program may be *steered* or *adapted* – monitoring information captured with Falcon may be attached to arbitrary user-provided analysis code, graphical views for output or program steering, and adaptation algorithms. Analyses may employ statistical methods, boolean operators like those described in [41], or simply reorder the events being received. Graphical views may be displayed with multiple media or systems, currently including X windows, Motif, and the SGI Explorer environment.

For brevity, this paper does not elaborate on two additional aspects of Falcon, which are (1) the support of *multiple heterogeneous computing platforms* – current extensions of Falcon address both single parallel computing platforms running threads-based programs as well as distributed computational engines using PVM and Unix sockets as software bases (*e.g.*, for Falcon's use on the IBM SP-2 platform) – and (2) the provision of default graphical performance displays and of tools for the construction of application-specific displays for program monitored using Falcon. Specifically, Falcon offers several default on-line graphical animations of the performance of threads-based parallel program (see [15]). In addition, Falcon uses the Polka system for program animation which provides users with easy-to-use tools for creating application-specific 2D animations of arbitrary program attributes[48]. Furthermore, Falcon is now being used for experimenting with alternative parameter settings in a large-scale atmospheric modeling application[26], using interactive 3D data displays via the Silicon Graphics Explorer environment[23].

Falcon emphasizes low latency, on-line monitoring, performed by capturing only those program attributes required for specific performance analyses or for specific program steering. This distinguishes our work from related research on performance monitoring and tuning, including that of Reed[42] and Miller[21], both of which generally address the issue of performance debugging using program traces stored in intermediate files. These projects' primary concern is not the latency with which program events are transferred from the program to the end user (*i.e.*, to an interactive user interface or to an adaptation algorithm). Instead, they focus on reducing or controlling program perturbation due to performance monitoring[32]. A further distinction between Falcon and other projects on performance debugging [37, 30, 3] is Falcon's support of application-specific monitoring. Such support is essential when end users wish to use monitoring output to steer their programs or to simply understand their runtime behaviors in terms of familiar quantities (*e.g.*, total energy in the MD application).

Related research in program steering (*e.g.*, the Vase system[22]) differs from our work in its focus on steering by human users. In contrast, because Falcon supports both algorithmic and interactive program steering, it emphasizes monitoring latencies and overheads more than strictly human interactive systems like Vase. By offering low monitoring latencies, interesting program events may be recognized with suitable delays for adaptation algorithms' or human users' corrective actions. The latency requirements imposed on Falcon are made precise by on-line configuration of a sample high performance application, a molecular dynamics code constructed jointly with physicists.

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 an overview of the Falcon system, and it describes its implementation addressing the on-line monitoring and steering of threads-based multiprocessor programs. The performance of this implementation is evaluated in Section 4, followed by a more detailed description of related research in Section 5. The final section presents our conclusions and future research.

2 Motivation

Monitoring and steering may be utilized for understanding and improving program performance and for experimenting with program characteristics whose effects are not easily understood. For example, atmospheric scientists working with our group utilize program steering to reduce turnaround time when determining certain model parameter settings such that simulation outputs match observational data[26]. However, our work has often been more concerned with performance improvement by on-line program adaptation, including demonstrating significant performance gains [39] through on-line configuration of mutex lock implementations in threads-based multiprocessor programs. Similarly, object-based mechanisms for on-line program configuration are described in [5, 13, 10], where program improvements concern the dynamic adjustment of timing, performance, and reliability properties in response to changes in application needs or in characteristics of the execution environment. Other examples of the utility of program steering include the automatic configuration of small program fragments for maintaining real-time response in uniprocessor systems and the load balancing or program configuration for enhanced reliability in distributed systems[29, 43, 33].

Many of the research results listed above concern performance improvements attained by program steering. It is more difficult to demonstrate enhanced utility via steering, especially when such promises are based on expected increases in the effectiveness of end users. In general, we address this issue by exploring the use of Falcon with sample applications developed jointly with end users in physics[9] and atmospheric sciences[26]. This paper describes experimental results derived from our work with physicists on a molecular dynamics simulation (MD). In order to provide a context for later discussion of monitoring and steering this simulation, the remainder of this section describes the MD code, discusses its potential for utilizing program steering, and identifies the runtime support required for such steering. These requirements form one basis on which Falcon's functionality is evaluated.

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 [49, 8]. In this paper, we consider a physical system which 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. Specifically, 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 processors must communicate and synchronize between simulation steps.

2.2 Steering MD – Experimentation and Results

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

1. Decomposition geometries may be changed in response to changes in the physical system. For example, a slab-based decomposition is useful for an initial system, but a pyramidal decomposition may be a more appropriate choice when a probe is lowered into the simulated physical system.
2. The on-line modification of the cutoff radius can improve solution speed by computing uninteresting time steps with some loss of fidelity. End user interactions are essential for such modifications since judgments must be made concerning acceptable speed/fidelity tradeoffs.
3. The boundaries of spatial decompositions can be shifted for dynamic load balancing among multiple processes operating on different sub-domains, performed interactively or by on-line configuration algorithms.
4. 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.

Of these on-line program changes (2)-(4) are easily performed; the program's implementation already permits (3) to be easily varied, and (2) and (4) involve modification of a few variables of branch instructions in the MD code. In this code, changes in decomposition geometries are not very difficult to perform since such geometries are already explicitly described via data structures. This may not be the case for other implementations of MD simulations.

To demonstrate the potential utility of program steering, we next review some results of MD steering applied to the problem of improving system load balance. In particular, we examine the behavior of the MD simulation when the 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 processor. This is because the complexity of MD computation depends not only on the number of particles assigned to each processor but also on particle distances (due to the use of a 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 initially 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 monitor

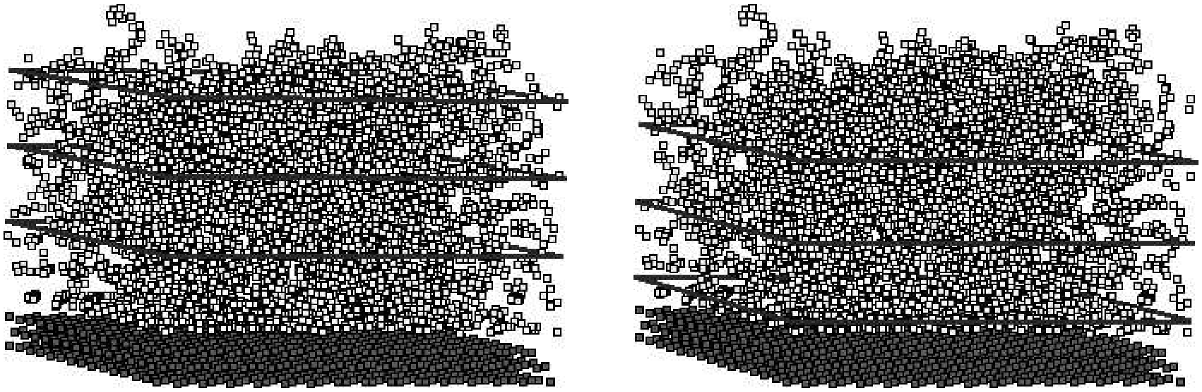
(a) Initial Decomposition (b) Balanced Decomposition

Figure 1: 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.

load balance over time and then steer the application code to adjust load balance (by adjusting domain boundaries) throughout the application's execution. For this example, we assume that such steering is performed interactively by end users. In the future, we are partially automating steering such that end users are required to interact with the application only when automated steering is not successful.

The interactive steering of MD uses the Falcon system to monitor process loads on-line, and to display workloads in bar graph form (see Figure 2). In addition, the MD code itself performs the 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 2 and 1, respectively, for a sample simulation run with four domains on four processors. In Figure 1, part (a) depicts the initial decomposition (domain boundaries are indicated by horizontal lines) for a certain program run, whereas part (b) depicts the final decomposition attained by explicit user manipulations of the domain boundaries indicated. In this example, such manipulations are performed using a textual user interface that permits users to change domain boundaries while the program is running, and the program is written to enable such on-line changes. The actual load imbalances experienced for the initial decomposition are depicted in Figure 2. From this figure, it is apparent that thread 0 (computing domain 0) is overloaded, while thread 3 does not have sufficient work.

The effects of dynamic steering when used to correct load imbalances can be quite dramatic, as shown in Figure 3 for the same initial and final configurations as those shown in Figures 1 and 2. For this sample run, several successive steering actions significantly improve program performance by repeated 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 interactive steering, rather than degrade performance

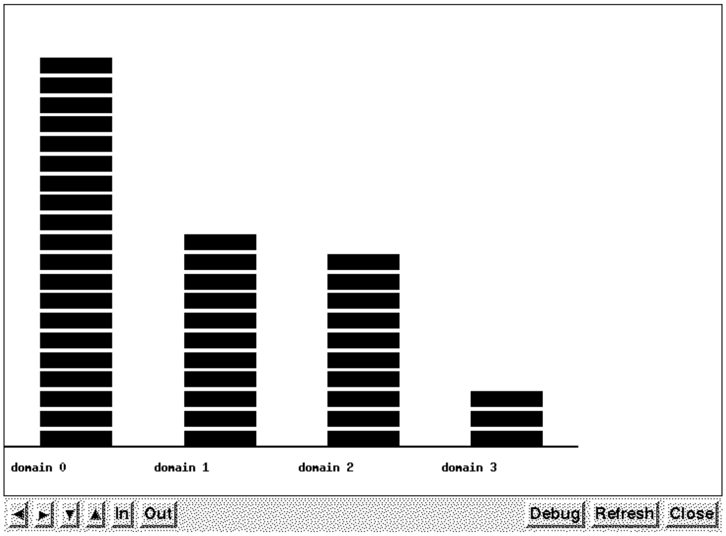

Figure 2: The load balance view of MD for its initial 4 processor configuration depicted in Figure 1, part (a). The vertical dimension depicts the a running average of the execution time of thread n executing domain n of the MD simulation.

due to the additional costs imposed by steering and monitoring on the parallel program's execution. Second, it should be apparent that this example's user interactions with the code can be replaced or at least assisted by steering algorithms used on-line, thereby partly automating steering. By permitting users to develop such algorithms and interactively employ them, they are given the ability to migrate their experiences and experimental knowledge about the application's runtime behavior into their application codes, without requiring extensive program changes. Third, and more broadly, these results indicate the potential of program steering for helping end users experiment with and understand the behaviors of complex scientific applications.

2.3 The Requirements of Steering

To realize the potentials of program steering presented in Section 2.2, several assumptions must be made, some of which may be removed or ameliorated by future work. First, program steering requires that application builders must write their code so that steering is possible. Second, users must provide the program and performance information necessary for making steering decisions. Third, steering cannot be successfully employed unless such information can be obtained with the latency required by the desired rate of steering. Concerning the first requirement, in MD, domains are represented in such a way that their boundaries are easily shifted to permit steering for improved workload balance. In general, however, programs can be made steerable only by requiring end users to write them accordingly, by requiring substantial compiler support[44], or by requiring that the programming language offer stronger mechanisms of abstraction than those existing

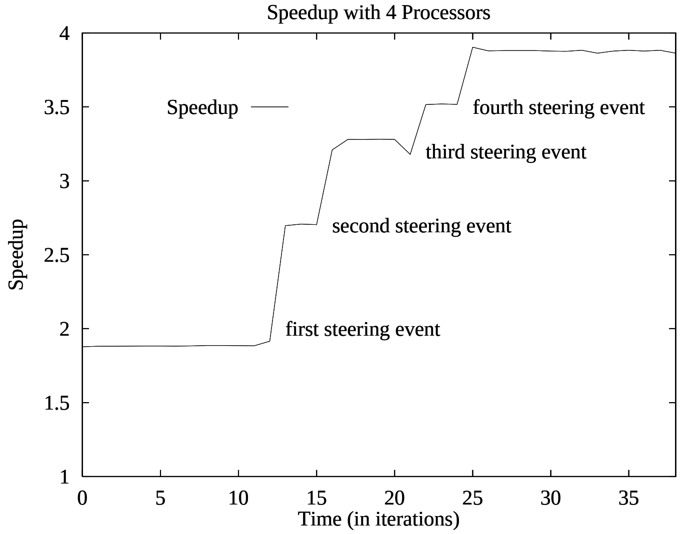

Figure 3: The effect of steering on performance over time with 4 processors.

in parallel Fortran or in the Cthreads library used in our work (*e.g.*, the object model[5, 10, 29, 13]).

The primary concerns of this paper are the second and third requirements for program steering. The on-line capture, analysis, and display of information about current program behavior and performance, at the rates required for program steering is of particular importance. For interactive steering, such information is presented by graphical displays or simple textual output. Sample displays for steering MD include the on-line data visualizations depicting molecular distributions (see Figure 1) and the associated current values of workload across different data domains (see Figure 2). For algorithmic steering, captured information is provided to steering algorithms. Many such algorithms have been described in the literature (see Section 5 and [17]), each requires information specific to the application and/or to the steering actions being performed. For example, one algorithm developed in our own work attempts to improve program performance by dynamically configuring mutex lock implementations for programs running on shared memory machines[39]. This algorithm requires the capture of small amounts of program information (*i.e.*, the average waiting times experienced by threads on individual mutex locks) with low latencies and high rates attainable only with the sampling techniques described and used in Section 4.3 below.

The third requirement of on-line steering recognizes that steering is effective only if it can be performed at a rate higher than the rate of program change. In the case of dynamic load balancing by shifting domain boundaries in MD, the rates of change in particle locations are sufficiently low so that human users can detect load imbalances and shift domain boundaries. However, when steering is used to dynamically adjust lock waiting strategies as in [39], changes in locking patterns must be detected and reacted to every few milliseconds. As a result, any on-line monitoring support for program steering must provide information necessary for steering with low latency. The Falcon system attempts to provide these capabilities.

3 The Design and Implementation of Falcon

This section presents the specific design goals for the Falcon system, an overview of the system's architecture, and detailed discussion of each component of the Falcon system.

3.1 Design Goals

Three attributes of Falcon are designed to address the requirements of on-line program steering. First, Falcon supports the *application-specific monitoring/steering, analysis, and display* of program information, so that users can capture, process, understand and steer those program attributes relevant to the problem at hand, be it a dynamic program modification or a specific performance problem being diagnosed or investigated.

Second, because systems like Falcon cannot design steering algorithms or determine appropriate rates at which steering should be performed, Falcon's role should be to enable a wide variety of possible steering actions. Accordingly, Falcon must provide users with the ability to *reduce or at least control monitoring latency* throughout the execution of a parallel program, such that they may also maintain acceptable workloads due to monitoring imposed on the underlying parallel machine. Dynamic control of monitoring latency and workload is important because the effectiveness of program steering generally depends 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 may not only affect the order of occurrences of program events, but can also offset performance gains achieved by steering. In response to these needs, the Falcon system can be configured by users to control its performance and associated resource usage.

A third attribute of Falcon is its support for *scalable monitoring*. This is achieve by permitting changes to the resources consumed by its runtime system in response to machine size and program needs. In Section 4, we show that Falcon can be used to monitor programs of varying sizes and with varying monitoring latencies and rates when executing on different subsets of a 64-node KSR multiprocessor.

3.2 System Overview

Falcon is constructed as a toolkit that collectively supports the on-line program monitoring and steering of parallel and distributed programs. The Falcon toolkit consists of tools for monitoring and steering specification and instrumentation, mechanisms for on-line information capture, collection, filtering, analysis, and storage, mechanisms for program steering, and a graphical user interface and several graphical displays for interfacing with end users. The major components of Falcon are shown in Figure 4.

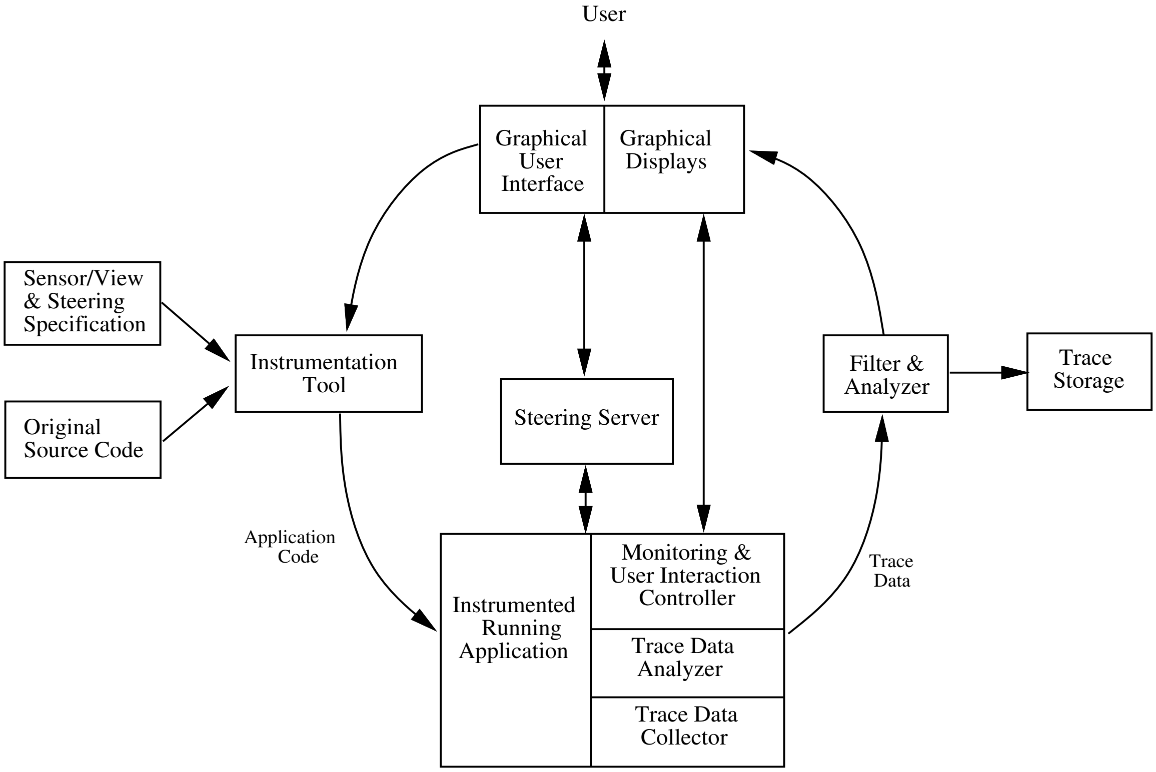

Figure 4: Overall architecture of Falcon.

To understand Falcon's functionality, consider the steps taken when using Falcon to construct a steerable program. First, the application code is instrumented with the sensors and probes generated from sensor and view specifications by users who understand both the application code and its instrumentation needs. Falcon supports this task by providing monitoring specifications and compiler support that permit users to express specific program attributes to be monitored and on which steering may be performed. Users knowledgeable about the application code and its instrumentation needs then include the stubs generated from those specifications with application code[2]. During program execution, program and performance information of interest to the user and to steering algorithms is captured by the inserted sensors and probes, and the information is partially analyzed. Falcon's on-line monitoring facilities consist of trace data output queues attaching the monitored user program to a variable number of additional components performing low-level processing of monitoring output. Falcon's graphical user interface, the graphical displays, and the steering mechanism directly interact with the runtime system to obtain the partially processed monitoring

---

[2] Stub insertion may not be trivial. Future work should address compiler and user interface support for this task, as well as additional functionality in the stub compiler to generate alternative stub implementations[41].

information. Further analysis of the trace information is performed before it is displayed to end users or used in steering algorithms. Trace information can also be stored in trace files for postmortem analysis. Once steering decisions are made by the end user or a steering algorithm, changes to the application's parameters and states are performed by Falcon's steering mechanisms.

The monitoring and user interaction controller and the steering server are part of Falcon's runtime system. They activate and deactivate sensors, execute probes or collect information generated by sampling sensors, maintain a directory of program steering attributes, and react to commands received from the monitor's user interface. For performance, the monitoring and user interaction controller is physically divided into several *local monitors* and a *central monitor*. The local monitors and the *steering server* reside on the monitored program's machine, so that they are able to rapidly interact with the program. In contrast, the central monitoring 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[48]. Several performance or functional views (*e.g.*, the bar-graphs in Figure 2) 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 is also able to interact with custom displays and with systems supporting the creation of high-quality 3D visualizations of program output data, such the SGI Explorer tools.

3.3 Implementation Description

This section explores selected implementation attributes of Falcon (1) to explain how Falcon attains its goals of application-specific monitoring, controlled monitoring overheads, and monitoring scalability, (2) to delimit the utility of Falcon in terms of its offered functionality and its associated performance characteristics, and (3) to demonstrate that the implementation of Falcon may be ported to a wide variety of target platforms.

Falcon's implementation relies on a Mach-compatible Cthreads library[38] available on a variety of hardware platforms, including the Kendall Square Research KSR-1 and KSR-2 supercomputers, the Sequent multiprocessor, uni- and multi-processor SGI, SUN SPARC, IBM RS6000 workstations, and various Linux machines. Falcon's implementation structure is depicted in Figure 5. This implementation is discussed in detail in the context of Falcon's specific contributions 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 Falcon's monitoring specification

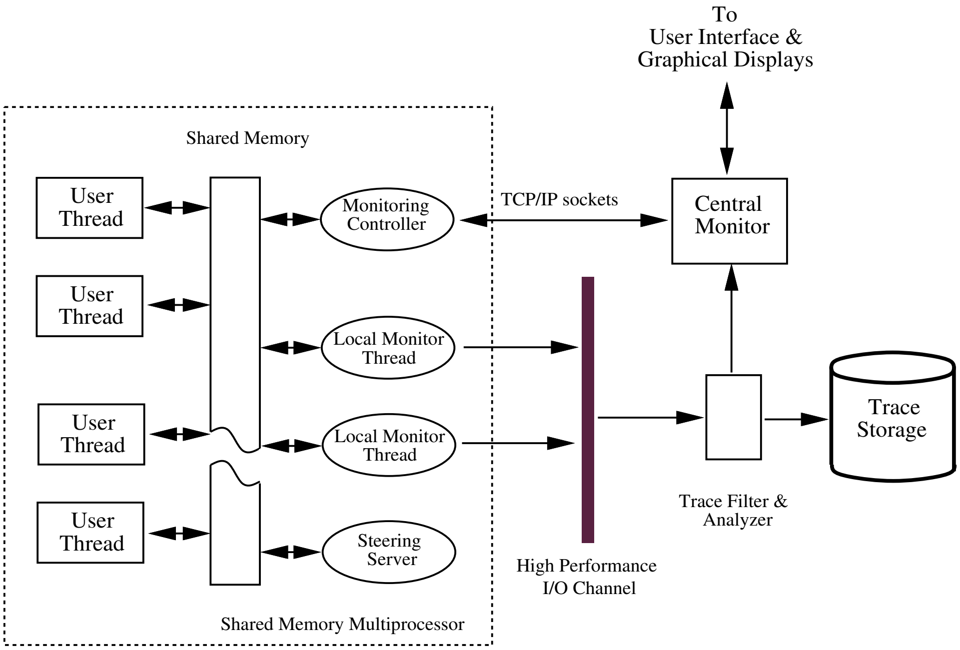

Figure 5: Implementation of the monitoring mechanism with Cthreads.

language, programmers may define application-specific *sensors* for capturing the program and performance attributes to be monitored and based on which steering may be performed. The specification of a sample *tracing sensor* is shown below:

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

The sensor `work_load` is used to monitor the work load of each molecular domain partition in the MD application. The specification simply describes the structure of the application data to be contained in the trace record generated by this sensor. This declaration generates the following sensor stub:

```
int
user_sensor_work_load(int domain_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.domain_num = domain_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;
            }
        }
    }
```

The body of this stub 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 (*i.e.*, the monitoring system itself may be dynamically steered). 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 displays of total program execution time.

Controlling monitoring overheads – sensor types and sensor control. The monitoring overheads experienced when extracting program information may be controlled by use of different sensor types: sampling sensors, tracing sensors, or extended sensors. A *sampling sensor* is associated with a counter or an accumulator located in the shared memory periodically accessed by the local monitors. When a sampling sensor is activated, the associated counter value is updated. A *tracing sensor* generates timestamped event records that may be used immediately for program steering or stored for later analysis. In either case, trace records are stored in *trace queues* from which they are removed by local monitoring threads. An *extended sensor* is similar to a tracing sensor except that it also 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. Among the three types of sensors, sampling sensors inflict the least overhead on the target application's execution. 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[3].

In order to control monitoring loads, sensors can be controlled dynamically and selectively to monitor only the information currently being used by the end user or the steering algorithms. Sensors may be turned

---

[3] Falcon also permits the inclusion of *probe* code in local monitors, which may be used to inspect program variables asynchronously to the program's execution and without requiring prior instrumentation of the program's code.

off if events captured by those sensors are not currently used by the end user or the steering algorithm.[4] Sampling and tracing rates can also be dynamically reduced or increased depending on monitoring load and tolerance of inaccuracies in monitored information. For example, a tracing sensor that monitors a frequently accessed mutex lock can have its tracing rate reduced to every five mutex lock accesses, thereby improving monitoring perturbation at the cost of reducing trace accuracy. A selective monitoring example can be found in the MD code, where a large amount of execution time is spent in a three-level nested loop computing forces between particles. At each loop level, distances between closest points of particles and bounding boxes of molecules are calculated and compared with the cutoff radius to eliminate unnecessary computations at the next loop level where specific particles are considered. To evaluate the efficiency of this scheme, at each loop level, we use a "cheap" sampling sensor to monitor the hit ratio of distance checks and a more "expensive" tracing sensor to monitor the correlations between the calculated distance and hit ratio at the next loop level. To reduce the perturbation, the "expensive" tracing sensor is not turned on until ineffective distance checks are detected. The performance of such selective monitoring is analyzed in Section 4.

Controlling monitoring overheads – concurrent monitoring. Local monitoring threads perform trace data collection and processing concurrently and asynchronously with the target application's execution. As depicted in Figure 5, the local monitors 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 can be predicted fairly well (results on the KSR-2 are presented in Section 4), and therefore, its effects on the correctness of timing information can be eliminated using known techniques for perturbation analysis[32].

Falcon's runtime monitoring 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 the 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

---

[4] Related work by Hollingsworth and Miller [21] removes instrumentation points in order to completely eliminate the overheads of such 'turned-off' instrumentation points.

detected by a simple monitor-monitor mechanism, new local monitoring threads 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 are 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.

Distributed on-line trace data analysis. Falcon offers a distributed on-line trace data analysis mechanism; trace data is processed in different physical components of the monitoring system. At the lowest level, simple trace data filtering and analysis may be performed by extended sensors. At the local monitor level, trace data is further analyzed to produce high level information. The partially processed monitoring information 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 to end users, and can also be stored in trace data files for postmortem analysis. The central monitor, user interface, and graphical displays may reside on a different machine to reduce interference from monitoring activities to the target application's execution, and to capitalize on efficient graphics libraries and hardware and data analysis tools existing on modern workstations. The current implementation of Falcon assumes that programmers instrumenting the program determine where analysis actions are taken. An interesting topic for future research is the automatic and adaptive determination of where such analyses are performed by dynamically 'shifting' analysis functions among participating parties (*i.e.*, extended sensors, local monitors, global monitor, and additional processes performing analysis tasks).

The performance of Falcon's on-line monitoring system is presented in Section 4. Prior to that presentation, we discuss the portability of Falcon and the limitations of its current implementation. In addition, we briefly describe Falcon's steering mechanisms and its on-line display system, which consume the information obtained from the on-line monitoring system.

Portability and limitations. The current implementation of Falcon relies on the availability of Georgia Tech Cthreads[38] on the desired target machine and on the availability of shared memory between application processes and local monitoring threads. Local monitoring threads are written as Cthreads programs and rely on Falcon's event buffering mechanisms for event transfer. These buffering mechanisms require memory to be shared between the application and the local monitoring thread. For local monitoring threads to run on a different processor from the application in a multicomputer, the processors must share memory via hardware or software. On a uniprocessor, the local monitoring thread runs as a separate process and it must be possible for two processes to share some portion of their virtual memory space. Since the GTthreads library has been ported to many useful target machines, all of which offer shared memory support, neither (1) nor (2) significantly constrain Falcon's portability. However, the control of monitoring latencies and overheads in Falcon relies in part on the underlying operating system's ability to execute application threads asynchronously with local monitoring threads, so that varying amounts of computational resources may be allocated to both. Such resource allocation is straightforward on uniprocessor platforms offering explicit scheduling support (*e.g.*, prioritized threads in Solaris) or on multiprocessors platforms (like the KSR and SGI machines) offering the ability to execute different processes or threads on different processors, but it is not easily performed on target systems where users are given no control over the allocation of available processor resources to different application components (*e.g.*, SUNOS). As a result, in order to attain meaningful experimental results concerning controlled monitoring overheads, this research exploits the target KSR machine's ability to control resource allocation by binding certain application processes or threads to different processors. On target machines not offering any user-level resource control, end users are reduced to exploiting Falcon's alternative mechanisms for capture and analysis, such as sampling vs. tracing sensors, and extended sensors.

Lower bounds on the latency with which information capture may be performed (on the relatively slow KSR machine's processors and memory) with Falcon are described in Section 4.1 below. For further latency reductions, additional sensor compiler functionality must be provided, perhaps generating alternative representations of stubs generated from sensor specifications or even using on-line recompilation and relinking techniques like those described for the Synthesis operating system[34].

In summary, while Falcon's functionality is easily provided on a wide variety of target machines and platforms, its mechanisms for controlling monitoring overheads and latencies rely in part on the availability of parallelism in the underlying machine. Any constraints on the portability of Falcon derived from this requirement should be of decreasing importance, given the increasing numbers of SMPs used as both standalone systems or as nodes in larger scale parallel machines.

3.4 Program Steering

The purpose of this section is to demonstrate that Falcon's program steering mechanisms are natural extensions of its monitoring support. This fact is due to two attributes of Falcon: (1) the application-specific nature of Falcon's monitoring support permits users to perform monitoring and steering for exactly those program attributes required for specific steering actions, and (2) the mechanisms dynamically controlling how monitoring is performed provide a natural basis for implementation of steering actions.

Figure 6 depicts some internal features of steering as well as its relationship with other components of Falcon. Similar to local and central monitors, a *steering server* on the target machine performs steering, and a *steering client* provides the user interface and control facilities remotely. As with local monitors, the steering server is a separate execution thread to which local monitors forward only those monitoring events

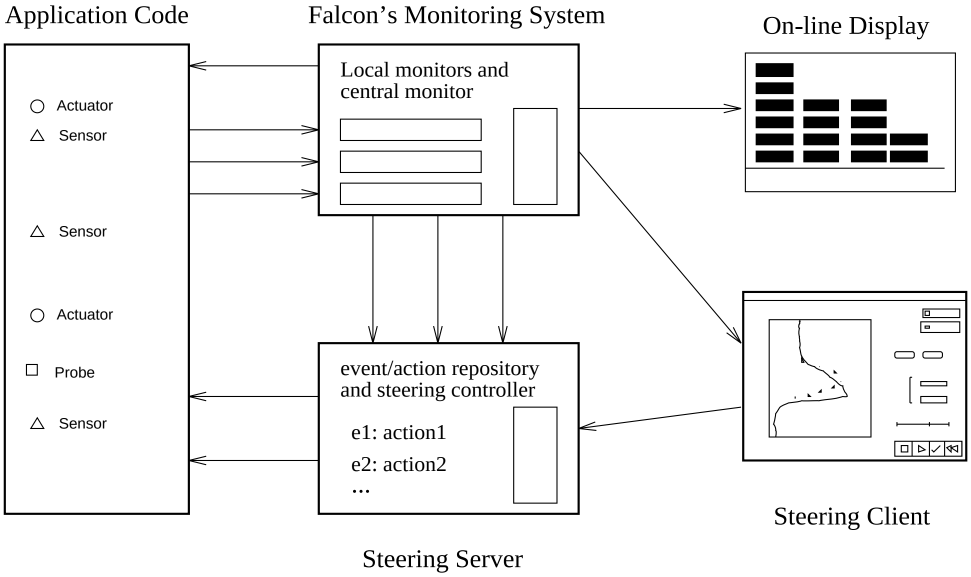

Figure 6: Overall structure of the steering system.

that are of interest to steering activities. Such events tend to represent a small proportion of the total number of monitoring events, in part because event analysis and filtering is done by local monitors rather than by the steering server itself.

Steering decisions are made based on program state captured via specific monitoring attributes and observed and analyzed by human users or steering algorithms. Therefore, the primary task of each steering server is to read incoming monitoring events and to respond with appropriate actions. These responses are based on previously encoded decision routines and actions, which are encoded in a steering event/action repository in the server. This repository contains entries for each type of steering event, specifying the appropriate action to take in response. The responses represented here may perform some actual steering action on the application, note the occurrence of some monitoring event for future reference, or probe for additional information by access to selected monitoring attributes, or simply forward the event or analyzed information to the client for display or further processing. 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 repository, and input steering commands directly from end users to the server.

Two principal abstractions are supported by the steering mechanisms. First, *program attributes* are the steering system's representation of modifiable program variables or characteristics As with an extended sensor, with each attribute is associated a procedure that operates on it. Second, by associating any number of such attributes with a program abstraction, developers can create a *steerable object*, which can be 'registered' with the steering system. This object appears to the steering system as one that exports a number of invokable methods, each of which concerns a specific modifiable program attribute. Therefore, steerable objects are similar to adaptable objects first developed for real-time systems[5] and to configurable objects developed for object-oriented operating systems[11]. Each steerable object can be "registered" with the steering system, which maintains a repository of all such objects and their methods in the steering server.

Falcon offers both synchronous and asynchronous modes for invoking the methods of steerable objects. A synchronous invocation permits the steering server to execute the attribute's method directly, called a *steering probe* (similar to the asynchronous probes of program variables or states performed by local monitors). Such an invocation is useful when a steering action may be performed independently of the program having reached certain execution states. An asynchronous method invocation modifying a certain attribute simply posts an *action* to an *actuator* implementing the attribute and embedded in the application itself. Such an actuator is a portion of code inserted into application code at locations deemed "safe" for steering actions. When actions are pending at the time the application executes the actuator, all such actions are performed by the actuator *in the context of* the application thread's execution. Therefore, while actuators are enabled asynchronously to the application's execution, they are executed in concordance with application code. This permits the steering system to relegate the responsibility for determining when certain steering actions may be enacted to the application developer, in contrast to previous work performed for real-time systems and described in [5, 12] where program adaptations are always performed in conjunction with method invocations on real-time objects.

Steering actions are composite operations to be performed by the steering system in response to requests from the user or to monitoring events generated by the program. Each such action may modify any number of program attributes, perform computations, and even initiate other actions. Steering actions are similar to existing models of event/action systems[4] in that they are triggered by the receipt of specific events. However, they differ from those models in that the information based on which steering actions are made may include any number of program characteristics inspected by those actions, of current or past program or environmental state collected for purposes of steering, and even user knowledge about desired application behavior.

In summary, Falcon's steering facilities are a natural extension of its basic monitoring support: (1) as with local monitors, a steering server is an additional thread spawned at the time of program initialization; it interacts with local monitors for gaining access to application state and with the application for enacting program changes; (2) as with probes vs. sensors for monitoring, steering probes enable the steering server to directly perform simple program changes, whereas more complex changes are enacted using actuators placed by developers into application programs. In contrast to Falcon's monitoring support, the exploratory nature of our research concerning program steering has not yet permitted us to develop compile-time support for program steering, such as language support placing constraints on possible steering or the integration of steering support with the application's programming language performed in the Vase system[22].

The performance evaluation of the steering library's current implementation presented in Section 4.4 demonstrates that the current implementation easily supports interactive steering. It also identifies the limits on possible rates with which steering may be performed due to minimum monitoring, decision making, and enactment delays implied by the library's implementation.

3.5 Falcon's On-line Display System

Graphical displays have been shown useful in presenting data structures, algorithms[46], runtime program behavior[31], and performance information[19, 42] to human users. However, most current work deals primarily with off-line graphical and animated presentations of program and performance information. Instead, Falcon supports the on-line use of displays to help users understand a target program's performance and runtime behavior, as well as to interactively steer their parallel codes. Moreover, Falcon's display support must permit the construction and simultaneous attachment of multiple displays that may be performance-relevant and/or application-specific. This is shown in Figure 7, where event streams from local monitors are routed to both types of displays via the system's central monitor. A sample performance display is the thread life-time view indicated at the right bottom of the figure and depicting the creation, execution, blocking, and deletion of program threads over time. A sample application-specific view depicting chemical concentrations in the atmosphere and accepting steering commands was developed for steering the atmospheric modeling application described in [26]. This view appears on the bottom right of Figure 6.

For brevity, this paper discusses only one interesting topic concerning on-line vs. off-line displays, which is the analysis of monitoring information such that displays depict actual program behavior rather than artifacts of the monitoring system's implementation. Additional information about how performance-relevant and application-specific graphical displays are constructed, how they are used for performance understanding and for gaining application-specific insights on program behavior, and how they are integrated with the rest of the Falcon system is available elsewhere[15, 14]. General methods for supporting performance understanding in program animation systems are described in [27, 47].

In Falcon, monitoring information captured for use by on-line displays may be analyzed at several different sites, including in extended sensors, local monitors, the central monitor, and in additional analysis packages interposed between central monitor and displays. The ability to distribute such analysis is important even for the thread life-time view supported by the system. In this case, analysis code must be placed into extended sensors and into the event-reordering filter (see Figure 7) interposed between the central monitor and the display. Analysis is performed in order to guarantee the *behavior-preserving* nature of the thread life-time display. A behavior-preserving display depicts only valid program behavior, even when current trace information captured by the monitoring system appears to indicate different program behavior.

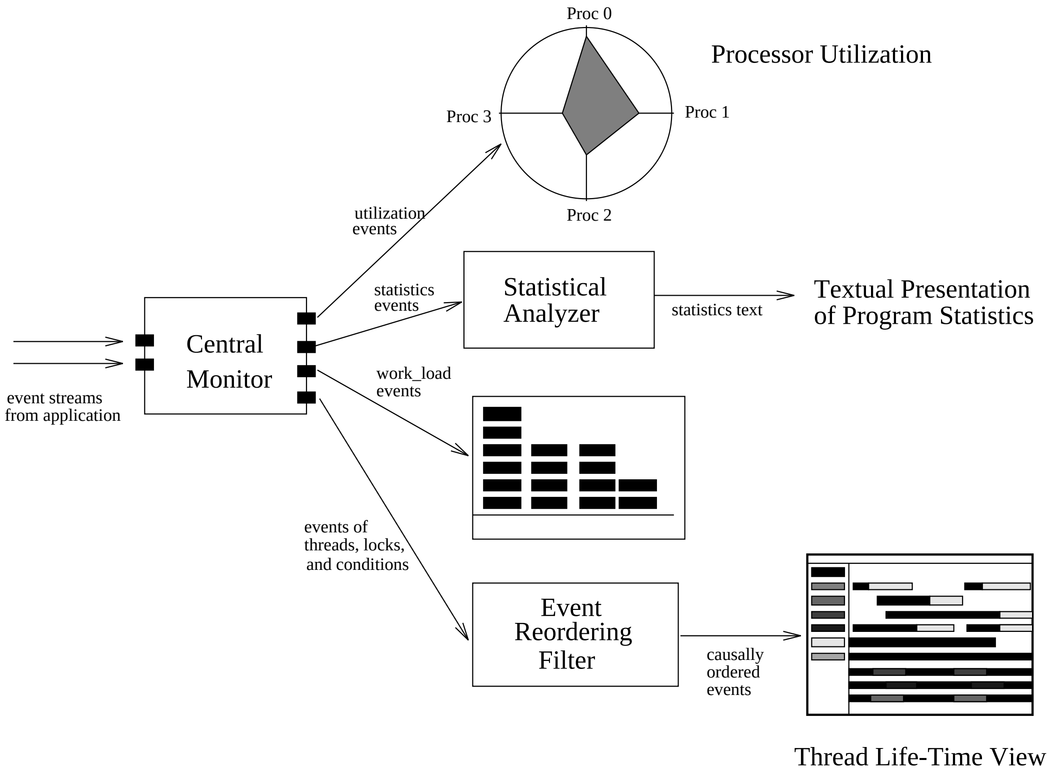

Figure 7: A sample on-line display system.

The on-line and distributed analysis of captured trace data is a necessary requirement for behavior preservation in the life-time view for two reasons. First, excessive program perturbation may arise when the rates at which local monitors process trace data differ too much from the rates at which sensors generate their input data. This causes the buffers placed between both to become full, which in turn results in program threads blocking and waiting on full buffers. If not 'caught' by perturbation analysis placed with tracing sensors, then a straightforward display of the resulting thread execution times would depict imbalanced thread execution times without indicating to end users the monitoring system's role in this imbalance. As already mentioned in Section 3.3, Falcon addresses this issue by inclusion of an implicit *perturbation field* with all events generated by tracing sensors. This field records the perturbation experienced by threads due to blocking on monitoring buffers. Interestingly, the principal additional overheads on tracing arising from this field's presence are the slight increase in event record size, since on-line analysis computing actual perturbation is performed only when program threads already experience non-zero waiting times on the shared buffers.

The second need for on-line analysis concerning behavior preservation in the life-time view derives from the distributed nature of Falcon's event capture mechanisms. Specifically, in any parallel or distributed system, it is difficult to guarantee that the monitoring system's method of event collection preserve the actual time ordering of events being produced at the time those events are received by the on-line display. In Falcon, since monitoring events are first buffered on the parallel machine and since local monitoring threads are not perfectly synchronized, events are not guaranteed to be in time order when received by the central monitor and ultimately, by analysis and display packages. As a result, trace-file based monitoring systems[42] sort such files prior to displaying the events they contain. For on-line monitoring, it must be possible to construct and then include with the event stream temporary event storage and reordering routines. Such analyses are performed in the event reordering filter constructed for the thread life-time view. Its implementation and use for performance understanding are discussed in detail elsewhere[27, 47, 15].

To summarize, Falcon supports the on-line display of captured program information in two ways: (1) by permitting the placement of necessary analysis code at any level of its event capture hierarchy and (2) by facilitating the attachment of multiple performance-relevant and application-specific displays to captured event streams. Two interesting future topic of research addressed by our group are (1) the dynamic configuration of distributed event analysis and (2) the combination of monitoring events with program output typically generated via file system calls, so that users can 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 used for steering a large-scale atmospheric modeling application[26].

4 System Evaluation

End users will not employ Falcon for program monitoring and steering if its use results in undue degradation in the performance of their application programs. Moreover, since Falcon targets parallel user applications, Falcon must deliver acceptable program perturbation and high monitoring performance across a range of parallel machine sizes and within the required monitoring latencies and bandwidths. This section demonstrates Falcon's ability to offer such scalable and predictable performance on large-scale shared memory multiprocessors (SMPs). As explained in Section 3.3, this demonstration in part exploits the parallelism available on the underlying SMP machine. It does not rely on any architecture-specific attributes of the machine chosen for performance evaluation.

In this section we first evaluate the basic performance of Falcon's monitoring mechanisms, including mea-

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.

surements of the average costs of tracing sensors and of alternative collection mechanisms, and measurements of minimum monitoring latencies. Second, we use several configurations of the MD application to impose different workloads on local monitors so that we can evaluate Falcon's ability to control monitoring overheads and to scale to different performance requirements. Increased resource allocations to local monitors are shown to enable a wide range of monitoring bandwidths, while retaining almost constant monitoring latencies. Third, we demonstrate the utility of simultaneously and dynamically employing a mix of mechanisms for information capture, called 'selective monitoring', resulting in the attainment of good performance for events produced at very high rates (*i.e.*, in the inner loop of an HPC application). Finally, we demonstrate the overheads of program steering in order to determine the latencies at which program steering may be performed with the current implementation of the Falcon system.

All measurements reported in this section are performed on a 64-node KSR shared memory supercomputer (SMP). Like most other currently available SMPs, this machine's cache-only architecture provides consistent shared memory across all processor nodes. The machine differs from SMPs like the Silicon Graphics PowerChallenge machines in the scalable nature of its bus structure, which consists of hierarchically interconnected rings, each of which can support up to 32 nodes. Compared to SGI machines, the 64-bit KSR processor nodes are somewhat slow, with a CPU clock speed of 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. Each node has 32 MBytes of main memory used as a local cache, a higher performance 0.5 Mbyte sub-cache, and a ring interface. The KSR machine's OSF Unix operating system implements the POSIX Pthreads standard parallel programming model. The measurements reported in this section take advantage of the operating system's ability to bind threads to processors, thereby enabling us to capture the effects of monitoring more precisely than on other target machines. However, the Falcon system itself does not rely on Pthreads availability on the target machine, since it is constructed with a user-level Cthreads package layered on top of Pthreads. On platforms not supporting Pthreads, Falcon employs multiple Unix processes sharing memory in place of the kernel-level Pthreads existing on the KSR machine.

4.1 Sensor Performance and Monitoring Latency

Some of the most basic measures of performance of a monitoring system are the program perturbation generating an event imposes and how quickly the monitoring system can transport events out of the application. This section characterizes these attributes of Falcon.

4.1.1 Sensor Performance

The basic cost of generating a monitoring sensor record on the KSR-2 is summarized in Table 1. These values represent direct program perturbation imposed by generating an event. The costs of generating an event are small enough to 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% total program 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. These percentages are derived from the cumulative sensor execution times while generating all of the sensor records in the MD program's critical execution path. For each such sensor, the dominant factor in its execution time is the cost of accessing the buffer shared between application and monitoring threads, namely, the cost of event transmission and buffering. This cost is determined by both the size of the sensor's event data structure and the cost of event transmission and buffering from sensors to local monitors. The use of multiple monitoring buffers (one per user thread) in Falcon reduces buffer access contention between user and monitoring threads, so that the effective cost of buffer access is dominated by the cost of copying a sensor record to the buffer. This latter cost depends on event size, as is clearly evident from the measurements in Table 1, which displays measured execution times on a 64-node KSR-2 supercomputer[5].

The measurements shown in Table 1 are the composite costs of executing a tracing sensor of a particular size. They include the costs of accessing the sensor switch flag, computing the values of sensor attributes, and writing the generated sensor record into an event queue. Falcon performs no additional inline processing, such algorithms for detecting more complex side effects of program perturbation[32], so these subcosts characterize the basic cost of tracing sensors. However, these values do not include excessive perturbation that might be caused by bottlenecks in the processing and transmission of the events (which would result in delays in obtaining buffer space). Such perturbation may be avoided by making dynamic adjustments to the monitoring system itself, such as turning off non-critical sensors, reducing a sensor's tracing rate, and forking new local monitoring threads.

From the measurements in Table 1 we conclude that any on-line monitoring system should offer users the ability to control program perturbation by customizing event formats, especially with respect to user-defined vs. implicit attributes (*e.g.*, timestamps, thread identifiers, etc.) carried by such events. The value of this

---

[5] The 64 node KSR-1 machine at Georgia Institute of Technology was upgraded to a 64 node KSR-2 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.

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

(a) Minimum monitoring latency

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

(b) Latency at high monitoring rates

Table 2: Latency in microseconds on the KSR-2.

conclusion is demonstrated further in Section 4.3, which makes mixed use of 'standard' vs. customized event formats by employing both tracing sensors and customized 'probe events' in monitoring the performance of the MD program's inner loop.

The relatively low program perturbation reported in Table 1 is experienced only when buffer access times are not distorted by lack of space or by access contention. These conditions are determined by a variety of monitoring system attributes, including the number of event queues and local monitor threads, and the actual event processing demands placed on local monitors. These performance effects are evaluated in the context of monitoring latency and perturbation in Section 4.1.2.

4.1.2 Monitoring Latency

For on-line monitoring, it is important to reduce both program perturbation and *monitoring latency*, which is 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 such as is required to support the configuration done in [39]. 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 and upon the size of the monitoring buffers.

Table 2 depicts the results of two experiments to measure monitoring latency with a synthetic workload generator instrumented to generate sensor records of size 32 bytes at varying rates and using a single local monitoring thread. In Table 2a, monitoring latency is evaluated under low loads, resulting in an approximate lower bound on latency. The resulting latency varies with event record sizes, but not with buffer size, demonstrating the independence of monitoring latency on the size of the monitoring buffers at low loads. Table 2b, however, uses a much higher monitoring load[6] and shows that larger monitoring buffers can lead to increased event latency.

While we have not yet fully explored the full range of possible conditions with experiments, it is intuitively clear that there are some tradeoffs in determining the size of monitoring buffers. Buffers are principally valuable in that they allow event 'bursts' to occur without excessive program perturbation. Events rates within these bursts may exceed the saturation rate of the rest of the monitoring system, but as long as the buffer is large enough to contain the burst and the average event rate is below the saturation point of the rest of the monitoring system, no excessive perturbation will occur. Conversely, monitoring buffers that are too small may increase direct perturbation to the program because buffers will fill during bursty activity. However, large monitoring buffers may also directly increase monitoring event latency. In particular, it is clear that under the extreme circumstance of monitoring system saturation, the application speed will be limited by the rate at which the monitoring system can handle events. Then perturbation of the program is extreme and monitoring buffers will always be full. In this circumstance, the size of the monitoring buffers determines how far "ahead" of the monitoring the program gets. Since the events spend much of their time stored in the buffers between production and analysis, buffer size translates directly into program latency and a smaller buffer size would be desirable to minimize latency. In Falcon, the ability to configure the size of monitoring buffers can be useful for applications with strict monitoring requirements. If a fixed latency is required, perhaps for steering, small monitoring buffers can be used to guard against monitoring system saturation. Conversely, large monitoring buffers will promote minimal perturbation if that is imperative. Falcon's default configuration uses a monitoring buffer size of $2K$ as a compromise that provides reasonable protection against program perturbation without causing extreme latency if the system saturates.

Figure 8 shows how event latency increases with increasing average event rates in a sample application. An idealized graph of the relationship between event rate and latency would look more like a step function. A constant low latency value would prevail until the event rate exceeded the saturation rate for the monitoring system. Then the monitoring buffers would fill, latency would increase by an amount of time dependent on the size of the monitoring buffers and the event rate would be limited by the monitoring system rather than the speed at which the application proceeded. At the point, both the event rate and the latency are fixed by the rate at which the monitoring system can process them.

The fact that the measured latency in Figure 8 deviates from an idealized step function can probably be attributed implementation artifacts and to the fact that event rates in real applications tend to be dynamic

---

[6] The monitoring rates used in Table 2b were approximately 40,000 events per second.

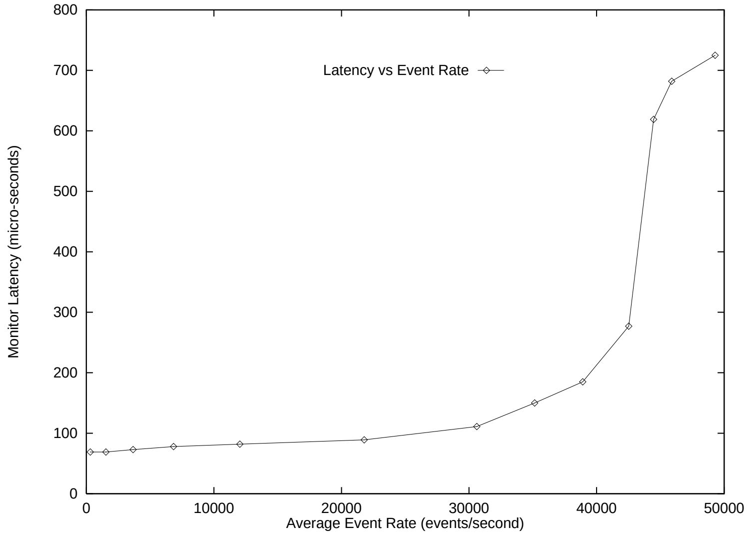

Figure 8: Monitoring latency versus average event rate on the KSR-2 when using one local monitor.

and bursty. This variation in rates causes occasional saturation and increases the average latency at overall event rates which average below the saturation value. The results in Figure 8 show that a single local monitor thread is often saturated when the overall event rate approaches 40,000 to 50,000 events per second.

If the application attempts to generate events at a rate higher than the monitoring system's saturation rate, event latency is maximized and the program is heavily perturbed by monitoring. However, Falcon has the ability to spawn multiple local monitoring threads to increase the trace event processing capability of the monitoring system. Figure 9 shows the relationship between monitoring latency and attempted event rates for Falcon configurations of 1, 2, 3 and 4 local monitors. Because actual event rates in MD are bursty might be limited by monitoring system saturation, the X-axis is calibrated by the number of processors used by the MD simulation. In this experiment, all procedure calls to the Cthreads library are traced. As MD uses more processors, the frequency of calls to the Cthreads library increases and results in higher event rates. It is evident from the results shown in Figure 9 that additional local monitors are effective in reducing monitoring latency at higher event rates. Configurations of Falcon with more local monitors have the same or lower latency for a given number of MD domains because they can handle higher event rates before saturating.

These results show the value of a configurable monitoring system. Falcon can be adapted to handle a range of application monitoring demands through the configuration of such things as buffer sizes and the number of local monitors. Such configuration could even be performed dynamically in a fashion similar to on-line program steering, where the saturation points for local monitors might be detected by monitoring event latency and used as triggers for configuring the monitoring system itself. Falcon is not yet capable of

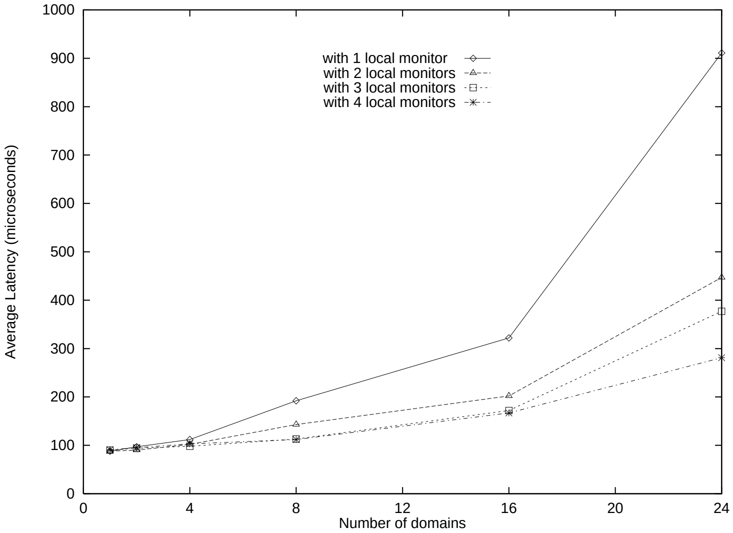

Figure 9: Monitoring latency with multiple local monitors on the KSR-2. Each domain of particles is assigned to one processor.

this type of self-configuration but it will be investigated in future work.

4.2 Monitoring Performance in a Real Application

While the figures in the previous section provide insights into Falcon's basic costs and performance, it is more revealing to explore Falcon's behavior in the context of understanding a real application. In order to this behavior, Falcon was used to monitor the MD application on a 64-node Kendall Square Research KSR1 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].

The experiments presented in this section demonstrate that the multiple monitoring mechanisms (*e.g.*, tracing versus sampling sensors) supported by Falcon can be employed such that monitoring overheads remain moderate for realistic parallel application programs. More precisely, several insights from the previous section are illustrated and demonstrated with the MD example:

- While it is not feasible to trace the inner loop of a high performance application like the MD program, acceptable program perturbation may be attained by judicious use of both tracing and sampling techniques using Falcon's sensors.
- A comparison with a common program profiling tool (*i.e.*., Unix GProf) demonstrates that it is important to conduct monitoring such that only the program attributes of interest to specific experiments are captured by the monitoring system.

- Parallelism in local monitoring can be important even for modestly sized parallel applications, such as the MD program running on 25 processors.

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

An overview of this section's measurements appears in Table 3, which depicts the results of four different sets of MD runs, normed against a run of MD without monitoring (Original MD). These experiments compare the performance and perturbation when using Falcon for four different cases: (1) when tracing only MD calls to the underlying Cthreads package (Dft Mon Only), (2) when tracing Cthreads events as well as sampling (using sampling sensors) the 10 most frequently called procedures in MD (Dft Mon & Sampling), (3) when using the Unix Gprof profiler existing on the KSR-1 machine (MD with Gprof), and (4) 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 the costs of either forwarding trace events to a front end workstation or storing them in trace data files, since those costs are not dependent on Falcon's design decisions but rather on the performance of the networking and/or file system implementation on the KSR. Specifically, the 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.

Performance with different amounts of monitoring. The summarized results appearing in Table 3 are presented with respect to program execution times and speedups in Figures 10 and 11. Figure 10 depicts the MD application's execution time with different amounts of monitoring or profiling, whereas the resulting program perturbation due to monitoring is shown in terms of speedup degradation in Figure 11.

Specifically, one experiment (Dft Mon Only – default monitoring) measures the overhead of monitoring

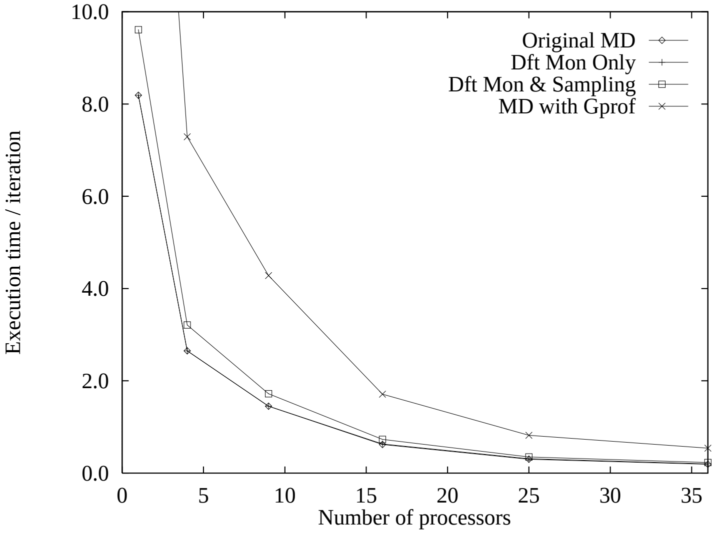

Figure 10: Comparing average execution time of each iteration of MD on the KSR-1 (**Original MD** and **Dft Mon Only** are very close to each other).

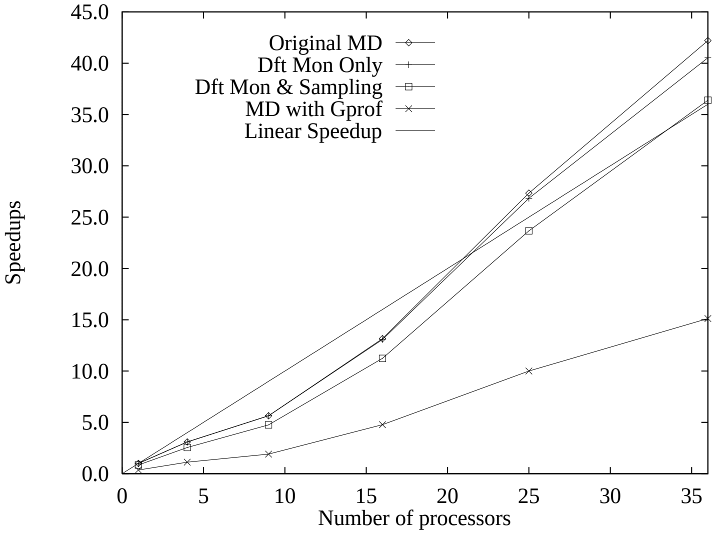

Figure 11: Comparing speedups of MD on the KSR-1[7].

when Falcon traces all calls to the underlying Cthreads package. 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 noticeably perturb the execution of MD. Interestingly, even the moderate amounts of tracing performed for default monitoring result in slight increases in monitoring overheads with an increasing number of processors. These increases are caused by increasing numbers of events (more user threads imply more cthreads calls, and hence more monitoring events) generated over a shorter time duration. These increases would eventually saturate the available local monitoring threads. This problem may be remedied by creation of additional local monitoring threads.

Falcon overhead versus other tools. In comparison to tools like Falcon, existing program profiling tools like Unix GProf do not offer adequate performance in program monitoring. The results described next are not surprising, since such profiling tools typically maintain large amounts of compiler-derived information about a parallel program's attributes, whereas Falcon maintains only the precise information required for the specific program measurements being made. Specifically, 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 this implementation with Falcon, we exclude the time spent 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% [16].

Performance of alternative monitoring techniques. While the gross program perturbation associated with the more event intensive activities in Table 3 clearly demonstrate the importance of monitoring only the program attributes of interest to the user, the table also shows that it is equally important to adjust or select the techniques being used for information capture. Specifically, in the column labeled Tracing all Mon Events, 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 increase 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

---

[7] Super-linear speedups are due to cache effects in the KSR'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. On fewer processors, slower memory must be utilized to hold all the data.

local monitoring thread being used in the experiment. In contrast, acceptable performance is attained when employing tracing sensors for default monitoring while using sampling sensors for tracing the inner loop of the MD program (see column Dft Mon & Sampling). These performance results clearly demonstrates two points. First, since tracing sensors are too expensive for procedure profiling, it is again apparent that 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 (also see Sections 4.3 and 3.5), 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.

Complexity of MD instrumentation. The instrumentation of MD performed in this section was straightforward, since it only involved using a small number of Falcon-generated sensors in a well-defined piece of MD code. Furthermore, since Falcon supports both sampling and tracing sensors, both types of monitoring are easily performed and/or interchanged. Similarly, default monitoring may be enabled and disabled by use of initialization time flags associated with the underlying Cthreads package. However, the current implementation of Falcon still requires the explicit instrumentation of applications with sensor code, so that the addition or removal of sensors requires code recompilation. This problem may be addressed by use of dynamic linking and by using the techniques for adding and removing instrumentation points into code presented by Miller et al.[21]. Moreover, while the results in this and the next section demonstrate the importance of using multiple monitoring mechanisms (*e.g.*, tracing and sampling) as well as dynamically controlling mechanism use, Falcon does not yet offer any runtime tool support for enabling or disabling individual sensors, for 'morphing' sensor implementations, or for stating higher level specifications of what to monitor and how to control such monitoring over time.

4.3 Application Specific Selective Monitoring

The previous section explored Falcon's performance in several common monitoring situations in the MD application. However, the role of monitoring in those situations was rather generic. Providing insight into thread-level events and procedure calls frequencies is important, but there are circumstances where such generic information does not go directly to the characteristics of the application under study. To test Falcon's ability to analyze and reveal dynamic behavior in an application's core computation we constructed an experiment to answer specific questions about MD's behavior.

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 cannot understand and evaluate code performance without both summary (*e.g.*, total number of invocations) and sequencing or dependency information (*e.g.*, 'b' was done after 'a' occurred). This sort of 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 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 (pseudo code 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
            for (each particle part_1 in molecule mol_1) do
                if (within_cutoff_radius(part_1, mol_2)) then
                    for (each particle part_2 in molecule mol_2) do
                        if (within_cutoff_radius(part_1, part_2)) then
                            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 is the smallest box that contains all of its particles. The minimum distance between two molecules is 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 in the 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 (a) distances between molecules' bounding boxes with (b) 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?

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

The performance of such dynamically selective monitoring is evaluated in terms of execution time of each MD iteration. The results are presented in Table 4. 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 (levels are numbered from zero to three starting at the outermost level). 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 versus 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 runs. One conclusion from these results is that Falcon's monitoring mechanisms themselves should be steered, so that runtime monitoring overheads and latencies are controlled throughout the program's execution. We have not yet developed algorithms that can perform such steering.

4.4 Performance of On-line Steering

Earlier parts of this section have explored various aspects of Falcon's monitoring system. The remainder will examine Falcon's steering component. As outlined in Section 3.4, the steering component of Falcon operates in conjunction with its monitoring components; the steering server makes steering decisions based on trace information collected and analyzed by the Falcon's on-line monitoring system. The purpose of this section is to understand the limitations on program steering defined by its current implementation. Specifically, we consider the basic performance of the steering library by measuring the *latency of steering*, which is the period of time from the occurrence of a program activity to when the event is noticed by the steering server. This latency constitutes the system's minimum response time, not taking into account the costs of steering algorithms.

Steering latency is comprised of the following elements, of which (1) and (2) have already been evaluated in Section 4.1.2: (1) an inserted sensor captures the program activity and writes a trace event to a monitoring buffer, (2) a local monitor picks up the event, processes it, and then forwards the event to the event queue connected to the steering server, (3) the steering server receives the event, looks up its event/action repository, and decides what steering actions to take, and (4) the steering server uses a steering probe or an actuator to change the application state or program parameters. We next present the evaluation of (3) and (4) using a synthetic program instrumented to generate monitoring events at a variable rate. Each event causes a simple steering action essentially changing the value of a memory location in the program via a steering probe. Applications threads, local monitors, and the steering server execute on different processors to make use of the parallelism available on the target KSR-2 machine. For these measurements, the event/action repository in the steering server only contains a moderate number of different steering event types and their respective actions. A total of 100,000 sensor events are generated to obtain the average steering latency.

At moderate event rates, the average latency for closed-loop steering using Falcon is 610 *micro*seconds, with a minimum latency of 224 *micro*seconds. This latency may be reduced further by inclusion of steering functionality directly in local monitors, thereby avoiding additional event transfers between steering and monitoring threads (recall that monitoring latency is approximately 70 *micro*seconds). This latency may be reduced further by performing steering actions within local monitor threads, at the cost of reducing monitoring performance for non-steering relevant event streams.

The measurements presented above imply that program steering can be performed rates approximating the execution times of the set of inner loops in programs like MD. 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 algorithmic program configuration[5] and interactive steering by application programmers. It is not possible to use Falcon's current mechanisms to perform steering of program abstractions accessed with very high frequencies, like the adaptable locks described in [39]. Such high-rate and low-latency steering must be performed by local monitors themselves, or even by using custom implementations of sampling sensors. Ongoing research with the steering component of Falcon includes its evaluation with a large scale atmospheric modeling application[26], as well as its integration with user interface facilities for program steering. A general overview of research on program steering appears in [18]. More information on the utility and challenges concerning steering appears in [7].

5 Related Research

Other research related to Falcon falls largely into three different categories: (1) work on program steering, (2) research addressing program and performance monitoring, and (3) specific results concerning program perturbation and other analyses of monitoring information implemented by monitoring systems. Each of these topics are reviewed in turn below.

Interactive program steering. The concept of steering can be found in many interactive scientific visualization and animation applications which allow users directly to manipulate the objects to be visualized or animated [24, 22]. 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 [24], 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 [22] 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, 39, 8]. Early work in this research area focuses on the dynamic adjustment of parallel and real-time applications in order to adapt them to different execution environments [44]. More recent experiments demonstrate that changes to specific program states or program components, such as locks [39] and problem partition boundaries [8], can significantly improve overall program performance.

Such previous work differs from the results presented in this paper in the extent and nature of its support for program steering. The Falcon system explores in depth the monitoring requirements necessary to support on-line program steering. Also, the steering mechanisms of Falcon are intended to support both algorithmic and interactive steering, which imposes performance requirements not shared by other systems. Readers interested in the general topic of program steering can find a complete review of such work in [18].

Program monitoring. Past work addressing the monitoring of parallel and distributed programs has typically focussed on performance understanding and debugging. These performance monitoring systems (*e.g.*Miller's IPS[36] and IPS-2[35], Reed's Pablo[42]) provide programmers with execution information about their parallel codes and lead their attention to those program components on which most execution time is spent. A variety of performance metrics like 'normalized processor time'[1] and 'execution time on the critical execution path' [35] are employed to describe the program's runtime performance. Recent research by several of these groups is attempting to relate performance measurements to the specific program components impacting performance, typically by interfacing performance tools to program compilers. The intent of such work is to overcome the difficulty of relating measured performance numbers to specific program details that might be changed or corrected. Such work is entirely complementary to the results presented in this paper and to the current implementation of Falcon, where it is assumed that programmers specify both the program attributes to be monitored and the steering actions to be taken in response to certain program behavior.

An alternative approach to performance debugging (and also complimentary to our research) is the W^3 search model described in [20]. This model is designed to assist users in interacting with the monitoring system while searching for performance bottlenecks in the target program. In Falcon's terminology, the W^3 search model implements a number of useful 'views' derived from lower level sensors included with the target code and provides guidance to programmers concerning the use of those views for performance debugging. W^3 could be implemented on top of Falcon's monitoring facilities if desired, but its current implementation relies on its own hooks inserted into target code. These hooks are implemented such that their presence in 'disabled' mode has no performance effects on the target program. The use of such techniques with Falcon would further improve system performance.

The topic of application-specific program monitoring has been addressed previously by Snodgrass and in our own research[45, 41]. In these systems, users can explicitly specify what variables or program states to monitor using specification languages [41, 25], some of which are based on the Entity-Relational model[45]. We are only now beginning to add 'view'-level specifications to the Falcon system.

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 [35]. In addition, more sophisticated analysis techniques may be used to reduce and correct perturbation to the measured program performance due to monitoring [32], and to apply various statistical filtering techniques prior displaying the data to users. All such techniques may be applied to Falcon's monitoring data as well, but at this time only the simplistic perturbation analysis required for the thread life-time view has been implemented with

Falcon. More interestingly, a PhD thesis addressing parallel program animation has used Falcon's threads performance data to evaluate the utility of a variety of animations for performance understanding [27, 28]. This thesis implemented on-line techniques for handling "out-of-order" events produced during program monitoring that may violate program causality.

6 Conclusions and Future Work

In this paper, we have demonstrated the utility and potentials of program steering with a large-scale parallel application program, a molecular dynamics simulation. Steering is performed using the Falcon monitoring and steering system, which permits 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 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.

The MD program and 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, including the IBM SP-2 machine. 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 [48]. Falcon's 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 and future extensions of Falcon not only address additional platforms (*e.g.*, the IBM SP-2 machine 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 [41, 45]. Such views will be implemented with library support resident in both local and central monitors. The resulting higher level abstractions presented by views to end users should be helpful in permitting them to both understand program behavior and to design suitable steering algorithms. 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 topic 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.

Longer term research with Falcon address 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 program steering and the dynamic configuration of the required monitoring, in parallel and distributed systems. 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[26], 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.

Acknowledgments. We thank Niru Mallavarupu for contributing to early implementations of Falcon components. Eileen Kraemer and John Stasko are responsible for some of the event reordering analysis and the detailed graphical displays of Cthread program performance.

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*, University of Wisconsin, Madison, WI, 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] Devesh Bhatt, Rakesh Jha, Todd Steves, Rashmi Bhatt, and David Wills. SPI: an Instrumentation Development Environment for Parallel/Distributed Systems. In *Proceedings of The 9th International Parallel Processing Symposium*, pages 494–501. IEEE, April 1995.

[5] T. Bihari and K. Schwan. Dynamic adaptation of real-time software. *ACM Transactions on Computer Systems*, 9(2):143–174, May 1991. Older version available from the Department of Computer and Information Science, The Ohio State University, OSU-CISRC-5/88-TR, newer version available from College of Computing, Georgia Institute of Technology, Atlanta GA, GTRC-TR-90/67.

[6] Charles Consel, Calton Pu, and Jonathan Walpole. Incremental specialization: The key to high performance, modularity and portability in operating systems. In *Proceedings of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM, June 1993.

[7] Greg Eisenhauer, Weiming Gu, Thomas Kindler, Karsten Schwan, Dilma Silva, and Jeffrey Vetter. Opportunities and tools for highly interactive distributed and parallel computing. Technical Report GIT-CC-94-58, Georgia Institute of Technology, College of Computing, Atlanta, GA 30332-0280, December 1994.

[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] Ahmed Gheith, Bodhi Mukherjee, Dilma Silva, and Karsten Schwan. KTK: Kernel support for configurable objects and invocations. In *Second International Workshop on Configurable Distributed Systems*, pages 236–240. IEEE, ACM, March 1994.

[11] Ahmed Gheith, Bodhisattwa Mukherjee, Dilma Silva, and Karsten Schwan. KTK: Configurable objects and invocations. In *Proceedings of the International Workshop on Object-Orientation in Operating Systems*, August 1993. (Position Paper).

[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] 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.

[14] Weiming Gu. *On-line Monitoring and Interactive Steering of Large-Scale Parallel and Distributed Applications.* PhD thesis, Georgia Institute of Technology, August 1995.

[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. Technical Report GIT-CC-94-21, Georgia Institute of Technology, College of Computing, Atlanta, GA 30332-0280, April 1994. Also in Frontiers 95.

[16] 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.

[17] Weiming Gu, Jeffrey Vetter, and Karsten Schwan. An annotated bibliography of interactive program steering. Technical Report GIT-CC-94-15, Georgia Institute of Technology, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280, February 1994. Also appear in ACM SIGPLAN Notices 29(9), pages 140–148.

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

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

[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] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. In *Proceedings of SHPCC'94*, pages 841–850, Knoxville, TN, May 1994. IEEE Computer Society Press.

[22] D. Jablonowski, J. Bruner, B. Bliss, and R. Haber. VASE: The Visualization and application steering environment. In *Proceedings of Sumpercomputing '93*, pages 560–569, Portland, OR, November 1993.

[23] Yves Jean, Thomas Kindler, William Ribarsky, Weiming Gu, Gregory Eisenhauer, Karsten Schwan, and Fred Alyea. Case study: An integrated approach for steering, visualization, and analysis of atmospheric simulations. In *Visualization '95.* IEEE, October 1995. Also published as GIT-GVU-95-15, http://www.cc.gatech.edu/gvu/reports).

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

[25] Carol Kilpatrick and Karsten Schwan. Using languages for describing capture, analysis, and display of performance information for parallel and distributed applications. Technical report, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, February 1990.

[26] Thomas Kindler, Karsten Schwan, Dilma Silva, Mary Trauner, and Fred Alyea. Parallelization of spectral models for atmospheric transport processes. Technical Report GIT-CC-95-17, Georgia Institute of Technology, College of Computing, Atlanta, GA 30332-0280, December 1994.

[27] Eileen Kraemer. *A Framework, Tools, and Methodology for the Visualization of Parallel and Distributed Systems.* PhD thesis, Georgia Institute of Technology, 1995.

[28] Eileen Kraemer and John T. Stasko. Issues in visualization for the comprehension of parallel progr ams. In *Proceedings, Workshop on Program Comprehension*, pages 116–125, Washington, DC, November 1994.

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

[30] 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.

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

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

[37] B. Mohr, D. Brown, and A. Malony. Tau: A portable parallel program analysis environment for pc++. In *Proceedings of CONPAR 94 - VAPP VI*, pages 29–40. University of Linz, September 1994. LNCS 854.

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

[39] Bodhisattwa Mukherjee and Karsten Schwan. Improving performance by use of adaptive objects: Experimentation with a configurable multiprocessor thread package. In *Proc. of the second International Symposium on High Performance Distributed Computing*, pages 59–66, July 1993.

[40] Bodhisattwa Mukherjee and Karsten Schwan. Survey of real-time operating systems. Technical Report GIT-CC-93/18, College of Computing, Georgia Institute of Technology, March 1993.

[41] 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.

[42] 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.

[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, 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.

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

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

[47] John T. Stasko and Eileen Kraemer. A methodology for building application-specific visualizations of parallel programs. Technical Report GIT-GVU-92-10, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, June 1992.

[48] 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.

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

Contents

List of Figures

List of Tables