Submitted to Concurrency: Practice and Experience

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

Weiming Gu

IBM Austin
11400 Burnet Road
Austin, TX 78758

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 enable 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 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 allows programs to be steered during their execution, by human users or algorithmically. This paper presents our basic research motivation, outlines the Falcon system's functionality, and includes 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 Introduction

Recent advances in high performance computing and communications permit large-scale applications to make simultaneous use of multiple networked parallel machines and workstations as their computational,

[1]This research was supported in part by NASA grant No. NAGW-3886, the NASA Graduate Student Researchers Program (No. NGT-51154), NSF equipment grants CDA-9501637, CDA-9422033, and CECS-9411846 and funding from Kendall Square Research Corporation.

graphical display, and input/output engines. Increased computational performance, more mature system software, and higher network bandwidth also allow 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, 8].

This paper describes and evaluates the Falcon system, emphasizing the system's contributions with respect to program monitoring. Briefly, Falcon is a set of tools that support three tasks. The first task is the on-line capture and collection 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. 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[41]) 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 specific contributions of Falcon to program monitoring are driven by its desire to support the online steering of programs for a wide range of high performance applications and target platforms. This implies the following three characteristics of Falcon. First, it supports *application-specific monitoring*, thereby enabling end users to capture and analyze the program information they need in order to understand program behavior and direct programs as required by online steering. With Falcon, such information ranges from data about single program variables to data about program states defined by complex expressions involving multiple program components. As a result, end users can view, analyze, and steer their applications in terms with which they are familiar (*e.g.*, 'time step size', 'current energy', etc.).

Second, Falcon supports *scalable monitoring*, offering 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.

Third, Falcon supports *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, 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.

Reiterating this paper's contributions, its focus is on monitoring high performance programs, addressing the topics of application-specific, scalable, and on-line information capture and analysis as required by the program steering task for which Falcon has been developed. The topic of program steering itself is explored in this paper to the extent of defining its requirements on the monitoring system and describing the its low-level mechanisms as 'duals' of the monitoring system's mechanisms. More detail on steering appears in [51] as extensions of Falcon's basic steering functionality. For brevity, this paper does not elaborate on two additional aspects of Falcon, which are (1) its 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

[18, 10]). Toward this end, Falcon uses the Polka system for program animation, which also provides users with easy-to-use tools for creating application-specific 2D animations of arbitrary program attributes[50]. Furthermore, Falcon is now being used for experimentation with alternative parameter settings in a large-scale atmospheric modeling application[28], using interactive 3D data displays via the Silicon Graphics Explorer and OpenInventor environment[25].

Falcon emphasizes low latency, on-line monitoring, achieved 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[43] and Miller[23], both of whom 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[34]. As low monitoring latency and low monitoring perturbation are conflicting attributes, Falcon offers several mechanisms to balance the tradeoff between latency and perturbation and this paper includes a study of the such tradeoffs. A further distinction between Falcon and other projects on performance debugging [40, 32, 3] is derived from 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).

This work differs from related research in program steering (*e.g.*, the Vase system[24]) in our focus on steering by human users. Because Falcon enables both algorithmic [44] and interactive [51, 18] program steering, it emphasizes monitoring latencies and overheads more strictly than human interactive systems like Vase. By offering low monitoring latencies, interesting program events may be recognized with suitable delays for corrective actions by adaptation algorithms or human users. 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 text, 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 to explore 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

The broader context of our work on Falcon is the 'Distributed Laboratories Project'[45]. This project is developing tools to enable physically distributed end users to interact with each other and with high-performance applications as they might in a traditional physical laboratory setting. The overall goal of the project is to enhance the end-users' insight into the application-level behavior of the high-performance program under study. The ability to extract application-level information from the running program, transmit it, process it, and ultimately display it to an end-user for interpretation is critical to achieving this goal. It is also important that these functions be performed efficiently, without undue latency, and with a minimum of disruption to normal application function and timing. This provides the fundamental motivation for our work on program monitoring.

Support for program steering is another important aspect of providing insight into application behavior. 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[28]. However, our work has often been more concerned with performance improvement by on-line program adaptation, including demonstrating significant performance gains through adaptive resource allocation in embedded applications [44] and through on-line configuration of mutex lock implementations in threads-based multiprocessor programs [41]. Similarly, object-based mechanisms for on-line program configuration are described in [5, 16, 47], 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[31, 15, 35].

Many of the research results listed above concern performance improvements attained by program steering. Because the utility and importance of program monitoring is relatively well accepted, and its necessity for program steering is obvious, we will limit our further discussions to the presentation of novel concepts, results, and evaluations concerning program monitoring. We do not aim to demonstrate simultaneously the general utility of program steering. Instead, by using Falcon to steer sample applications developed jointly with end users in physics[13] and atmospheric sciences[28], we identify the requirements steering imposes on the monitoring system being used. In order to provide a context for these discussions, we next describe the molecular dynamics simulation (MD) developed jointly with physicists in some detail.

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 [54, 13]. 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.

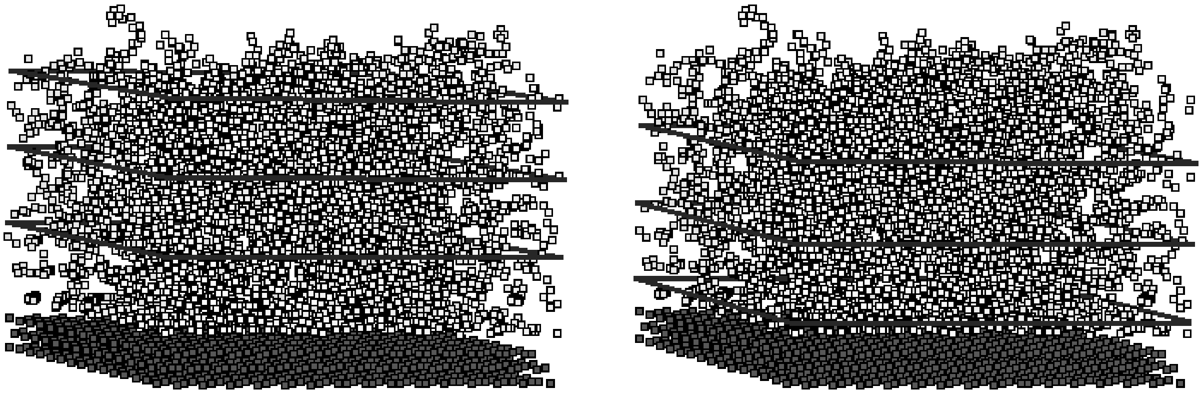
The implementation of the MD application attains parallelism by domain decomposition. Specifically, the simulation system is divided into regions and the responsibility for computing forces on the particles in each region is assigned to a specific processor. In the case of MD, we can assume that the 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.

Next we describe the opportunities for steering offered by the MD application. These descriptions are focussed on MD, but it is clear that the properties and behaviors of MD that enable such steering are shared by many scientific and engineering simulations, including the stencil codes developed by our group in conjunction with researchers at Los Alamos Labs [51], the atmospheric modeling application we are developing jointly with atmospheric scientists [28] as well as programs like the Ocean benchmark part of the Splash benchmark set [51].

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



(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.

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 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 so that end users are required to interact with the application only when automated steering is not successful.

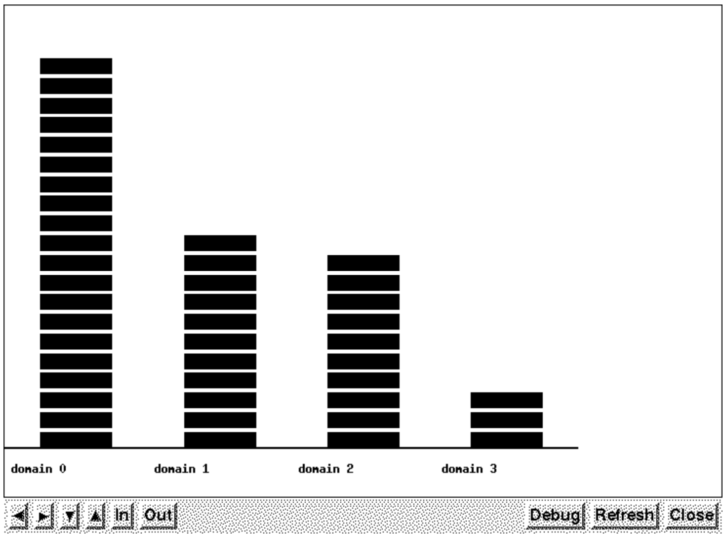


Figure 2: The load balance view of MD for its initial 4 processor configuration depicted in Figure 1. The vertical dimension depicts the a running average of the execution time of thread n executing domain n of the MD simulation.

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 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 due to the additional costs imposed by steering and monitoring on the parallel program's execution. Second, it shows 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. First, program steering requires that application builders write their code so that steering is possible. Second, users must provide the program and performance information necessary for making steering decisions. Third,

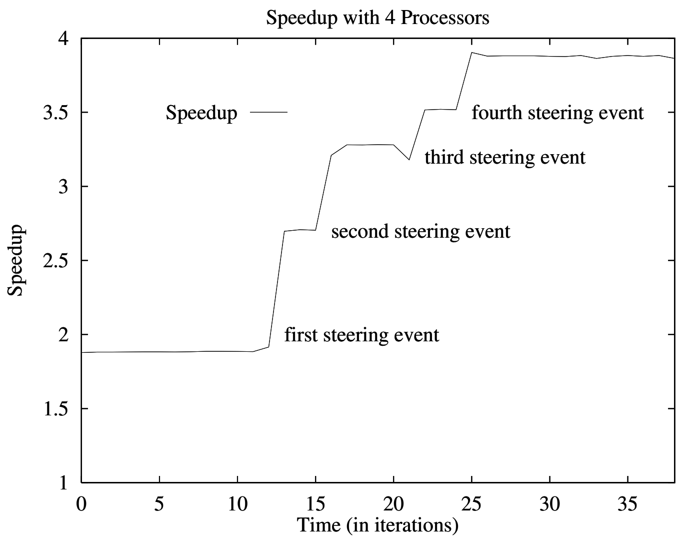


Figure 3: The effect of steering on performance over time with 4 processors.

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 assuming substantial compiler support[46], or by requiring that the programming language or system offer stronger mechanisms of abstraction than those existing in parallel Fortran or in the Cthreads library used in our work (*e.g.*, the object model[5, 47, 31, 16]).

It is interesting to note that such abstraction mechanisms are now being developed for high performance applications, including the object-based parallel programming languages described in [7, 6] and object-based programming libraries that target specific application domains, such as the VSIP library for signal processing applications [20] and the POET frameworks for generalized particle applications [2]. In the case of the POET frameworks, for instance, this programming library offers abstractions with which alternative decomposition geometries and boundaries are easily defined for MD simulations. Such abstractions may themselves be instrumented for steering, thereby permitting end users to construct steerable applications without paying attention to how such steering might be performed. Similar arguments may be made for numerical libraries employed with high performance codes: by making libraries steerable using instrumentation like Falcon's, end users are provided with steerable application at little additional cost in programming complexity.

The requirements that program steering imposes on monitoring have directed our research in program monitoring. Accordingly, this paper's primary concerns 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 [19, 52]), each requiring 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[41]. 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 [41], 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 next section discusses how the Falcon system provides 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 and Contributions

Three attributes of Falcon are designed to address the on-line monitoring requirements of program steering:

1. Falcon supports the *application-specific monitoring/steering, analysis, and display* of program information, so that users can capture, process, understand and steer the program attributes relevant to their steering tasks, be it a dynamic program modification or a specific performance problem being diagnosed or investigated.
2. A monitoring system cannot anticipate the steering actions or algorithms with which it will be employed, or the rates at which steering and therefore, monitoring will be performed. Therefore, the system's role should be to maximize the variety of steering and associated monitoring actions it can support. Falcon attains this goal by providing users with the ability to *reduce or at least control monitoring latency* throughout the execution of a parallel program, and to maintain acceptable workloads due to monitoring imposed on the underlying parallel machine. The realization of this goal relies on providing end users with runtime control concerning the monitoring mechanisms and the monitoring system configuration being employed.
3. Falcon's support of monitoring is *scalable*, in terms of machine size and program needs. Such scalability is attained by dynamic variation of the resources consumed by its runtime system. 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 large SMP multiprocessor.

3.2 System Overview

Falcon is constructed as a toolkit containing 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.

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. During program execution, program and performance information of

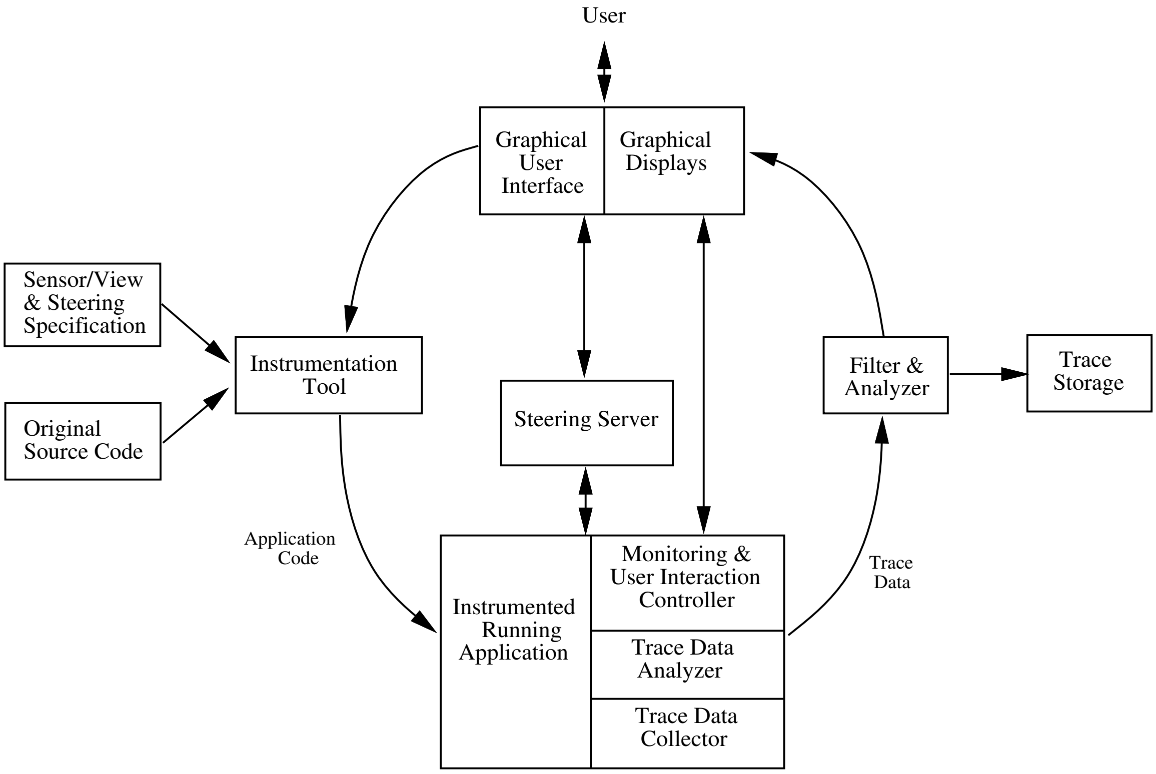


Figure 4: Overall architecture of Falcon.

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 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 monitor 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[50]. 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 and OpenInventor tools.

3.3 Implementation of On-line Monitoring

This section explores selected implementation attributes of Falcon to explain how Falcon attains its goals of application-specific monitoring, controlled monitoring overheads, and monitoring scalability, to delimit the utility of Falcon in terms of its offered functionality and its associated performance characteristics, and 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 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 machines, and various Linux platforms. Falcon's implementation structure is depicted in Figure 5. This implementation is discussed next in

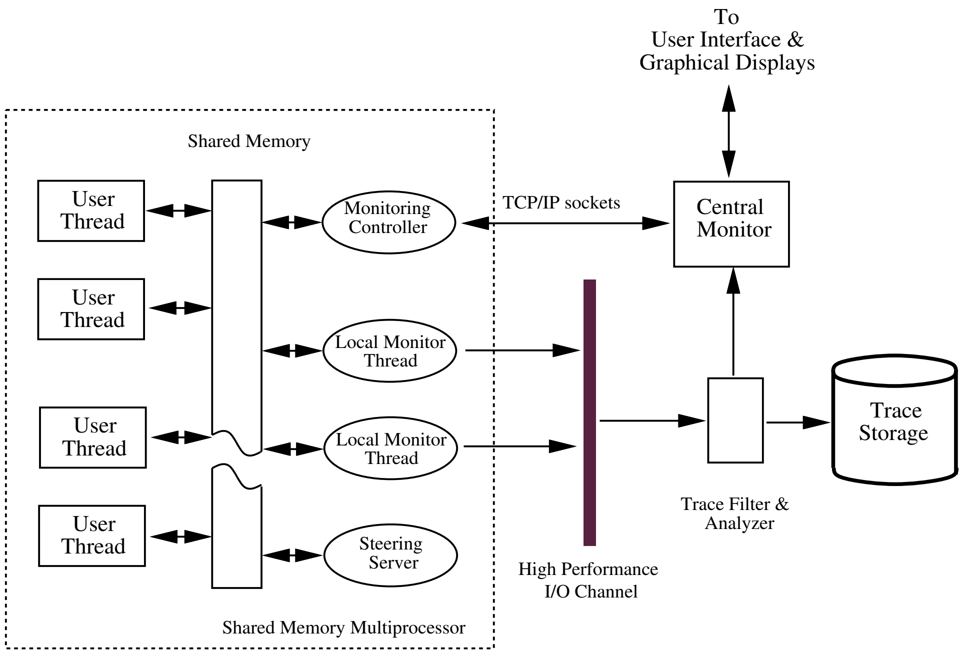


Figure 5: Implementation of the monitoring mechanism with Cthreads.

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 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. A simple compiler generates from this declaration 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.

Stub insertion may not be a trivial task. Specifically, it is important to place stubs only into those locations in the target code where they are needed to capture the relevant data in some consistent state. Furthermore, it is desirable to utilize alternative stub implementations[42], perhaps generated automatically by the stub compiler, and to delay the binding of these implementations to stub locations until the stub's usage for steering is known. This also implies the need for dynamic rebinding of stub implementations to locations, as steering needs may change over the course of a user's experimentation with the target program [51].

Controlling monitoring overheads – sensor types and sensor control. The monitoring overheads experienced when extracting program information may be controlled by using 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. Another sensor variant is described in [51]. More interestingly, however, it is also possible to shift workload from the target application to local monitors by employing *probes* code executed by local monitoring threads. Such probes may be used to inspect program variables asynchronously to the program's execution and without requiring prior instrumentation of the program's code.

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 off if events captured by those sensors are not currently needed by the end user or the steering algorithm.[2] 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, 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 analysis, 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 are presented in Section 4), and its effects on the correctness of timing information can be eliminated using known techniques for perturbation analysis[34].

Falcon's runtime monitoring system itself may be configured, including disabling or enabling sets of sensors, varying sensor activation to improve execution rates, etc. The utility of one such on-line variation is demonstrated later in this paper, where we change the number of local monitoring threads and communication buffers to configure the system for parallel programs and machines of different sizes. In general, dynamic configuration of the monitoring system is critical for matching monitoring performance to specific monitoring and steering tasks, to adapt the monitoring system to dynamic changes in workload imposed by the target application, and to deal with variations in steering behavior when users dynamically explore application behavior. Such dynamic configuration may also be automated. For example, when heavy monitoring is 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

[2] Related work by Hollingsworth and Miller [23] removes instrumentation points in order to completely eliminate the overheads of such 'turned-off' instrumentation points.

environment. It can be sent to Falcon's central monitor for further analysis and for display to end users, and it can also be stored in trace data files for postmortem analysis. The central monitor, user interface, and graphical displays may reside on different machines, reducing interference from monitoring activities to the target application's execution, and capitalizing 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 decide the location at which analysis actions are to be performed. 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). Some measurements demonstrating the utility of these ideas appear in [51].

Portability and limitations. The current implementation of Falcon relies on the availability of Georgia Tech Cthreads on the desired target machine and assumes that shared memory is available between application processes and local monitoring threads[3]. 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. 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, SGI, and SUN multiprocessors) offering the ability to execute different processes or threads on different processors. 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. The most recent release of the Solaris operating system for Sparcstations offers this capability as well, but on target machines not offering such user-level resource control, end users may affect monitoring system performance only by exploiting Falcon's alternative mechanisms for capture and analysis, such as sampling vs. tracing sensors, extended sensors, and probes.

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, relinking, or binary editing techniques like those described for the Synthesis operating system [36, 8] or for the Paradyn monitoring platform [37].

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 and explicit resource control on 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 stand-alone systems or as nodes in larger scale parallel machines.

3.4 Implementation of Program Steering

The implementation of Falcon's program steering mechanisms is a natural extension of the monitoring system's functionality for two reasons. First, the application-specific nature of Falcon's monitoring support permits users to perform monitoring for program attributes of their choice, which is also an essential characteristic of most of the steering tasks we have investigated and undertaken with end users. Second, local monitoring threads and shared memory between monitors and application components provide a suitable basis for implementation of a wide variety of steering functionality.

[3] A process-based implementation of Falcon is available, but offers reduced functionality with respect to on-line configuration, probes, and sensor types.

Asynchronous and synchronous steering with actuators, steering servers, and clients. Figure 6 depicts some internal features of the steering system as well as its relationship with the monitoring components of Falcon. A *steering server* on the target machine performs steering, and a *steering client* provides remote user interface and control facilities. Steering servers are like local monitors because they typically operate as threads in the application's address space, thereby gaining direct access to application components and the ability to execute asynchronously with application threads. A steering server performs two major

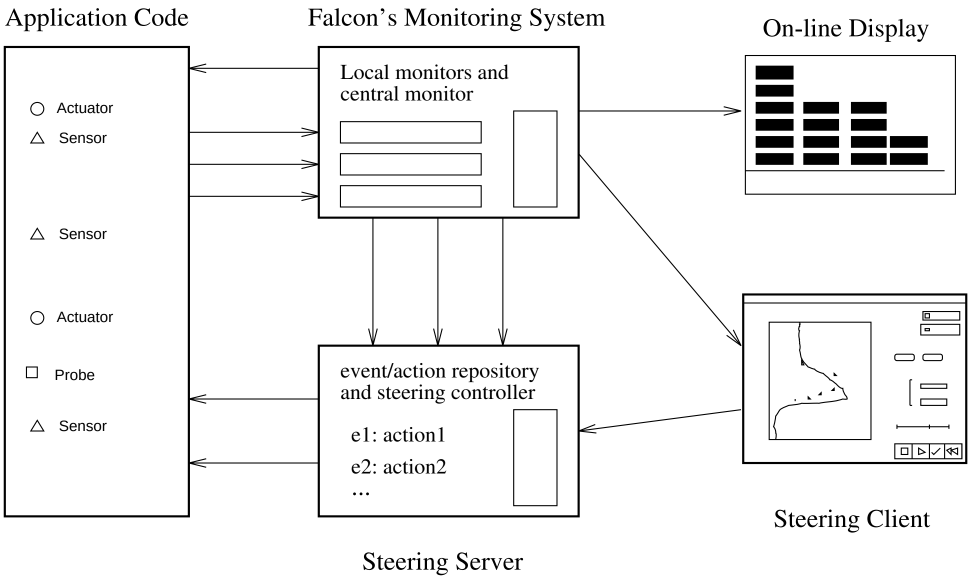


Figure 6: Overall structure of the steering system.

tasks: (1) reading incoming monitoring events from local monitors, and (2) responding to these events with appropriate steering actions. Concerning (1), local monitors perform information filtering and forward only the events of interest to current steering activities, thereby reducing the volume of data a steering server must handle. Toward this end, each steering server shares with each local monitor a circular buffer located in jointly accessible memory.

Human users or steering algorithms make steering decisions based on program state gathered with the monitoring system; these steering decisions are communicated to the steering server by the steering clients. The second task of the steering server is to respond to monitoring events with appropriate steering actions. These responses are based on previously encoded decision routines and actions that are stored in a repository within the server. This repository contains entries for each type of steering event, specifying the appropriate action for that event. Possible actions are quite diverse. An action may enact steering on some application component; it may accumulate monitoring events for future use in monitoring analysis or for some steering action; it may probe the target program for additional information; or it may simply forward the information to the steering client for display or further processing.

A steering client typically runs as a separate program on a remote machine. It provides an interface for end-users to interact with the application under investigation. For example, the steering client may be used to initiate a particular steering action, display and update the contents of the steering event repository, and input steering event/action pairs directly from end users to the steering server.

Diverse tasks require diverse monitoring and steering mechanisms. Two principal abstractions support Falcon's steering mechanisms. First, the steering system views the application as a collection of *program attributes*, which are modifiable program variables or characteristics. As with an extended sensor, each attribute has an associated procedure that operates on it. Second, by associating any number of such attributes with a program abstraction, developers create a *steerable entity*. This entity appears to the steering system as one that exports a number of invocable methods, each of which concerns a specific modifiable program attribute. Therefore, such abstractions have some resemblance to the adaptable objects first developed for real-time systems[5] and to configurable objects developed for object-oriented operating systems[15] or applications[47]. Each steerable entity is 'registered' with the steering system that maintains a repository of all such entities and their methods in the steering server[53].

As with Falcon's monitoring mechanisms, the specific details of steering can also be varied to match application requirements. Specifically, we offer both synchronous and asynchronous modes for invoking the methods of steering entities. An asynchronous 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 without requiring the program to have reached a certain execution state. A synchronous 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 by developers at locations deemed 'safe' for steering actions[51]. 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 the steering server enables actuators asynchronously with the application's execution, they are executed synchronously by the application thread. This permits the steering system to delegate the responsibility for determining when certain steering actions may be enacted to the application developer. This method contrasts with previous work performed for real-time systems [5, 15] 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, this method differs from those models in that the information on which steering decisions are made may include any number of current or accumulated program characteristics that are inspected by the steering system or even user knowledge about desired application behavior.

In summary, Falcon's steering facilities naturally extend its basic monitoring support; as with local monitors, a steering server is an additional thread spawned at the time of program initialization; it interacts with local monitors to gain access to application state and to cooperate with the application for steering. Steering probes enable the steering server to directly perform simple program changes, whereas more complex changes are enacted using actuators because they are executed by the application thread. 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[24]. Instead, we have developed a simple, interpreted language with which sequences of steering actions may be expressed, resulting in the runtime interpreter's ability to perform optimizations across such sequences [51].

The performance evaluation of steering in Section 4.4 demonstrates that the library's current implementation easily supports interactive steering. In addition, we identify the limits on possible rates with which steering may be performed due to the library's minimum monitoring, decision making, and enactment delays. An important result of our work is the insight that as with monitoring, the mechanisms for steering must match the tasks for which they are being employed, in perturbation, in overheads experienced for certain steering rates, and in latencies of steering. Moreover, for highly interactive high performance programs, such matching should be performed at runtime, while the end user is interactively exploring application characteristics, experimenting with alternative steering methods and approaches, and/or evaluating diverse approaches to improving application performance. This topic is addressed in more detail in [51].

3.5 Falcon's On-line Display System

Graphical displays have been shown useful in presenting data structures, algorithms[49], runtime program behavior[33], and performance information[21, 43] 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 permits 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 [28]. This view appears on the bottom right of Figure 6.

While there are many interesting research issues in the related to useful displays of program information, this paper will limit itself to a brief discussion of the specific requirements of on-line displays. 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[18, 17]. General methods for supporting performance understanding in program animation systems are described in [29, 50].

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.

The on-line and distributed analysis of captured trace data is a 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 preserves 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[43] 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.

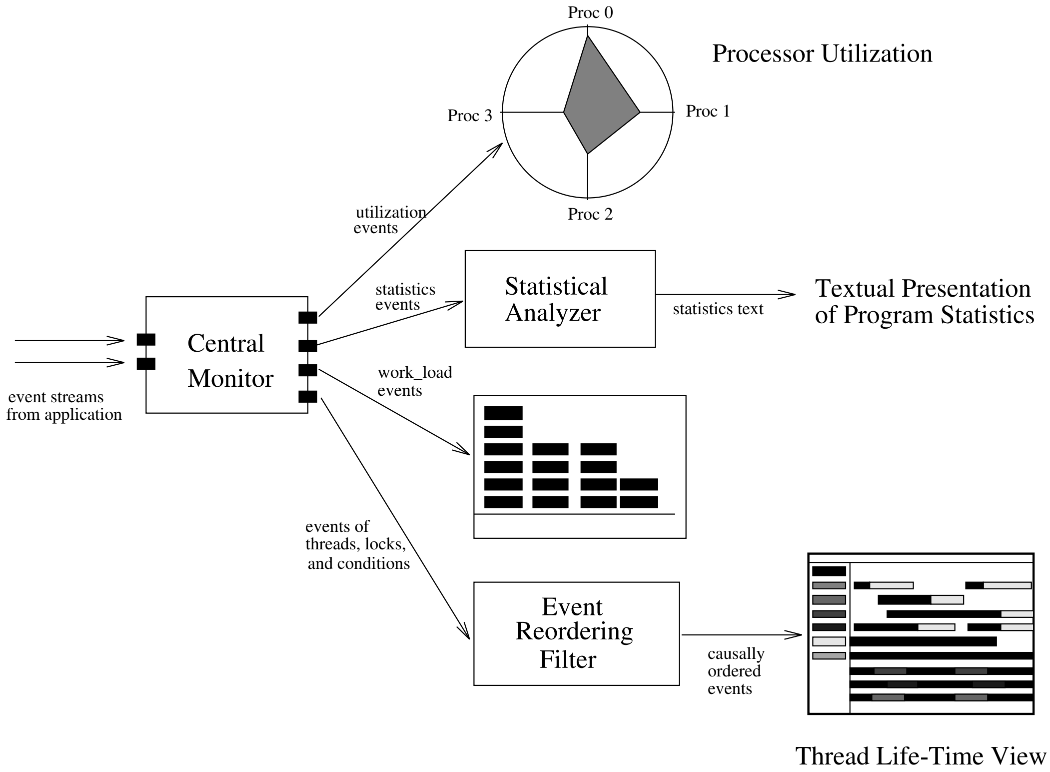


Figure 7: A sample on-line display system.

Its implementation and use for performance understanding are discussed in detail elsewhere[29, 50, 18].

To summarize, Falcon supports the on-line display of captured program information both by permitting the placement of necessary analysis code at any level of its event capture hierarchy and by facilitating the attachment of multiple performance-relevant and application-specific displays to captured event streams. Two interesting future topics of research addressed by our group are the dynamic configuration of distributed event analysis and 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[28].

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 applications, Falcon must deliver acceptable program perturbation and high monitoring performance across a range of parallel machine sizes and within the monitoring latencies and bandwidths desired by end users. 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. Although somewhat dated multiprocessors – KSR SMP machines[4] – are employed, the performance measurements discussed in this section do not rely on any architecture-

[4]KSR machines have not been in production since 1995.

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.

specific attributes of these machines. In measurements of similar mechanisms on current architectures like the SGI Powerchallenge and the DEC Alpha, we have observed similar relationships between the basic overheads of different monitoring and steering mechanisms[51], the tradeoffs in using extended sensors, local and remote monitors for capture and analysis, etc.

To evaluate the basic performance of Falcon's monitoring mechanisms, we first measure the average costs of tracing sensors, alternative collection mechanisms, and of minimum monitoring latencies. These measurements serve as background for the evaluation of Falcon's ability to control monitoring overheads and to scale to different performance requirements. This ability is then evaluated by using several configurations of the MD application to impose different workloads on local monitors. 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 by the current implementation of the Falcon system.

All measurements reported in this section were 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 directly rely on Pthreads availability on the target machine because 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 which generating an event imposes and the latency with which the monitoring system can transport events out of the application. This section characterizes these attributes of Falcon. The importance of these measurements is not derived from the absolute values being observed. Instead, they establish some basic properties of Falcon's implementation; Specifically the linear dependence of monitoring perturbation on the amounts of application data being captured, which prompt us to offer diverse event formats for sensors, and the strong effects on event latency of buffer sizes in high load conditions. These effects prompted us to conduct the additional experiments described in Section 4.1.2.

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. For example, 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. The total cost shown includes 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[34], 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 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 [41]. 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

[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.

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 using a single local monitoring thread. In Table 2(a), 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. Event 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" the program's execution is permitted to advance in reference to the monitor's knowledge of this execution. In this case, events will remain in buffers for some time between their generation and their consumption by the monitoring system's analysis routines. Therefore, smaller buffers imply smaller differences in the time of event generation and consumption. If the application blocks when its sensors cannot deposit their events, then Falcon's ability to configure buffer sizes can used to control differences in a rates of progress of the program and the monitoring system. Similarly, the latency of transport of event from application to monitoring analysis is also directly proportional to buffer size. Conversely, large monitoring buffers typically result in reduced program perturbation (*i.e.*, low likelihood of blocking due to full buffers) as well as larger event transport latencies. Falcon's default configuration uses a monitoring buffer size of $2K$ as a compromise that provides reasonable protection against program perturbation without causing excessively high event latencies in high load situations.

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

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

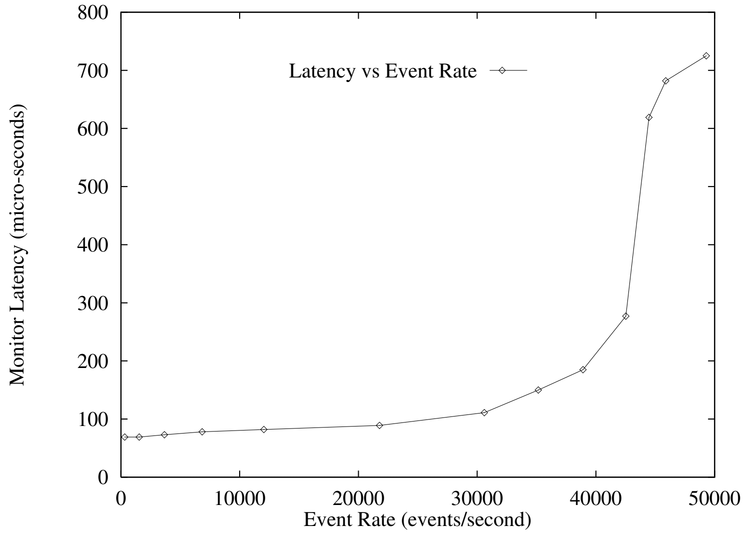


Figure 8: Monitoring latency versus average event rate on the KSR-2 when using one local monitor.

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 that 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 to implementation artifacts and to the fact that event rates in real applications tend to be dynamic 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. The number of processors used by the MD simulation is equal to the number of domains shown at X-axis. 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 attributes like 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 increases in monitoring event latency and used as triggers for configuring the monitoring system itself. Falcon is not yet capable of this type of self-configuration but this will be investigated in future work.

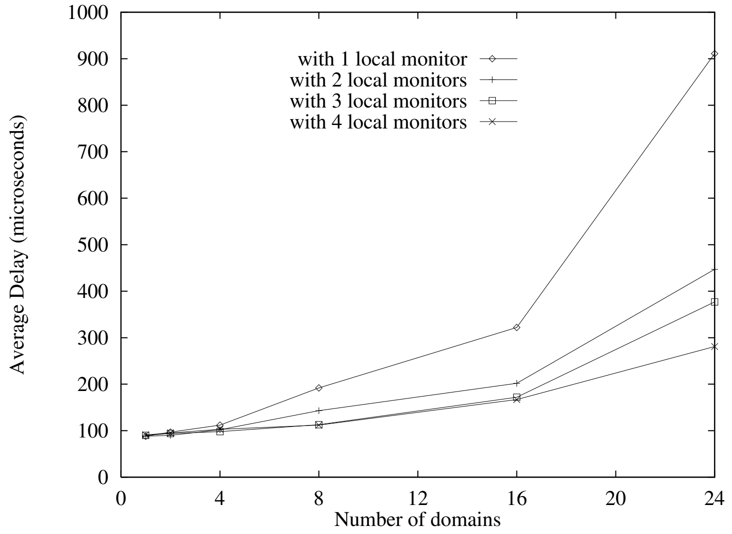


Figure 9: Monitoring latency with multiple local monitors on the KSR-2. Each domain of particles is assigned to one processor.

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 understand this behavior, we used Falcon to monitor the MD application on the 64-node KSR1 machine. The specific MD simulation used in these measurements employs a cylindrical domain decomposition; MD performance and speedups with different decompositions are evaluated in detail elsewhere[13].

The experiments presented in this section demonstrate that the multiple monitoring mechanisms (*e.g.*, tracing and sampling sensors) supported by Falcon can be employed to ensure 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 limit monitoring to those program attributes of interest to specific experiments.
- Parallelism in local monitoring can be important even for modestly sized parallel applications, such as the MD program running on 25 processors.

Table 3 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) tracing only MD calls to the underlying Cthreads package (Dft Mon Only), (2) tracing Cthreads events as well as sampling (using sampling sensors) the 10 most frequently called procedures in MD (Dft Mon & Sampling), (3) using the Unix Gprof profiler existing on the KSR-1 machine (MD with Gprof), and (4) 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

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.

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

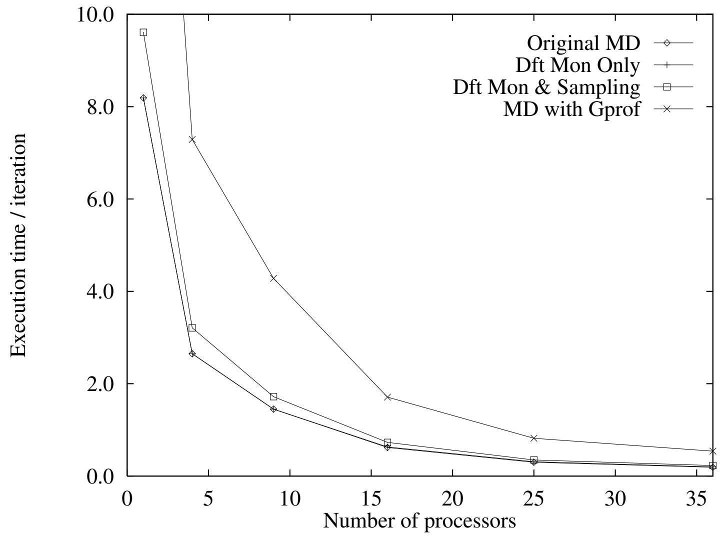


Figure 10: Comparing average execution time of each iteration of MD on the KSR-1 (Original MD and Dft Mon Only are very close to each other).

when Falcon traces all calls to the underlying Cthreads package. The monitoring information being

[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.

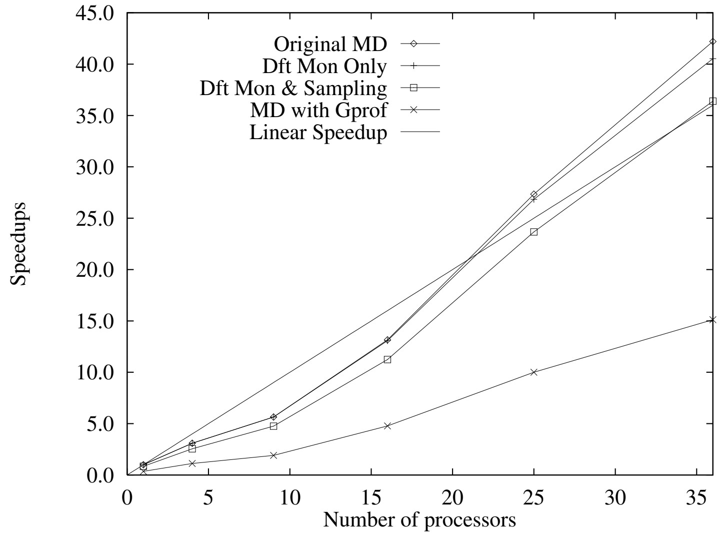


Figure 11: Comparing speedups of MD on the KSR-1[7].

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. The KSR implementation of Gprof used in these measurements was 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 over other profiling tools are demonstrated with Prof. Experimental results not reported in detail here show that Prof's overhead is approximately 130% [18].

Performance of alternative monitoring techniques. The large program perturbation associated with the more event-intensive activities in Table 3 clearly demonstrates 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

On fewer processors, slower memory must be utilized to hold all the data.

by monitoring tens of millions of procedures calls and are exacerbated by the saturation of the single local monitoring thread being used in the experiment. 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 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 Section4.3.

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 the application 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[23]. 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. While providing insights into thread-level events and procedure calls frequencies is important, there are circumstances where such generic information is not sufficient for understanding characteristics of applications 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 (pseudocode is shown below):

```
for (each molecule mol_1 in my domain) do
    for (each molecule mol_2 in domains within cut_off_radius) do
        if (within_cutoff_radius(mol_1, mol_2)) then
            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's 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 distances between molecules' bounding boxes to 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?

	Execution Time of each MD time step (seconds) & Monitoring Overhead					
No. of domains	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. But 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 explore 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 performing steering actions within local monitor threads (recall that monitoring latency is approximately 70 *micro*seconds), at the cost of reducing monitoring performance for non-steering relevant event streams. More important than the actual values of steering latencies presented in this paragraph are the following insights derived from them: (1) Falcon permits steering at rates similar to the execution rates of mid-level (not inner) loops of high performance programs like MD, and (2) variations in steering mechanisms and in the 'location' of steering (directly in the application's code, in local monitors, or in steering servers) are necessary to enable steering tasks ranging from high-rate algorithm-based performance steering to user-directed interactive steering. These insights are also validated in recent work by our group with the faster DEC Alpha processors and memory result in ratios of possible steering latencies to loop execution times that are similar to those experienced with the KSR multiprocessor.

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 [41]. Such high-rate and low-latency steering must be performed by local monitors themselves or by custom implementations of sampling sensors. Ongoing research with the steering component of Falcon includes its evaluation with a large scale atmospheric modeling application[28] as well as its integration with user interface facilities for program steering.

5 Related Research

The research related to Falcon may be classified as follows: (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 [26, 24]. 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 [26], 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 [24] offers tools that create and manage collections of steerable Fortran codes.

Our additional work on Magellan[51] uses a domain-specific language to describe steering requests to the monitoring and steering infrastructure. These language descriptions allow the steering system to optimize the requests based on steering frequency, latency and application perturbation.

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, 41, 11]. 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 [46]. More recent experiments demonstrate that changes to specific program states or program components, such as locks [41] and problem partition boundaries [11], can significantly improve overall program performance.

Previous works differ from the results presented in this paper in the extent and nature of their 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 [19]. A more comprehensive overview of research on program steering appears in [19, 52]. More information on the utility and challenges concerning steering appears in [9].

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[39] and IPS-2[38], Reed's Pablo[43]) 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' [38] 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 [22]. 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 by Falcon would further improve system performance.

The topic of application-specific program monitoring has been addressed previously by Snodgrass and in our own research[48, 42]. In these systems, users can explicitly specify what variables or program states to monitor using specification languages [42, 27], some of which are based on the Entity-Relational model[48]. 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 [38]. In addition, more sophisticated analysis techniques may be used to reduce and correct perturbation to the measured program performance due to monitoring [34], 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 within Falcon. Additionally, 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 [29, 30]. 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

This paper has explored the requirements steering imposes on program monitoring and explore the utility and potentials of program steering with a large-scale parallel application program, a molecular dynamics simulation. The paper's results focus on the on-line monitoring task. Specifically, the Falcon system permits programmers to capture and view the program attributes of interest to them. Such application-specific monitoring may be performed on-line (during the program's execution), with dynamically controlled monitoring overheads and latencies, thereby supporting a wide variety of steering tasks. To attain such controls, Falcon's monitoring mechanisms themselves may be configured on-line to realize suitable tradeoffs in monitoring latency, overhead, and perturbation.

The MD program and the Falcon system are implemented and evaluated on a shared memory multiprocessor that permits the detailed evaluation of the overheads of alternative implementations of monitoring and steering, a 64-node KSR shared memory supercomputer. The basic insights derived from these evaluations have been applied to Falcon's current execution platforms, as well, including SGI and SUN SPARC parallel workstations. A version of Falcon for cluster computers and for heterogeneous sets of networked machines is available, as well, and has been used on machines like the IBM SP-2, clusters of SUN Sparcstations and SGI Origin machines, the Linux- and Solaris-based x86-based PC clusters. These distributed versions of Falcon utilize a flexible event transport mechanisms – called DataExchange [12] – that does not depend on artifacts of the parallel programming environments used for application implementation, like PVM or MPI. 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 [50]. Falcon's low-level mechanisms are available via the Internet since early Summer 1994.

Current and future extensions of Falcon not only address additional execution platforms, but also include several essential additions to its functionality. First, users can insert simple tracing or sampling sensors into their code and 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 [42, 48]. Such views will be implemented with library support resident in both local and central monitors. The resulting higher level abstractions presented by views should be helpful in permitting them both to understand program behavior and to design suitable steering algorithms. Second, we are developing 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 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, and 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 will address the integration of higher level support for program steering, including graphical steering interfaces, and the embedding of Falcon's functionality into programming environments or libraries that support the process of program steering and the dynamic configuration of the required associated monitoring, in parallel and distributed systems. Such work with Falcon is one of the foundations of our current research on adaptable embedded applications [44], and it forms the basis of the development of 'distributed computational laboratories'[45] in which end users can inspect, control, and interact on-line with virtual or physical instruments (typically represented by programs) spread across physically distributed machines. One specific example being constructed by our group is a laboratory for atmospheric modeling research[28, 45], 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] Rob Armstrong, Pete Wyckoff, Clement Yam, Mary Bui-Pham, and Nancy Brown. Frame-based components for generalized particle methods. In *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing (HPDC'97)*, pages 50–59, August 1997.

[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. Also available from College of Computing, Georgia Institute of Technology, Atlanta GA, GTRC-TR-90/67.

[6] François Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby X. Yang. Distributed pc++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3), Fall 1993.

[7] K. Mani Chandy, Adam Rifkin, Paolo A. G. Sivilotti, Jacob Mandelson, Matthew Richardson, Wesley Tanaka, and Luke Weisman. A world-wide distributed system using java and the internet. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing (HPDC-5)*, August 1996.

[8] 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.

[9] Greg Eisenhauer, Weiming Gu, Thomas Kindler, Karsten Schwan, Dilma Silva, and Jeffrey Vetter. Opportunities and tools for highly interactive distributed and parallel computing. In Rebecca Koskela and Margaret Simmons, editors, *Parallel Computer Systems: Performance Instrumentation and Visualization*. ACM Press, 1996.

[10] Greg Eisenhauer, Weiming Gu, Eileen Kraemer, Karsten Schwan, and John Stasko. Displays of parallel programs: Problems and solutions. In *Proceedings of the Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, pages 11–20, Las Vegas, Nevada, July 1997.

[11] 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 Computer Society.

[12] Greg Eisenhauer, Beth Schroeder, Karsten Schwan, Vernard Martin, and Jeff Vetter. Dataexchange: High performance communication in distributed laboratories. In *Proceedings of the Ninth International Conference on Parallel and Distributed Computing and Systems (PDCS'97)*, October 1997.

[13] Greg Eisenhauer and Karsten Schwan. Design and analysis of a parallel molecular dynamics application. *Journal of Parallel and Distributed Computing*, 35(1):76–90, May 25 1996.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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. Also available as Technical Report GIT-CC-94-21, College of Computing, Georgia Institute of Technology.

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

[20] William J. Harrod and David A Schwartz. Vsip digital signal processing library routines: Version 0.3. *Draft Standard: Vector, Signal, and Image Processing (VSIP)* http://www.vsip.org/, September 1997.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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).

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

[27] Carol Kilpatrick and Karsten Schwan. Chaosmon – application-specific monitoring and display of performance information for parallel and distributed systems. In *ACM Workshop on Parallel and Distributed Debugging*, pages 57–67. ACM SIGPLAN Notices, Vol. 26, No. 12, May 1991.

[28] Thomas Kindler, Karsten Schwan, Dilma Silva, Mary Trauner, and Fred Alyea. A parallel spectral model for atmospheric transport processes. *Concurrency: Practice and Experience*, 8(9):639–666, November 1996.

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

[30] 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.

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

[32] 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.

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

[34] 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.

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

[36] 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.

[37] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tools. *IEEE Computer*, 1995.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] Daniela Ivan Rosu, Karsten Schwan, Sudhakar Yalamanchili, and Rakesh Jha. On adaptive resource allocation for complex real-time applications. In *18th IEEE Real-Time Systems Symposium, San Francisco, CA*. IEEE, Dec. 1997. to appear.

[45] Beth Schroeder, Greg Eisenhauer, Karsten Schwan, Jeremy Heiner, Vernard Martin, and Jeffrey Vetter. From interactive applications to distributed laboratories. *To appear in IEEE Concurrency*, 1997.

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

[47] Dilma Menezes da Silva and Karsten Schwan. A framework for developing high performance configurable objects. In *To appear in the Proceedings of the XI Brazilian Symposium on Software Engineering (SBES'97)*, Fortaleza, Brazil, October 1997.

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

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

[50] 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.

[51] J. Vetter and K. Schwan. High performance computational steering of physical simulations. In *Proc. Int'l Parallel Processing Symp.*, pages 128–32, 1997.

[52] J.S. Vetter. Computational steering annotated bibliography. *SIGPLAN Notices*, 32(6):40–4, 1997.

[53] J.S. Vetter and K. Schwan. Progress: a toolkit for interactive program steering. In *Proc. 1995 Int'l Conf. on Parallel Processing*, pages II/139–42, 1995.

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