From Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications
PDPTA'97 – Las Vegas – July 1997

Online Displays of Parallel Programs: Problems and Solutions

Greg Eisenhauer, Weiming Gu[*], Eileen Kraemer[†], Karsten Schwan, John Stasko
College of Computing, Georgia Institute of Technology
Atlanta, GA 30332

Abstract *The continuous display of information derived from on-line monitoring presents problems that are more difficult to resolve than those associated with off-line displays. Handling these problems requires a systematic approach affecting both the displays and the monitoring system itself. This paper discusses the problems inherent in on-line display and the approach to them taken by the Falcon monitoring system.*

Keywords: online displays, parallel programs

1 Introduction

In order to support monitoring and interactive steering of computer programs during execution, program users need accurate and easy-to-understand program information. Graphical displays have been shown useful in presenting data structures, algorithms, runtime program behavior, and performance information to human users. Most of the research on graphical displays has focused on the offline presentation of information, however. For users to interactively steer running programs, mechanisms are needed to handle the continuous stream of program information generated by online monitoring.

In the case of off-line displays, monitoring information can be sorted, analyzed, and filtered before the graphical display is invoked. The continuous display of on-line monitoring data is subject to different constraints. In particular,

- on-line monitoring systems tend to perturb the execution of the monitored program to an even larger extent than off-line monitoring systems
- efforts to minimize this perturbation often come at the expense of desirable characteristics in the data stream, such as a reasonable ordering of events
- despite all efforts, monitoring may still seriously perturb application behavior due to resource limitations

The ultimate goal of any monitoring system is to accurately display aspects of program execution to the user. The characteristics described present a serious obstacle in achieving prompt and accurate continuous display of program state information generated by online monitoring systems.

Various issues relating to time, ordering and perturbation appear in both off-line monitoring systems and distributed systems in general. In off-line monitoring, post-mortem processing is often used to resolve problems with ordering and perturbation[6]. The techniques used in off-line monitoring often rely on knowledge of the entire data stream, but the lack of this knowledge in on-line monitoring greatly complicates the resolution of these problems.

While the general problem of establishing a consistent ordering between events in a distributed system has been widely studied, the use of techniques such as "Lamport" and vector timestamps in the monitoring system are

[*] Weiming Gu is now at IBM Austin

[†] Eileen Kraemer is now at Washington University

[‡] This work was supported in part by the NASA AISRP Program under contract number NAGW-3886 and by NSF equipment grants CDA-9501637, CDA-9422033, and ECS-9411846.

not a panacea[5]. If the application already employs such techniques, a monitoring system might make use of them to maintain its own consistent view. But to impose distributed time-keeping techniques on all monitored applications is both undesirable and potentially quite difficult, both because of the program perturbation implied by the addition and because employing these techniques may require modifications to whatever communication mechanism is used. Given the wide variety of communication libraries and mechanisms used in parallel and distributed programming, such additions are impractical. Even if the additional perturbation was tolerable and distributed timestamps were available, they do not solve all the problems encountered in online monitoring. In particular, while these techniques provide a means to causally order two known events, they do not necessarily provide any indication of the existence of an as yet unreceived event. This creates a problem for online displays which must decide when they can act to create a comprehensible display.

Our research addresses the issue of maintaining comprehensible display behavior in the presence of perturbation and ordering problems which are endemic to the on-line monitoring of parallel and distributed applications. In this paper, we discuss an end-to-end tool-based approach that minimizes the program perturbation imposed by monitoring and resolves the ordering problem through semantic ordering of events. The use of an end-to-end approach also allows us to introduce the concept of *perturbation events* generated by the monitoring system and then used by the display environment. We use perturbation events as a mechanism through which the monitoring system can indicate exceptional levels of perturbation so that some aspects of the perturbation may be displayed to the user. In order to demonstrate the utility of these approaches and provide a context for discussing them, we also present a novel visualization for continuously displaying program events from threads-based parallel programs, called the *threads lifetime view.*

2 The Falcon System and its Thread Lifetime View

2.1 The Falcon System

Falcon is a toolkit that supports the on-line monitoring and steering of parallel and distributed programs. This toolkit contains mechanisms for monitoring and steering specification and instrumentation, a run-time system for on-line information capture, collection, filtering, analysis, and storage, mechanisms for interactive steering, and a variety of on-line displays. Part of the run-time system for information capture is based on Cthreads, a user-level threads package that runs on a variety of platforms. The detailed discussions of the Falcon monitoring and steering system in [2] highlight the system's ability to provide scalable monitoring services and describe its characteristics with respect to end-to-end latency and bandwidth. For this paper, we describe only those aspects of Falcon relevant to the on-line displays it supports.

In order to minimize program perturbation, the Falcon/Cthreads implementation uses per-thread event buffers. These buffers are emptied by a 'local monitor' agent represented as an additional thread that executes asynchronously with the program being monitored. This permits the efficient association of additional analyses with program monitoring, enables parallel execution of the monitoring facility by using multiple local monitoring threads for monitoring event capture and analysis, and dissociates monitoring activities from program execution, thereby reducing perturbation. It also enables monitoring of distributed programs by interposing a message mechanism between local monitoring agents and remote monitors or displays.

While it dramatically reduces monitoring perturbation, the use of monitoring can produce two undesirable side effects. First, because event orderings are not preserved across multiple buffers, the monitoring of even single-processor threaded programs is subject to the type of event misordering problems normally

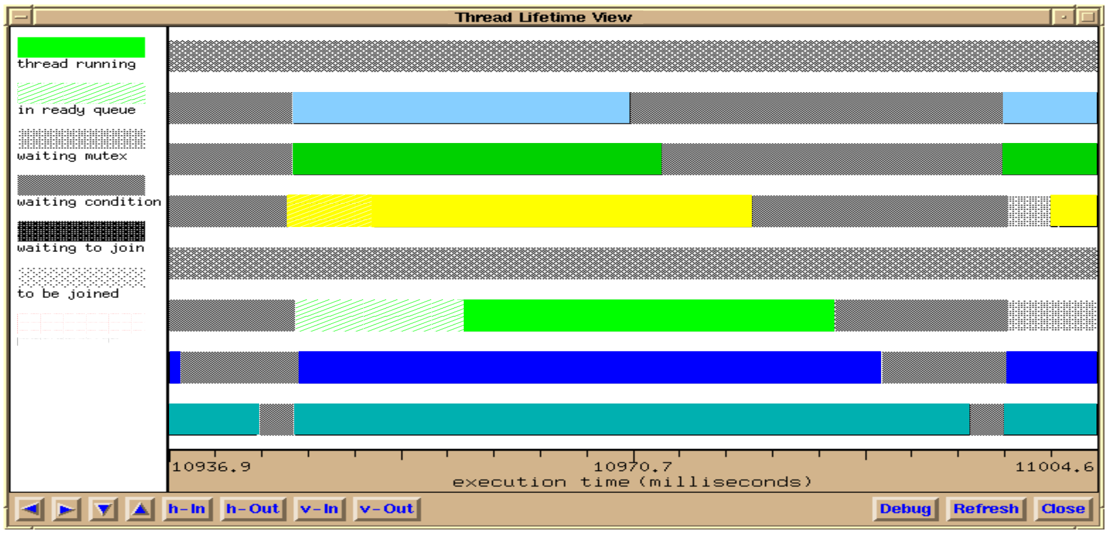

Figure 1: A thread lifetime view of a parallel atmospheric modeling application. The display uses colors and patterns to represent different threads and different thread states.

associated with monitoring in a distributed system. That is, while events may be ordered within each individual buffer, they are not necessarily collected from these buffers by the local monitor in an order corresponding to the actual order of program events. Secondly, if event rates are higher than buffer transmission rates or the rates at which they are emptied by local monitors, event buffers can fill up. When this occurs, perturbation becomes abnormally high because the application generating the event blocks on the buffer until it is emptied so that no events are lost. Such exceptional perturbation, if not recognized by the monitoring system and depicted in the display, can seriously distort the user's perception of program behavior.

Falcon's use of event buffers to minimize perturbation does exacerbate the problems of event misordering and large program perturbation due to resource limitations in the monitoring system, but they are by no means unique to Falcon.

2.2 The Thread Lifetime View

One on-line display provided by Falcon is the *thread lifetime view*, which is designed to help users understand the dynamic behavior of threads-based parallel programs. This display uses the default monitoring sensors embedded in interactive Cthreads to provide system-level information about application behavior. The display is implemented with the Polka animation library [8]. Polka provides a flexible animation scheduling policy, permits different temporal mappings of program events to their animations, and therefore facilitates rapid construction of on-line displays *.

The thread lifetime view shows the different states of threads over time. The state information depicted in the view includes thread execution time, blocking time, waiting time in ready queues, the identifiers of conditions or mutex locks on which threads are blocked, and thread identifiers. From this information, users can easily discern the time a thread spends doing useful computation versus waiting for other threads, the degree to which different threads' executions are synchronized, processor utilization, and other useful program and performance information. When a new thread is forked, a narrow horizontal bar is created to represent the new thread's lifetime, and a vertical line is drawn from the parent thread to the child thread at the time of the fork event. A narrow bar terminates when the thread exits, or, in the case of attached threads, when the thread is joined. In the case of a join, another vertical line is drawn from the caller thread to

*Polka is described and evaluated in detail in [8].

the thread it is joining. After a thread exits, the resulting empty space in the display can be reused for depiction of a new thread, if present.

Figure 1 shows a snapshot of the execution of an atmospheric modeling application. In this figure, the solid color patterns represent threads in running states, while the dark gray pattern represents threads waiting for conditions. The lightly dotted pattern indicates that a thread has called thread_exit and is waiting to join to another thread. The heavily dotted pattern indicates that a thread is in a processor's ready queue, that is, it is waiting for another thread using the processor to complete its execution.

The thread lifetime presented above has proven useful for diagnosing performance problems and for understanding unusual program behavior that may indicate coding errors. However, the accuracy and integrity of the lifetime view can be threatened by out-of-order events and program perturbation.

3 On-line Event Reordering and Program Behavior

Displays like the thread lifetime view of Figure 1 can be misleading if the information being displayed does not correspond to the execution the program would experience without monitoring. As discussed above, many mechanisms that monitoring systems employ to reduce undesirable program perturbation and event latency have the side-effect of destroying some or all of the ordering in the event stream. This is not a difficult problem for off-line displays because the set of monitoring events can be sorted before they are sent to the display. However, an online display may find that the order in which it receives monitoring events may not correspond to the actual or semantically meaningful order in which program events occur. Such misorderings can confuse users and, more critically, can cause failures of the animation itself. As an example, consider that a semantically meaningful ordering of events from a thread library requires that the thread_fork event creating a thread precede any event executed by the new thread. An off-line display that processes sorted events would likely rely on this ordering and might crash or produce an inaccurate display if presented with mis-ordered events. One can take the approach that every display is responsible for handling events in any order, but a system-level approach to the problem is more efficient.

The reordering problem. The diagnosis and correction of out-of-order events is a common problem in parallel and distributed monitoring systems. Existing offline systems (e.g., ParaGraph[3] and SIEVE[7]) rely on a sort by timestamp value to impose a total order on all events stored in event files. The on-line nature of the Falcon monitoring system precludes using such a solution. Additionally, sorting by timestamp order does not entirely eliminate the problem of out-of-order events[5] because coarse clock granularities and poor clock synchronization among different processors may lead to event timestamps that do not accurately reflect the actual order of program execution. While perfect clocks, or the use of techniques such as "Lamport" or vector timestamps in the monitoring system, can give the ability to causally order two known events, they do not necessarily provide any indication of the existence of an as yet unreceived event. This creates a problem for online displays which must decide when they can create a comprehensible display.

Our approach. The previous discussion of out-of-order events makes clear that the use of timestamps is not sufficient for determining and enforcing suitable, global event orderings. Falcon addresses this issue by employing an *ordering filter* between the local monitors and the display system[1]. This filter ensures that the event stream reaching the display system adheres to a specified, known causal ordering among events of various types. The ordering filter has knowledge of all execution threads, mutex locks, and conditions identified as occurring in the event stream. The algorithm employed by the filter follows a "minimum-intervention policy"; it examines each event in

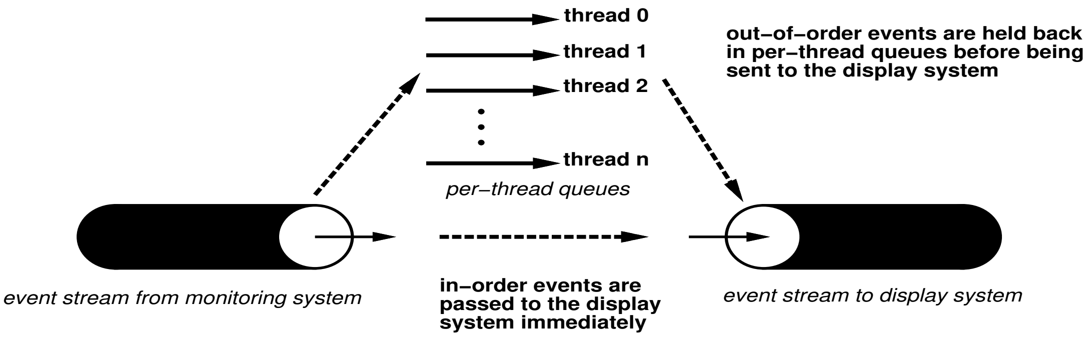

Figure 2: Architecture of the on-line event reordering filter.

the stream arriving from the monitoring system, checks the applicable ordering rules for this event type, and if no rules are violated, forwards the event to the display system. If a rule violation is indicated, the event is held back until the rules are satisfied.

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

```
mutex_end_lock t m n ← ((thread_init t) &&
                        (mutex_init m) &&
                        (mutex_unlock m n-1))
```

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

The rule applied to a mutex lock is one of many rules implemented by the reordering filter (see the Appendix for a complete listing of these rules). Moreover, even for this single rule, each of its expressions is associated with another set of ordering rules that must be met. The rules appearing in the Appendix are written to reflect the logic of the current filtering code. Our future work addresses the automatic generation of filtering code from formal rule specifications like the one shown above. Moreover, the generality of the rule-based approach to capturing and representing relevant program semantics is demonstrated in [5] by applying it to other threads packages as well as to distributed computing platforms like PVM.

Implementation of on-line reordering. Figure 2 outlines the implementation of the event reordering filter. The event stream at the left arrives from the monitoring collection mechanisms only partly ordered. (Events from a particular thread arrive in order, but events from different threads can be misordered by the buffering and transmission scheme.) The event stream forwarded to the display system shown at the right hand side of the figure will be ordered according to the specified ordering rules. To attain this ordering, the filter maintains an ordered queue for all events with the same thread_id encountered in the event stream, shown at the center of the figure. This queue contains only events that are not ready to be processed (that do not yet satisfy the rules). Other events are immediately forwarded to the display system. The ordering filter then continues to examine new events, checking the head of each active queue in every round to see if it is now possible to place the

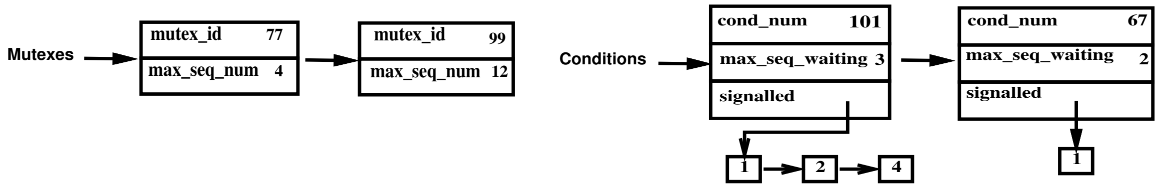

Figure 3: Sample data structures used in event reordering.

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

Additional data structures in the ordering filter are assigned to mutex_ids and condition_ids, each of which is represented by a data structure that keeps track of the sequence numbers associated with this abstraction that have been processed thus far. An example of these data structures is shown in Figure 3, where threads are waiting on both condition variables. These data structures are dynamically allocated as the events are observed in the stream. A mutex_init or mutex_alloc event causes data structure allocation for this mutex_id, and the sequence number for the mutex is initialized to 0. No event associated with this mutex_id may be processed until after the mutex_init or mutex_alloc events have occurred. In addition, mutex_end_lock and mutex_unlock events have sequence numbers, and are required to be processed in sequence number order. Similar data structures exist for events concerning condition variables, again requiring that condition_init or condition_alloc events precede any condition_signal or condition_broadcast events and that their initial sequence numbers be set to 0. More specifically, a condition_end_wait event for sequence number n must be preceded by a condition_signal on sequence number n or a condition_broadcast on a range of sequence numbers containing n. In turn, a condition_signal on n must be preceded by a condition_begin_wait on n. A condition_broadcast on $n1$ ... $n2$ must be preceded by a condition_begin_wait on $n2$. The condition_begin_wait on sequence number n must be preceded by condition_begin_wait on sequence number n-1.

Comparison with other approaches. Our approach to online event reordering shares some common elements with other online and offline systems[6, 3, 7]. Of these systems, the Program Visualization project[4] at IBM is perhaps the most closely related. This system relies on a Lamport-like algorithm to perform on-the-fly event reordering for program visualization. This approach has the advantage of not requiring application-level specification of causality, but it also requires the exclusive use of an appropriately modified message passing library to maintain the Lamport clocks. Our approach does not require such a library and is also applicable to applications which are implemented in other programming styles, such as the threads-based shared-memory atmospheric modeling application used as an example above. Coupled with tools to ease the process of creating causality filters[5], we expect our approach to prove more portable.

Effects on monitoring performance. While a complete analysis of the behavior of ordering filters is beyond the scope of this paper, several observations are relevant. Because the rules embody causality in program execution, extreme situations such as a rule requiring all events to be held back awaiting an event that only comes at the end of program execution should not occur. An event should be held back only for the time it takes for its causal predecessors to reach the ordering filter. This time is a function of the specifics of the monitoring system and its configuration. See [1] for typical values in the Falcon system. Also, ordering filters may affect the overall latency of the monitoring system, both for events that are held back and those that are immediately passed to the display system. In the former case, the additional latency waiting for causal predecessors to arrive is the price paid for order and should be limited in a properly functioning monitoring system. In the later case, the costs depend upon the nature of the implementation. In Falcon, the reordering filter is implemented as part of a central monitor which combines multiple event streams. In this case, the additional cost of reordering filter is limited to the computational cost of executing the rules. While those costs will vary with the nature of the rules, they are likely to be small compared with the other costs in the monitoring system.

4 Perturbation Events and Program Performance

The thread lifetime view described in Section 2.2 helps users understand program performance problems only if the thread running and waiting times correctly reflects the application's actual execution and synchronization behaviors. During normal operation, the perturbation imposed by monitoring with Falcon is largely limited to the cost of inserting an event into a shared-memory buffer[†]. Normally this small perturbation has little effect on the application and displays like the lifetime view depict program execution much as it would have been without monitoring. However, if the application is producing events at a rate faster than the monitoring system can pass them on to the display system, Falcon's event buffers can fill and the application will experience an exceptionally large perturbation due to monitoring. The result can be a highly misleading display. The earlier depiction of the thread lifetime view (Figure 1 actually results from such a misleading situation.

A more detailed inspection of the view in Figure 1 is helpful to understand the problem to be addressed. The solid color bars shown in the figure (the first and the fifth, counting from top to bottom) represent 'worker' threads used in the atmospheric modeling application. Each of these worker threads is responsible for a partition of the computation space. The three bars below each worker thread are 'helper' threads, which are employed by the worker thread to help calculate separate terms in its computation. The atmospheric modeling application is a time-stepped simulation and the simulation algorithm performs a barrier synchronization after each time step. The figure indicates that the third helper thread of the top worker thread and the first helper thread of the bottom worker thread waited a very long time for mutex locks. In addition, the second and third helper threads of the bottom worker thread have significantly longer computation times than other helper threads. These superficial observations imply that the program has unbalanced work loads and improper synchronizations. In fact, when first using this view, one of the atmospheric modeling application's implementors spent several hours investigating the the load imbalance problem indicated by the view only to discover that it was illusionary. In reality, the application behavior shown in this display has been seriously distorted by the monitoring system. In this case, the situation was created by the use of an unsuitable configuration of the monitoring system (a single local monitoring thread used to monitor the large-scale program running on 37 processors) which was quickly overwhelmed by events from a large number of computational threads.

All monitoring systems perturb the monitored system to some extent. While this perturbation should be minimized, it is most tolerable in small and predictable amounts as the application then behaves largely as it would behave if the monitoring were not present[‡]. However, when the event rates from an appli-

[†]Strictly speaking, the introduction of online monitoring may perturb the application in innumerable ways, including modifying application caching and paging behavior and consuming network and processor resources. See [1] for more discussion of these costs.

[‡]There exist some applications which cannot tolerate the smallest perturbation without serious behavior changes. These applications are not generally candidates for monitoring.

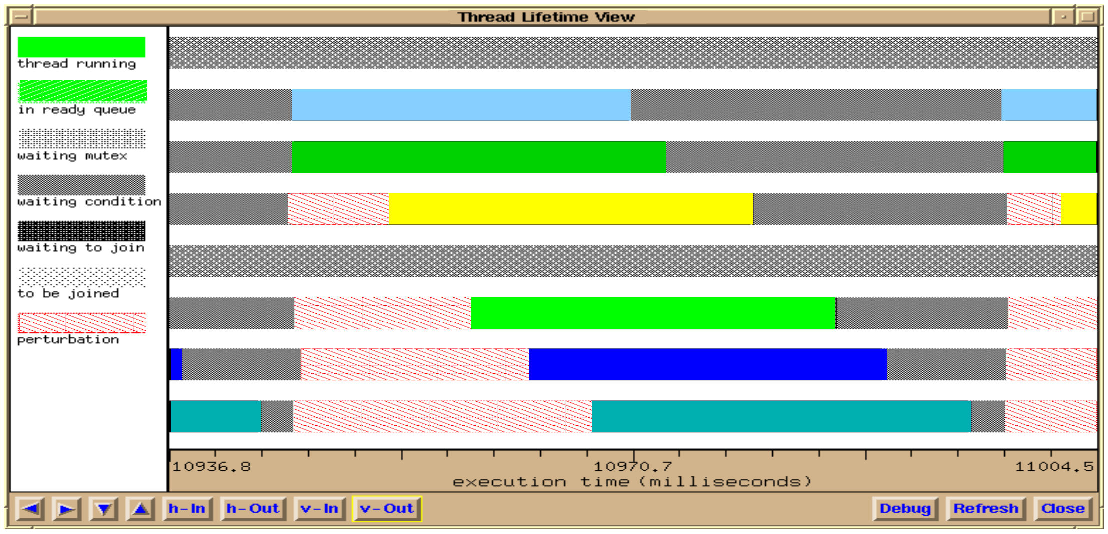

Figure 4: A thread lifetime view of the atmospheric modeling application. Perturbation events are shown in this display to identify indirect perturbation caused by the monitoring system.

cation exceed the monitoring system's ability to process data, perturbation grows unexpectedly large and can seriously distort the behavior of applications which are normally tolerant of small perturbation. In a system that normally minimizes monitoring perturbation, this exceptional perturbation can produce unexpectedly distorted displays.

Falcon uses an end-to-end approach to handle this sort of misleading situation. At the instrumentation end, Falcon captures perturbation due to filled buffers with *perturbation events*. Specifically, perturbation due to monitoring saturation is reported with additional perturbation fields added to regular monitoring events. In Falcon's instrumentation code, the last stage of event generation is a busy loop trying to insert the event into a monitoring buffer. Normal cost of instrumentation is incurred if the first try succeeds (buffer is not full). Otherwise, additional delay occurs because of full monitoring buffers. Such exceptional perturbation can be calculated by subtracting the time of the first insertion attempt from the time of the last successful attempt. Note that it causes little additional overhead to include this additional information as the timestamp calculations are performed when the application would be blocked (waiting for monitoring buffers to be cleared by monitoring threads) and the information is piggybacked with regular monitoring events.

For this information on exceptional perturbation to be useful, it must be displayed to the user. While the manner in which this information is best shown is inherently display-specific, we use the lifetime view as an example of one approach. Figure 4 shows the same execution of the atmospheric modeling code, this time using the special perturbation information to display a perturbation state. It is clear from this view that the helper threads have balanced work loads, but that their execution times are extended due to monitoring perturbation experienced by the helper threads of the bottom worker. Specifically, the extremely long running times apparently experienced by the third helper thread for the top worker and the first helper thread for the bottom worker (shown in Figure 1) are not due to a load imbalance. They are due to additional thread waiting times experienced when writing events to the monitoring buffers. From this example, it should be evident that perturbation events help users understand the monitoring system's contribution to total thread execution and waiting times.

The above example demonstrates that this simple approach of gathering perturbation information from the monitoring system and then displaying it to users can help users understand how the monitoring system has affected their programs. Similar perturbation problems occur when monitoring a distributed program with a distributed monitoring system, in which events are collected and inserted into separate buffers and then sent to a remote central monitor. While the particulars of generating perturbation events described above is specific to the Falcon system and its use of event buffers, the basic "send event" mechanism employed in other monitoring systems can be modified to generate perturbation events.

5 Conclusions

This paper describes a novel graphical view depicting the behavior of threads-based parallel programs (called the thread lifetime view), and it discusses two specific issues concerning the on-line display of monitoring information: (1) the identification of specific perturbation information from a stream of monitoring events and (2) the semantically meaningful correction of misorderings among such events prior to their display. Results are obtained with the Falcon on-line monitoring system and with application programs executing on shared memory parallel machines. Insights derived from this work concern the feasibility of the on-line analyses required for generating meaningful threads displays, and they demonstrate the need to make displays and analysis code aware of critical properties of the underlying parallel programming libraries (e.g., the Cthreads programming library used in this work) and of the monitoring system used for event capture and collection (in this case, the Falcon on-line monitoring system[2]).

References

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

[2] 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.

[3] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, pages 29–39, September 1991.

[4] Doug Kimelman and Dror Zernik. On-the-fly topological sort – a basis for interactive debugging and live visualization of parallel programs. In *Proc. of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1993.

[5] Eileen Kraemer. Causality filters: A tool for the online visualization and steering of parallel and distributed programs. In *Proc. of the 11th IPPS*, Geneva, Switzerland, April 1997.

[6] Allen D. Malony, Daniel A. Reed, and Harry A. G. Wijshoff. Performance measurement intrusion and perturbation analysis. *IEEE TPDS*, 3(4):433–450, July 1992.

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

[8] 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.

Appendix : List of Cthreads Event Reordering Rules

Following is terminology used in this section:

$\leftarrow$ = "is allowable if preceded by"

`p` = processor no.	`t` = thread no.
`ct` = child thread no.	`c` = condition no.
`jn` = join_to thread no.	`m` = mutex no.
`x` = don't care value	`n` = sequence no.

The ordering rules for all events from default monitoring of Cthreads programs are listed below. A brief explanation for each rule is provided. Each mutex number `m` and condition number `c` is initially set to 0. A distinguished thread, the main thread, is initially the sole active thread. Also, while Cthreads provides both `init` and `alloc` operations on mutexes and conditions, we ignore the `alloc` operation in this presentation for compactness and clarity.

```
thread_init t ← ((t ==0)
            || (thread_fork x t))
```

This is the initial event for thread `t`. All prior events pertaining to this thread are ignored. If this is the main thread (t==0), an internal buffer is created for this thread number, and it is turned "on".

```
thread_fork t ct ← (thread_init t)
```

The thread must have been initialized prior to this event. An internal buffer is created for the child thread, and the child's thread id is activated.

```
thread_exit t ← (thread_init t)
```

The internal buffer is de-allocated and the thread is turned "off". Any succeeding events recorded by this thread are ignored.

```
thread_begin_join t jt ← (thread_init t);
```

```
thread_end_join t jt ← ((thread_init t)
            && (thread_exit jt));
```

The `thread_exit jt` event for the join_to thread `jt` must have occurred before this event.

```
mutex_init t m ← ((thread_init t) &&
            && !((mutex_init x m)
                 || (mutex_alloc x m)));
```

No prior `mutex_init` or `mutex_alloc` event may have occurred.

```
mutex_begin_lock t m n ← ((thread_init t)
            && (mutex_init x m));
```

A `mutex_init x m` must precede this event.

```
mutex_end_lock t m n ← ((thread_init t)
            && (mutex_init x m)
            && (mutex_unlock x m n-1))
```

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

```
mutex_unlock t m n ← ((thread_init t)
            && (mutex_init x m)
            && (mutex_end_lock x m n));
```

A `mutex_init x m` event must precede this event, as must the acquisition of the lock, `mutex_end_lock t m n`.

```
condition_init t c ← ((thread_init t)
            && !(condition_init c));
```

No prior `condition_init x c` or event may have occurred.

```
condition_begin_wait t c n m ←
    ((thread_init t)&&(condition_init x c)
     && (condition_begin_wait x c n-1 m));
```

A `condition_init x c` event must have occurred, as must the preceding `condition_begin_wait t c n-1 m` event.

```
condition_end_wait t c n m ←
    ((thread_init t)&&(condition_init x c)
    && (condition_signal x c n));
```

A `condition_init x c` event must have occurred, and so must the corresponding `condition_signal x c n` event.

```
condition_signal t c n ←
    ((thread_init t)&&(condition_init x c)
     && (condition_begin_wait x c n m )
     && (condition_signal x c n-1));
```

A `condition_init x c` event must have occurred, and so must the preceding `condition_signal x c n-1` event.

```
condition_broadcast t c n1 n2 ←
    ((thread_init t)&&(condition_init x c)
     && (condition_begin_wait x c n2 m )
     && (condition_signal x c n1-1));
```

A `condition_init x c event` must have occurred, and so must the preceding `condition_signal x c n-1` event. Internally, the broadcast is translated to the corresponding sequence of `condition_signal` events, and the data structures updated accordingly.

Additionally, the following events must be preceeded by `(thread_init t)`:

`thread_detach t`	`mutex_clear t m`
`thread_yield t`	`mutex_free t m`
`thread_set_name t`	`mutex_set_name t m` +
`condition_free t c`	`condition_clear t c`
`condition_set_name t c`	