# MKM: Mach Kernel Monitor
# Description, Examples and Measurements

Ted Lehr*, David Black,
Zary Segall and Dalibor Vrsalovic

March 1989

CMU-CS-89-131

School of Computer Science
and *Department of Electrical and Computer Engineering

## Abstract

*Visualization* of parallel and distributed algorithms and their intimate interaction with the operating system is currently part of the research community's main agenda. This paper introduces and evaluates MKM, the Mach kernel monitor. We show that MKM, when coupled with a user level monitor and visualization system (the PIE system), is able to fulfill the double role of visualizing kernel behavior in the presence of a workload as well as the influence of the operating system kernel on user algorithms.

As MKM is expected to be widely used as part of the Mach standard distribution, this paper reports on the MKM design concepts with emphasis on examples showing the usefulness of the system. The report concludes with the measurement and analysis of MKM intrusiveness and ways to compensate for its overhead.

*Index Terms:* context switching, Mach, monitor, parallel programming, performance measurement, performance evaluation, PIE, thread scheduling

This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), Arpa Order No. 4864, monitored by the Space and Naval Warfare Systems Command under contract N00039-87-C-0251, and in part by the National Science Foundation, Grant No. CCR-86-02-143.

The views and conclusions contained in this document are those of the authors and and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the National Science Foundation or the U.S. Government.

# Table of Contents

1 Kernel Monitoring: Observing Context Switches
2 Background: Mach and PIE
2.1 Tasks and Threads
2.2 Interprocess Communication and Ports
2.3 Portability
2.4 Kernel-Monitoring and PIE
3 Examples of Kernel Monitoring: Understanding the Figures
3.1 Comparing Forking Behavior On Two Versions of Mach
3.2 Time Sharing on the Two Kernels
4 Monitor Description
4.1 General Architecture
4.2 Monitor system calls
5 Monitor Performance Overhead
5.1 Results on the micro-Vax
5.2 Examples of Overhead
5.3 Statistical Calculations of Overhead
5.4 Data Analysis
5.5 Measurement Methodology
5.5.1 Measuring Sensor Firing Times
5.5.2 Measuring Constant Overhead
6 Work Pending
6.1 Monitor Enhancements
6.2 PIE Enhancements
7 Conclusions

## List of Figures

Figure 1: Old Kernel: Entire view of matrix multiplication 3
Figure 2: New Kernel: Entire view of matrix multiplication 3
Figure 3: Old Kernel: A Zoom view of matrix multiplication 4
Figure 4: New Kernel: A Zoom view of matrix multiplication 4
Figure 5: New Kernel: Zoom view of a smaller matrix multiplication 6
Figure 6: Improved Kernel: Entire view of matrix multiplication 6
Figure 7: Improved Kernel: Zoom view of matrix multiplication 7
Figure 8: Improved Kernel: Context-switch "flutter" in main and collector 7
Figure 9: General Kernel Monitor Architecture 9
Figure 10: View of the Finish of another Matrix Multiplication (Old Kernel) 13
Figure 11: Compensated View of the Finish of the Same Matrix Multiplication (Old Kernel) 13
Figure 12: Code for measuring kernel sensor firing times 16
Figure 13: Code for measuring kernel monitor overhead 17

## List of Tables

Table 1: Variable Definitions 11
Table 2: Context-Switch Sensor Firing Statistics of Loop iterations 12
Table 3: Statistics for Measuring Static Overhead of Monitor Code 12

## 1 Kernel Monitoring: Observing Context Switches

During the creation of a computation, designers usually measure how well successive prototypes perform. The designer of an operating system's scheduler, for example, might gather statistics on how long processes wait in a run queue or the number of context-switches occurring during a time interval. If the distribution of the data points suggest that some processes wait for excessively long periods before being rescheduled, he would want to determine source of the unwelcome behavior. Distributions alone, however, are often little help in identifying where performance problems lie. Is there any pattern to when the scheduler forces some of its processes to wait? When does it do this and how often? These are questions statistical information can raise, but is hard pressed to answer.

The designer must seek additional information about scheduler. In the case of context-switches, he could try to order the data chronologically and find what entities were associated with each switch in hope of discovering event patterns surrounding the problematic behavior. If the designer could present the performance of the kernel in a more tractable way he could hone in on the source of the unsatisfactory performance. The problem we address then, is the collection and presentation of kernel behavior so that its performance can be evaluated quickly.

Our solution is a low overhead, high band-width kernel monitor designed for the Mach operating system [1] for recording context-switches and the entities associated with them. It is integrated with a unique programming environment called PIE [3], [11], which uses performance graphics to *visualize* context-switching in tractable and elegant ways. In section 2 we first give some background information on Mach and PIE. This is followed by a discussion in section 3 of the advantage of being able to visualize this kind of behavior using an example which vividly showing significant differences between two schedulers of separate versions of the Mach operating system. In section 4, we describe the basic architecture and system calls of the Mach Kernel-Monitor (MKM). Because a kernel-monitor is of little value if it imposes a heavy performance penalty upon the kernel and its computations, in section 5 we discuss the overhead incurred by Mach in the presence of the monitor. Finally, we discuss direction of our pending work in section 6.

## 2 Background: Mach and PIE

Mach is an operating system under development at Carnegie Mellon University for integrating support of networks of uniprocessors and multiprocessors while presenting a Unix style software environment. The basic Mach primitives support Unix functionality by placing it *outside* the Mach kernel. Although Mach has a number of abstractions for supporting networks and multiprocessors, only the salient abstractions like *tasks* and *threads* and *ports* need to be discussed here in order to more clearly understand the upcoming examples.

### 2.1 Tasks and Threads

A task is an address space and a set of system resources (eg. file descriptors) while a thread can be thought of as a program counter and register set. Each task holds a large virtual address space which can be allocated and de-allocated by any thread running within that task. At least one thread is always associated with a task (a Unix process can be emulated as a single thread executing within a task), although several threads may share the task's resources in parallel. The task and thread abstractions constitutes a major portion of Mach's support for parallel computing on multiprocessor systems.

### 2.2 Interprocess Communication and Ports

The basic Mach communication abstraction is a kernel protected entity called a *port* for sending and receiving typed messages. A port is basically a protected queue with associated *send* (enqueue) and *receive* (dequeue) rights. Interprocess communication (IPC) in Mach is transparently extendable over networks. The Mach kernel implements message passing between tasks on the same host. Messages sent to ports belonging to tasks executing on remote hosts are sent to a network server which forwards the message over the network. Mach provides an interface language to generate the client/server interfaces that place network communication functionality outside the kernel. By moving this functionality outside the kernel, Mach increases the flexibility of each host in choosing how data is represented, how network security issues are resolved and how communication protocols are implemented.

### 2.3 Portability

Because Mach is targeted to run on a variety of machines, close attention has been paid to ensuring that the Mach implementation is truly portable especially in the area of virtual memory support. Because virtual memory management units differ widely among different machines, Mach's virtual memory implementation is divided into machine dependent and independent parts. For more information about this and other portability issues addressed by the Mach designers, see [1], [2], [4], [10] and [12].

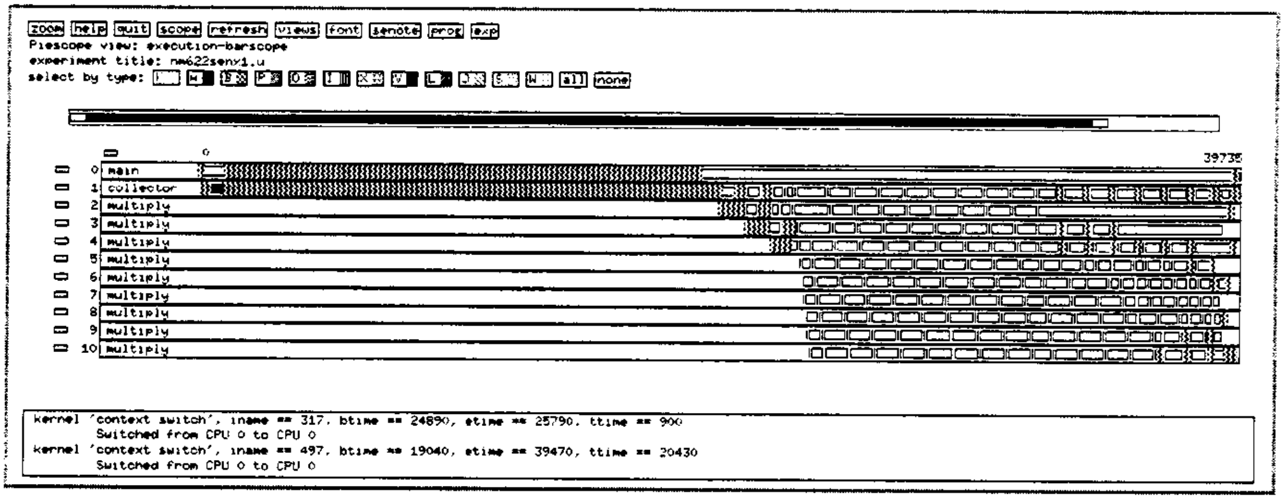
### 2.4 Kernel-Monitoring and PIE

The Mach kernel-monitor is modeled after the structure of Mach abstractions such as tasks and threads. The inspiration for MKM arose out of the needs of the Parallel Programming and Instrumentation Environment (PIE). PIE gives programmers ways to visualize the execution of computations. It allows a user to insert user-level sensors in a program using a special editor and then visually represents an execution of that program based on data from the sensors. PIE supports monitoring of executions using a run-time library that manages not only a user observation monitors but Mach kernel-monitors, MKMs, as well. Using data from the sensors, PIE displays graphical representations of the observed performance. Using a graphics tool called *PIEscope*, the environment displays context-switch data along with user-level information about the execution of computations. More information about PIE can be found in [11] and [3].

## 3 Examples of Kernel Monitoring: Understanding the Figures

Figures 1 through 8 are selected PIEscope views of a matrix multiplication computation executing on different versions of the Mach kernel. The initial thread of the matrix multiplier is `main`. In addition to forking off the `collector` thread and the first `multiply` thread, `main` also performs the I/O for the computation as well. As part of the PIE observation monitor, the `collector` records user and kernel events belonging to the computation. Although spawned by `main`, the `collector` is created and terminated independently of user written code by special libraries linked with the computation at compile time. The `multiply` threads are the workhorses of the computation. When each `multiply` thread begins executing, it first checks whether there is sufficient work left to warrant spawning another thread. If there is, it forks off another `multiply`. It then determines its share of the workload and begins working on its parts of the matrices.

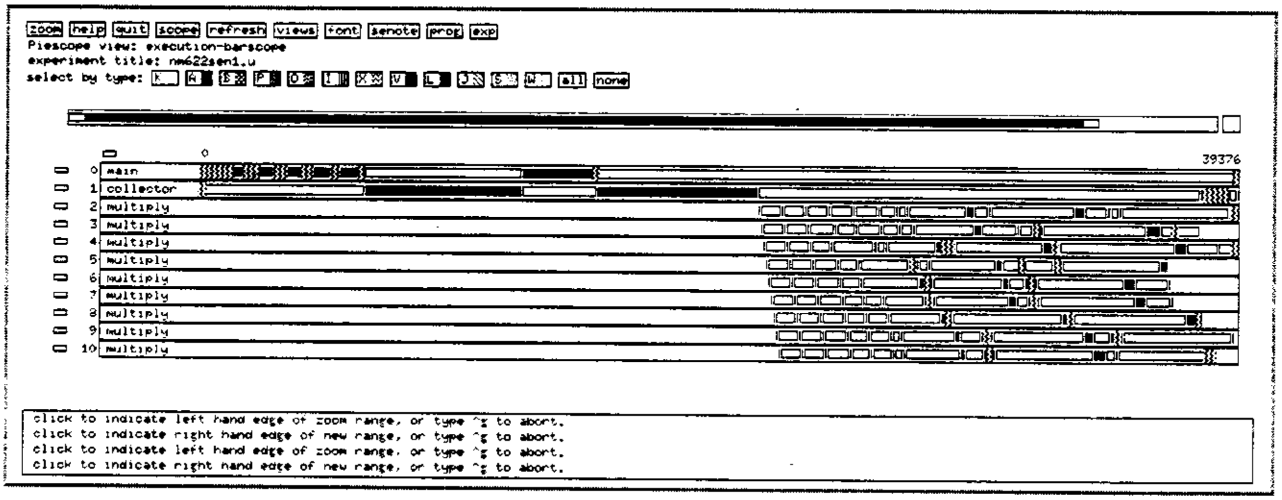
Figure 1 depicts the micro-Vax II execution of the matrix multipler on a kernel officially labeled as **XF29**, which will be referred to as the **Old**. Figure 2 depicts the the same computation on an newer kernel,



```
Piescope view: execution-barscope
experiment title: nm622senx1.u
```

```
kernel 'context switch', iname == 317, btime == 24890, etime == 25790, ttime == 900
        Switched from CPU 0 to CPU 0
kernel 'context switch', iname == 497, btime == 19040, etime == 39470, ttime == 20430
        Switched from CPU 0 to CPU 0
```

**Figure 1:** Old Kernel: Entire view of matrix multiplication



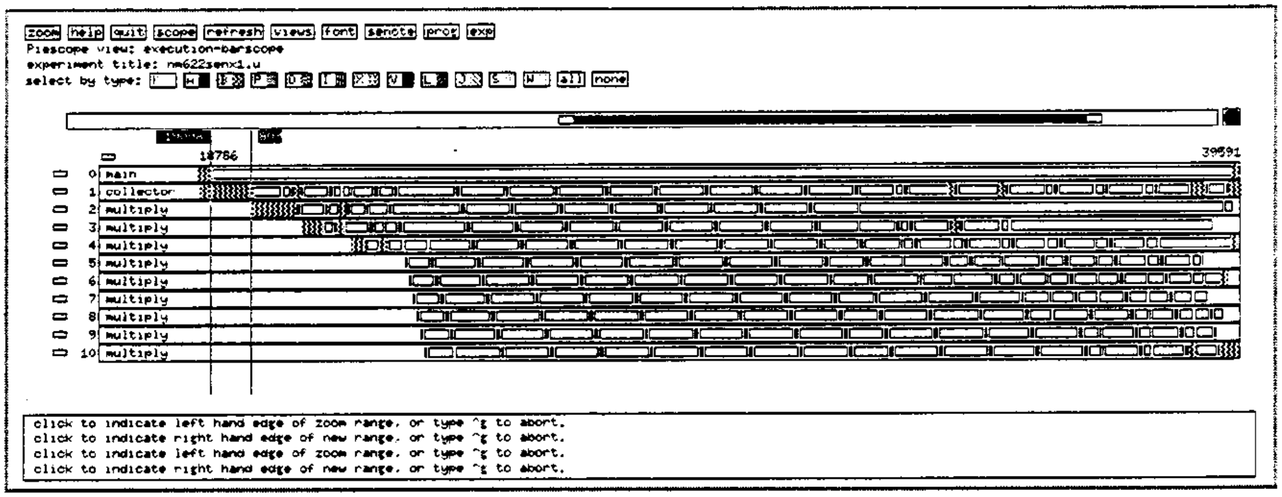
```
Piescope view: execution-barscope
experiment title: nm622sen1.u
```

```
click to indicate left hand edge of zoom range, or type ^g to abort.
click to indicate right hand edge of new range, or type ^g to abort.
click to indicate left hand edge of zoom range, or type ^g to abort.
click to indicate right hand edge of new range, or type ^g to abort.
```

**Figure 2:** New Kernel: Entire view of matrix multiplication

designated as **CS3c**, hereafter referred to as **New**. Figures 3 and 4 are "zoom" views showing greater detail of each computation. Each higher resolution view also contains a pair of "metering" lines which measure the time between them. Although the two kernels are dissimilar in a number of respects, the examples concentrate only on the difference between their scheduling policies. A third kernel officially designated **CS5a** but which we call **Improved** is shown in Figures 6 - 8. It is discussed after we first compare the New and Old kernels. In each of the kernel tests, the micro-Vax II is operating in single-user mode in order to reduce the number of competing processes and thereby the occurrence of seemingly random events during the computation's execution.

In all the PIEscope figures, time is measured in milliseconds on the horizontal while the computation's threads are ordered on the vertical. Dark rectangles represent periods when threads are running. White

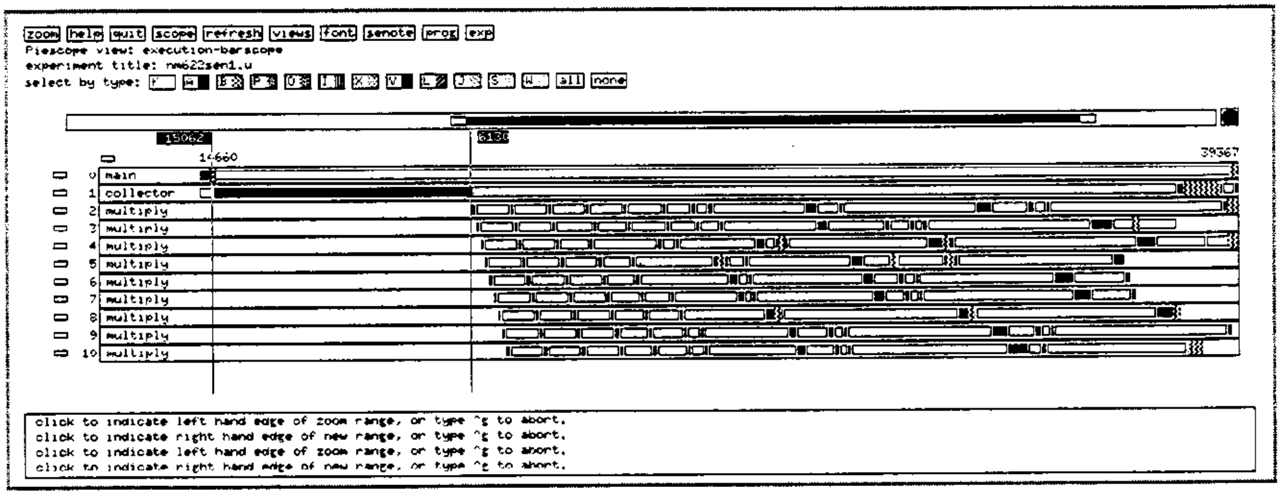


Piescope view: execution-barscope
experiment title: nm622senx1.u

```
click to indicate left hand edge of zoom range, or type ^g to abort.
click to indicate right hand edge of new range, or type ^g to abort.
click to indicate left hand edge of zoom range, or type ^g to abort.
click to indicate right hand edge of new range, or type ^g to abort.
```

**Figure 3:** Old Kernel: A Zoom view of matrix multiplication

rectangles represent periods when threads are switched out. Because the figures depict uniprocessor executions, if a swath is cut vertically through any point in a view, only a single thread (a single dark rectangle) will be found executing at that time. In parts of some of the views, in Figure 1 for example, there are slightly confusing sets of consecutive white rectangles. Between these apparently contiguous periods are comparatively shorter episodes when the associated thread is actually running but the graphic view does not have the resolution to show this. In cases where the resolution is insufficient to display large numbers of events in given period, PIEscope fills the affected periods with serrated vertical lines or "squiggles"[1].



Piescope view: execution-barscope
experiment title: nm622sen1.u

```
click to indicate left hand edge of zoom range, or type ^g to abort.
click to indicate right hand edge of new range, or type ^g to abort.
click to indicate left hand edge of zoom range, or type ^g to abort.
click to indicate right hand edge of new range, or type ^g to abort.
```

**Figure 4:** New Kernel: A Zoom view of matrix multiplication

[1] PIEscope allows the user to "zoom in" on any part of the view in order to increase the resolution .

### 3.1 Comparing Forking Behavior On Two Versions of Mach

As can be seen in Figures 2 and 1, the multiplier computation does not behave identically on the New and Old kernels. There is, for instance, a notable difference in the forking performance of their schedulers. Figures 3 and 4 show quite clearly that New successively spawns the first three `multiply` threads more quickly than does Old. These views also show that the new kernel eliminates the context-switch "flutter" that occasionally occurs in Old just as a thread is forked off. Figure 3 shows `multiply` threads 2, 3, and 4 undergoing considerable context-switching soon after they are spawned on the old kernel. The `collector` undergoes similar behavior. Interestingly, much of the context-switching is of the thread to itself.

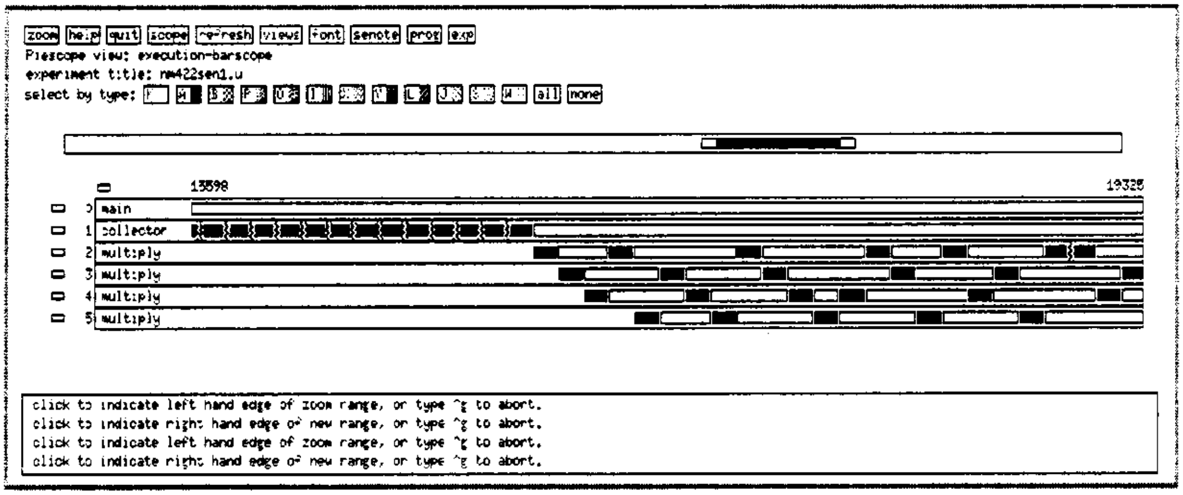
As might be expected, the spawning performance of New does not come without some undesirable baggage. Recall that the first `multiply` thread is forked off by `main`, not by the `collector`. After spawning the thread, `main` does a join and switches out, waiting for its children to finish. Examining Figure 4 again, it is clear that New does not do this until after giving the `collector` an execution time-slice of over six seconds. Figure 3 shows that Old, in contrast, forks the first `multiply` thread after about four-fifths of second, eliminating the spawning advantage New has over it. So it seems that while the new kernel spawns threads more efficiently than the old, it occasionally delays the actual creation of threads because its scheduler does not de-schedule running threads quickly enough.

### 3.2 Time Sharing on the Two Kernels

The forking behavior the two kernels is not their only difference. Examining the initial time slices allocated to `main` and the `collector` in Figure 2 for example, it is easily seen New's scheduling algorithm occasionally gives scheduled threads execution time slices of several seconds. On-the-other-hand, Figures 1 and 3 show that Old's scheduler generally gives shorter, more uniform time slices to its `multiply` threads as well as `main` and the `collector`. New's scheduler does not always give longer time slices than Old's. Rather, the new scheduler uses a progressive algorithm that gradually increases the length of the execution time slices it allocates. New's algorithm is an experimental one which tracks the history and load on the machine in order to decide whether execution time slices can be extended. The algorithm is designed to schedule fewer context-switches than would occur using the old kernel's simpler algorithm which attempts to switch threads every 100 milliseconds.

The new kernel's allocation of progressively longer time-slices is vividly shown in Figure 4. During the first moments after the `multiply` threads are spawned, there is regular and frequent context-switching among all the threads, much like the behavior of the threads on Old. But, after some time, the new kernel's scheduler allows each thread to execute for longer and longer periods before being rescheduled. Because there is no significant competition from other threads during the computation's execution, the scheduler determines that these threads can run uninterrupted longer without being unfair to any of them.

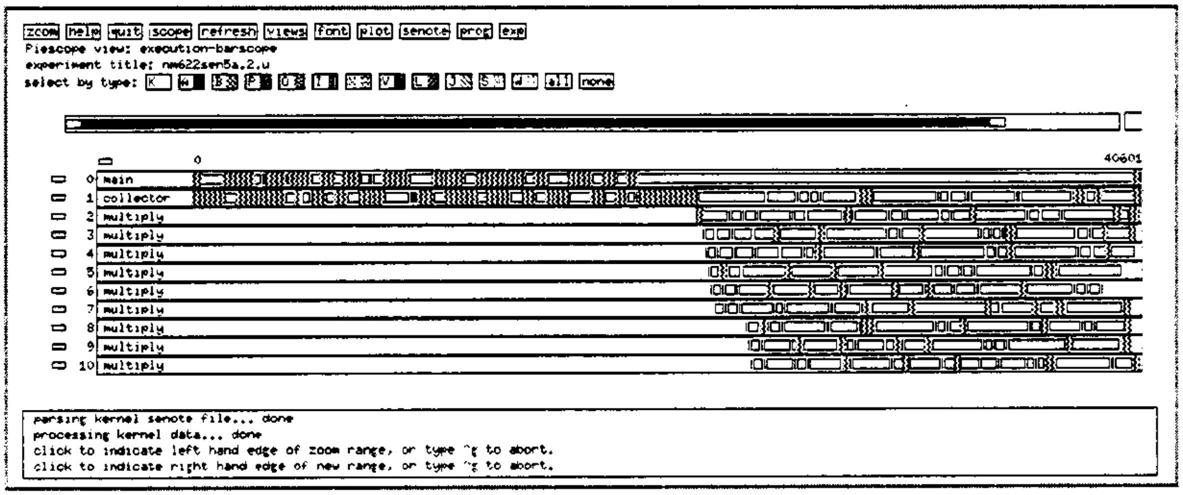
As with its forking performance, the way the new kernel time-shares running threads has some drawbacks. Examining Figure 4 again, it can be seen that just before the first `multiply` thread begins execution, New switches out the `collector` and does not run it again until nearly all the `multiply` children terminate. Alternately, Figure 3 shows that Old time shares the `collector` with the freshly spawned `multiply` threads. The new kernel's scheduler also seems to suffer frequent context-switch "flutter", where threads repeatedly switch to themselves. Earlier we noted that Old occasionally suffers from a kind of flutter occurring just after threads are forked off. There is a difference, however, between



```
Piescope view: execution-barscope
experiment title: nm422sen1.u
```

```
click to indicate left hand edge of zoom range, or type ^g to abort.
click to indicate right hand edge of new range, or type ^g to abort.
click to indicate left hand edge of zoom range, or type ^g to abort.
click to indicate right hand edge of new range, or type ^g to abort.
```

**Figure 5:** New Kernel: Zoom view of a smaller matrix multiplication



```
Piescope view: execution-barscope
experiment title: nm622sen5a.2.u
```

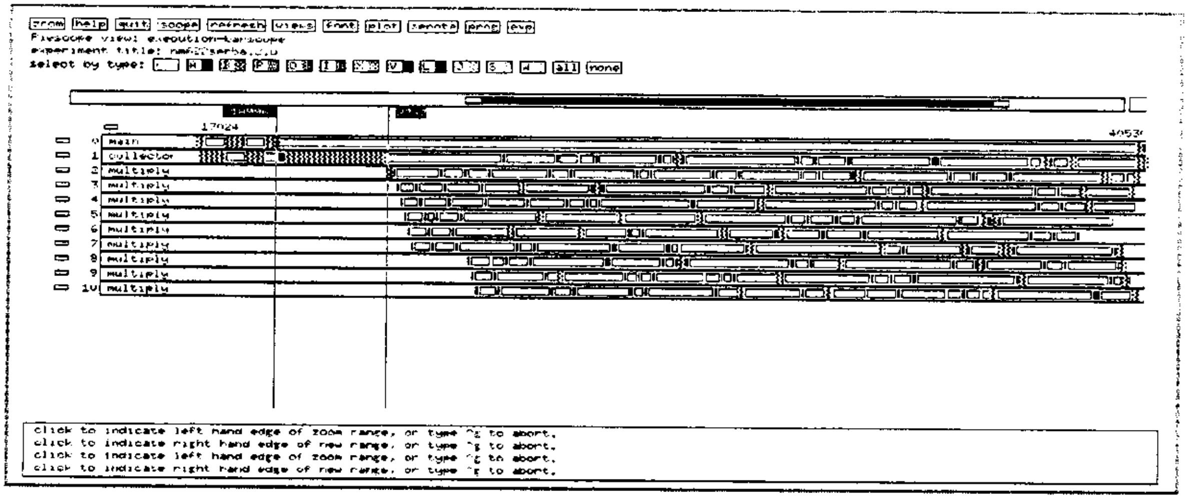
```
parsing kernel senote file... done
processing kernel data... done
click to indicate left hand edge of zoom range, or type ^g to abort.
click to indicate right hand edge of new range, or type ^g to abort.
```

**Figure 6:** Improved Kernel: Entire view of matrix multiplication

the flutter in New and in Old. In the case of Old, threads usually context-switch to themselves only for a short time after having been spawned. Threads running on the New however, context-switch to themselves in a more unpredictable, seemingly random fashion. Figure 5 shows an unambiguous example of this behavior occuring in matrix computation with only four `multiply` threads. In this figure, just before the `multiply` threads of a smaller matrix multiplier compution, are spawned, the `collector` thread repeatedly switches to itself; that is, the `collector` is repeatedly switched out and then immediately run again. Such behavior has been seen in other views of executions on New.

It is plain from these views that New's scheduler can be improved. In part because of the visual information we have presented, the designer of New's scheduler[2] identified the cause of threads occasionally switching out for long periods. It is primarily due to a scheduler "starvation" bug that causes

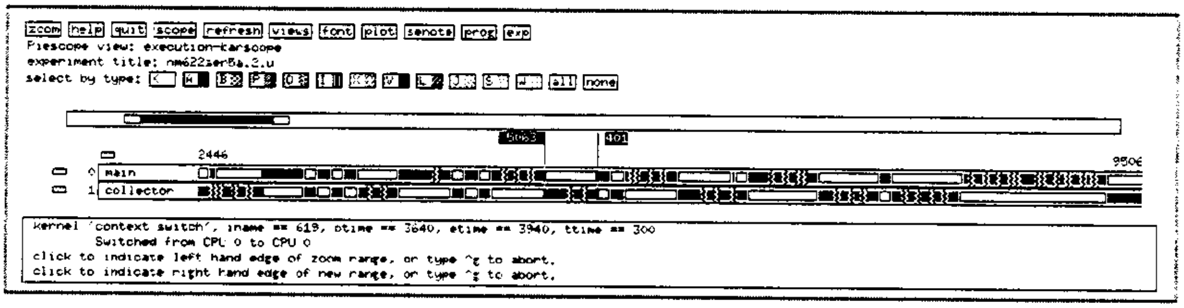
[2] David Black was the principal architect of the scheduling changes that differentiates CS3c, the new kernel, from XF29, the old.



```
Piescope view: execution-karscope
experiment title: nm622ser5a.2.u
```

```
click to indicate left hand edge of zoom range, or type ^g to abort.
click to indicate right hand edge of new range, or type ^g to abort.
click to indicate left hand edge of zoom range, or type ^g to abort.
click to indicate right hand edge of new range, or type ^g to abort.
```

**Figure 7:** Improved Kernel: Zoom view of matrix multiplication



```
Piescope view: execution-karscope
experiment title: nm622ser5a.2.u
```

```
kernel 'context switch', iname == 619, otime == 3640, etime == 3940, ttime == 300
        Switched from CPU 0 to CPU 0
click to indicate left hand edge of zoom range, or type ^g to abort.
click to indicate right hand edge of new range, or type ^g to abort.
```

**Figure 8:** Improved Kernel: Context-switch "flutter" in main and collector

threads to get stuck at low priorities. The **Improved** kernel is based on New but has the starvation bug fixed. Figures 6 and 7 show that, with the bug removed, the `collector` is not descheduled for an undesirably long period as it is on New. Using the same progressive scheduling algorithm as New, the improved kernel time shares the `collector` with the `multiply` threads. Figure 7 also shows that the starvation fix has partially alleviated another New performance problem. Recall that after spawning the first `multiply` thread, `main` does a join and switches out, waiting for its children to finish. Figure 7 shows that the amount of time given to the `collector` before the `multiply` is first allowed to run has been reduced from about six seconds to under three. Although Improved removes the starvation syndrome, it still suffers from a context-switch flutter problem. Figure 8 shows that, as in New, there are several cases of threads repeatedly switching to themselves. This problem is being addressed using the information obtained from MKM and PIE.

The unique marriage MKM and the visualization features of the PIE programming environment made this visual analysis of scheduling possible. Obviously, the synergy of the two yields an indespensable laboratory with which to run computational experiments. While PIE is discussed in literature cited earlier, the kernel monitor is a new development which needs to be described more fully.

## 4 Monitor Description

MKM has a different focus than monitoring work aimed at gathering general usage statistics. Kobayashi [5], for example, uses a kernel trace facility which records transitions between scheduable kernel states (eg: execution of file servers), unscheduable kernel states (eg: execution of interrupt handlers), and user states (eg: execution of application threads) to measure the effect of context-switching on cache hit-ratios. Used in conjunction with a hardware monitor, the trace facility yields interesting results on the impact of context-switching on cache performance. The only information it stores on individual threads, however, is whether they are user or kernel level threads. MKM is not like Kobayashi's trace facility where the context-switches of *all* measurable threads are recorded. Instead, MKM permits both user and system programmers to selectively *track* only those threads they purposively *choose.* MKM's goal is to give users a way to disambiguate the sources of performance problems in their individual applications and to give system programmers means to chronicle the behavior of services such as schedulers.

The MKM interface is compatible with abstractions already present in Mach. It supports simultaneous monitoring of independent computations so that users may selectively observe as many computations as they desire, collecting only the data about computations they are interested in, regardless of what else is running on the system.

### 4.1 General Architecture

Figure 9 depicts a case in which two independent non-communicating Mach tasks have created separate monitors. Each monitor is represented by a port in its parent task. Thus, the task that creates a monitor obtains rights to the port that represents the monitor; only tasks that possesses such rights can access the monitor. In our example, unless `task B` gives `task A` rights to the monitor created by `B`, `task A` cannot access it.

Figure 9 also shows non-intersecting sets of circular buffers allocated to each monitor for holding context-switch events. A buffer is assigned to each processor in order to eliminate contention between processors for buffers. When a thread context-switches, a software context-switch sensor detects which, if any, monitor is tracking the thread and writes an event to the appropriate buffer. Eventually, a task holding rights to a monitor will release those rights and terminate the monitor. This can be done either explicitly while the task is alive, or implicitly when the task terminates. In either case, the termination of a particular monitor is accomplished by the kernel.

### 4.2 Monitor system calls

There are several monitor system calls for accessing the kernel-monitor abstraction. All the calls take a monitor as a parameter. In order to create a monitor, `monitor_create()` is called with a requested kernel-monitor buffer size.

```
kern_return_t
monitor_create(my_task, new_monitor, requested_size)
    task_t              my_task;
    monitor_t           *new_monitor;
    int                 *requested_size;
```

This call returns a monitor to its caller and the total size of the monitor buffers that were allocated. The size of the buffer is needed by the caller so that he may allocate sufficient memory in the user space in order to hold the maximum amount of data the monitor may save before buffer overflows occur. The

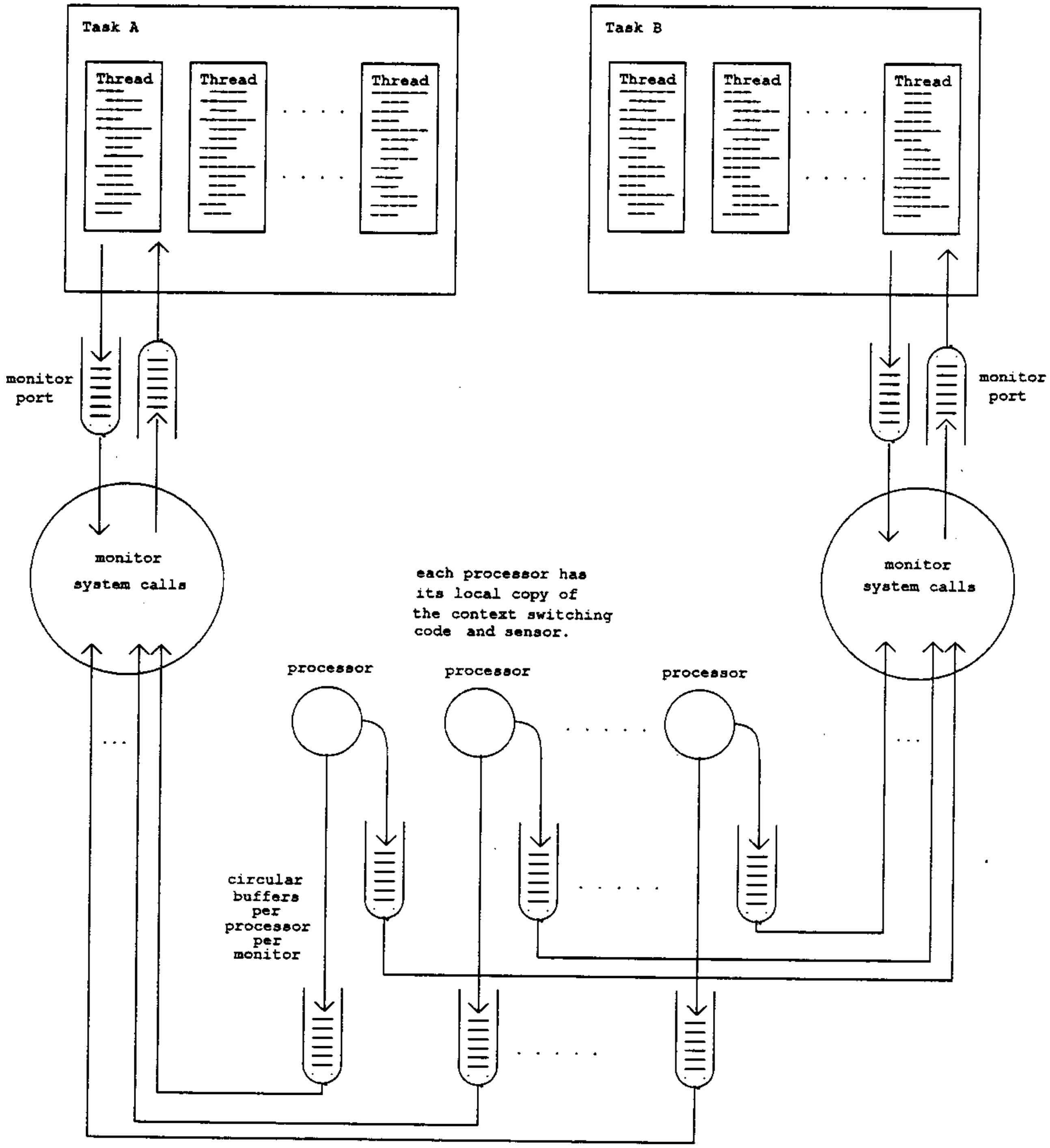


**Figure 9:** General Kernel Monitor Architecture

created monitor is returned in a suspended state and does not record any kernel events until `monitor_resume()` is called.

A monitor is terminated by calling the `monitor_terminate()`.

```
kern_return_t
monitor_terminate(this_monitor)
    monitor_t           this_monitor;
```

This call first breaks communication between the caller and the designated monitor (assuming the caller is the owner of the monitor). Then it places the monitor in a *shutdown* state so that the monitor state of any thread or other entity holding a reference to the monitor is set to a shutdown state. When an entity sees this state change, he removes his reference to the monitor and sets his state to a *null monitor* mode. Before the last reference removed, the holder removes all the dynamic data structures belonging to the monitor. Any attempt to access the terminated monitor after this call is made will return an invalid argument code.

When a monitor is created, it is suspended and no kernel events are recorded until `monitor_resume()` is called. The monitor can be suspended and resumed repeatedly by successive calls to `monitor_suspend()` and `monitor_resume()`. These calls give the user some flexibility in using his monitoring environment.

```
kern_return_t
monitor_suspend(this_monitor)
    monitor_t           this_monitor;

kern_return_t
monitor_resume(this_monitor)
    monitor_t           this_monitor;
```

To retrieve recorded kernel events, the user calls `monitor_read()`. This call reads all the relevant events that occurred since the first time `monitor_resume()` was called or since the last time `monitor_read()` was called.

```
kern_return_t
monitor_read(this_monitor, buffer, events_read)
    monitor_t           this_monitor;
    kern_mon_buffer_t   buffer;
    int                 events_read;
```

The call passes a monitor, a buffer for holding basic kernel events, and an integer into which the monitor returns the number of events that were read. The basic kernel event type is designed to be general enough to hold kernel events of different types. On a practical note, if `monitor_read()` is not called often enough, some kernel events might be lost if the kernel monitor buffers fill and overflow. Correcting this requires either creating a new monitor with larger buffers or making more frequent calls to `monitor_read()`.

As noted earlier, the current version of the monitor only observes selected context-switches. The `thread_monitor()` call enables the context-switches of selected threads to be observed. To disable a thread, `thread_unmonitor()` is called. Both calls are passed a monitor and a thread. The `thread_monitor()` call also requires an id which the user is responsible for keeping unique among threads.

```
kern_return_t
thread_monitor(this_monitor, unique_id, this_thread)
    monitor_t               this_monitor;
    int                     unique_id;
    thread_t                this_thread;

kern_return_t
thread_unmonitor(this_monitor, this_thread)
    monitor_t               this_monitor;
    thread_t                this_thread;
```

## 5 Monitor Performance Overhead

As was shown in Section 1, the kernel-monitor provides useful information about the scheduling behavior of a computation. But this information comes with a price of some overhead. The overhead is two tiered: *1*) sensor-firing overhead consisting of storing event information to special kernel buffers and *2*) constant overhead consisting of checking whether a switched thread is monitored. The measurements of both kinds of overhead are discussed below followed by a brief description of the measurement techniques.

| | |
|---|---|
| $T_{sensors-on}$ , $\overline{T}_{sensors-on}$ : | Inner loop time with sensors enabled; used in firing time measurement |
| $T_{sensors-off}$ , $\overline{T}_{sensors-off}$ : | Inner loop time with sensors disabled; used in firing time measurement |
| $T_{sensor-fire}$ , $\overline{T}_{sensor-fire}$ : | Sensor firing time |
| $T_{sensors-in}$ , $\overline{T}_{sensors-in}$ : | Inner loop time with sensors disable; used in constant overhead measurements |
| $T_{sensors-out}$ , $\overline{T}_{sensors-out}$ : | Inner loop time with sensors removed from code; used in constant overhead measurements |
| $T_{const-overhead}$ , $\overline{T}_{const-overhead}$ : | Constant overhead per context-switch |
| $N_{sensors-on}$ , $\overline{N}_{sensors-on}$ : | Inner loop iterations with sensors enabled |
| $N_{sensors-off}$ , $\overline{N}_{sensors-off}$ : | Inner loop iterations with sensors disabled |
| $N_{est}$ , $\overline{N}_{est}$ : | Estimated Inner loop iterations, sensors removed |
| $N_{loops}$ : | Number of iterations of outer loop; see experiment description |

**Table 1:** Variable Definitions

### 5.1 Results on the micro-Vax

Table 1 describes the statistics used in the overhead measurements. Tables 2 and 3 list the means, standard deviations and 99% confidence intervals for these statistics. The programs used to gather the data points for these statistics consisted of pairs of loops running on two concurrently executing processes. Described more fully in Section 5.5, the programs' loops were timed and, in the appropriate places, enabled for monitoring. Briefly, Tables 2 and 3 show that the average sensor firing time is about 211 microseconds and the constant context-switch overhead is about 17 microseconds.

| Statistic | $\mu$ | $\sigma$ | Confidence Intervals<br>99% |
|---|---|---|---|
| $T_{sensors-on}$ : | 3.453 secs | 0.126 secs | 3.453 ± 0.016 |
| $N_{sensors-on}$ : | 2414.4 | 14.0 | 2414.4 ± 1.8 |
| $T_{sensors-off}$ : | 2.943 secs | 0.102 secs | 2.943 ± 0.013 |
| $N_{sensors-off}$ : | 2412.3 | 14.0 | 2412.3 ± 1.8 |
| $T_{sensor-fire}$ : | 211 μ-secs | 50 μ-secs | 211 ± 6 |

**Table 2:** Context-Switch Sensor Firing Statistics of Loop iterations

| Statistic | $\mu$ | $\sigma$ | Confidence Intervals<br>99% |
|---|---|---|---|
| $T_{sensors-in}$ : | 2.907 secs | 0.088 secs | 2.907 ± 0.009 |
| $T_{sensors-out}$ : | 2.866 secs | 0.066 secs | 2.866 ± 0.006 |
| $N_{est}$ : | 2412.3 | 14.0 | 2412.3 ± 1.8 |
| $T_{const-overhead}$ : | 17 μ-secs | 3.5 μ-secs($\sigma_\mu$) | 17 ± 0.5 |

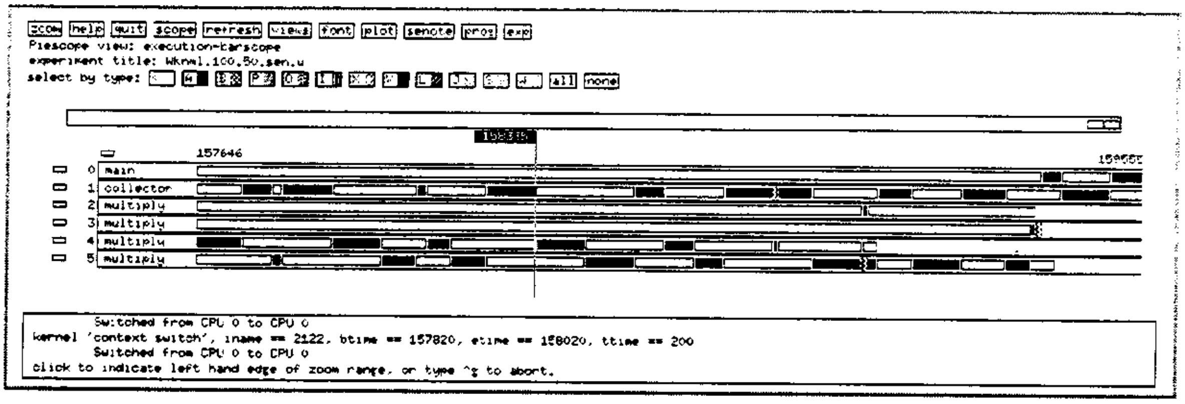
**Table 3:** Statistics for Measuring Static Overhead of Monitor Code

## 5.2 Examples of Overhead

What do these figures portend for the performance of typical computations? Let's assume that we are running computations on a kernel like the XF29 kernel described in Section 3.2. That is, the kernel regularly context-switches threads ten times a second. Because there are frequently miscellaneous deviations from a scheduler's regular behavior, let's also assume that the kernel must context-switch an additional five times a second. An unmonitored computation running under these circumstances would incur only the constant monitoring overhead each context-switch. Thus, if it normally takes a computation an hour to execute, the constant overhead of the monitor code would delay it by only $15 \times 3600 \times 0.000017 = 0.92$ seconds, or less than 0.03%. Now let's assume the same computation is running under the same conditions, but this time it's monitored. Realistically, if there are other computations running such as system utilities, not every context-switch will involve the computation. So let's say 12 out of 15 context-switches per second are monitored. Thus, if such a computation executed in an hour on an unmonitored kernel, it would require an extra $12 \times 3600 \times 0.000211 = 9.12$ seconds due to sensors firing and an additional $3 \times 3600 \times 0.000017 = 0.18$ seconds due to constant-overhead making up a total delay of about 9.3 seconds or only about 0.25 %.

These delay estimates are probably typical, but certainly not worse case. The experiments used to measure the sensor firing time and constant overhead gave results that are more like worse case scenarios. In the constant overhead experiments, for example, an average of over 2400 context-switches were estimated to have occurred in span of about 2.9 seconds (ie, almost one every millisecond). In these experiments, the constant overhead yielded a penalty of about 1.4%. When the sensors fired, the degradation rises to 14%. Since most computations do not context-switch every millisecond (and if they do, there is probably something awry), severe performance penalties are unlikely.

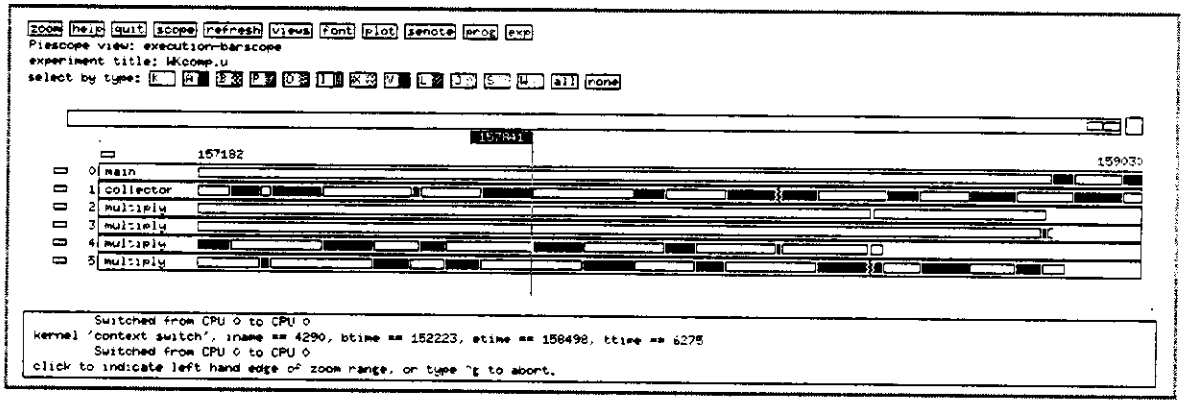
Although delay due to the context-switch sensors is the most regularly occurring intrusion of the monitor upon computations, the events detected by the sensors must occasionally be retrieved from their buffers and stored in user space, either in physical memory or to disk. This requires making periodic `monitor_read` calls each requiring about 1.3 milliseconds. The effect of `monitor_read` calls on the



```
Piescope view: execution-barscope
experiment title: Wkrnl.100.50.sen.u
```

```
        Switched from CPU 0 to CPU 0
kernel 'context switch', iname == 2122, btime == 157820, etime == 158020, ttime == 200
        Switched from CPU 0 to CPU 0
click to indicate left hand edge of zoom range, or type ^g to abort.
```

**Figure 10:** View of the Finish of another Matrix Multiplication (Old Kernel)



```
Piescope view: execution-barscope
experiment title: WKcomp.u
```

```
        Switched from CPU 0 to CPU 0
kernel 'context switch', iname == 4290, btime == 152223, etime == 158498, ttime == 6275
        Switched from CPU 0 to CPU 0
click to indicate left hand edge of zoom range, or type ^g to abort.
```

**Figure 11:** Compensated View of the Finish of the Same Matrix Multiplication (Old Kernel)

computations discussed in this paper is negligible because if it were the case that the `collector` never called the routine, the `collector` would have merely busy waited. The length of time the `collector` executed in these computations was not dependent upon what it was doing, but rather upon how long the `mulitply` threads executed. Thus the delays due to sensor firings were the dominant intrusion upon the computations. PIE provides means to compensate for the intrusion of the sensor firings on micro-Vaxes and other uniprocessors. The compensation algorithm is a fairly simple one consisting of adjusting the timestamp of each context-switch in accordance with how many have switches occured before it. The algorithm compensates for the delays by subtracting from each time stamp a quantity corresponding to the sum of all the sensor firing times of previous context-switch events. The current algorithm does not attempt to modify the total number of context-switches counted in a computations even though it is reasonable to expect that shorter computations should suffer fewer context switches. Figure 10, for example, shows the tail end of a matrix multiplication. Figure 11 shows the same computation after compensation by PIE. As can be seen, the computations appear identical except that their termination times and the metering line point to their differing execution times. Fortunately, the compensation is often unnecessary. Even without compensating for the sensor firings, the execution times of computations executing on micro-Vaxes with and without kernel monitoring are generally with one percent of each other.

### 5.3 Statistical Calculations of Overhead

The mean firing time is found by:

$$\overline{T}_{sensors\text{-}fire} = \frac{\overline{T}_{on}}{\overline{N}_{sensors\text{-}on}} - \frac{\overline{T}_{off}}{\overline{N}_{sensors\text{-}off}}$$

Because $\overline{N}_{sensors\text{-}off}$ could not be measured, $T_{sensors\text{-}fire}$ is approximated by:

$$\overline{T}_{sensors\text{-}fire} \approx \frac{\overline{T}_{sensors\text{-}on} - \overline{T}_{sensors\text{-}off}}{\overline{N}_{sensors\text{-}on}}$$

It turns out this approximation for the µVax does not alter the estimate of the sensor firing time since its effects fall outside the accuracy of the measurements. The mean constant overhead incurred by the kernel because of the monitor code is approximated by:

$$\overline{T}_{overhead} \approx \frac{\overline{T}_{sensors\text{-}in} - \overline{T}_{sensors\text{-}out}}{\overline{N}_{est}}$$

$\overline{N}_{est}$ is presumed equal to $\overline{N}_{sensors\text{-}off}$

The first columns in Tables 2 and 3 list the statistics of interest. Excepting a few notable cases, the standard deviation and confidence intervals of each statistic were obtained using ordinary techniques. Comments on the exceptional cases are discussed in section 5.4. It should be noted that $T_{sensors\text{-}off}$ and $T_{sensors\text{-}in}$ are measurements of ostensibly equivalent phenomena, namely, the loop times with monitor sensors off. The two statistics were measured under different conditions; $T_{sensors\text{-}off}$ was measured during tests run overnight with the µVax connected to the network. However, because it had been expected that the difference between the means of $\overline{T}_{sensors\text{-}in}$ and $\overline{T}_{sensors\text{-}out}$ would be small, it was desirable to ensure that no other user could load the machine so that the standard deviations of the two statistics could be kept small. To ensure this, the µVax was disconnected from the network when $T_{sensors\text{-}in}$ and $T_{sensors\text{-}out}$ were measured resulting in $T_{sensors\text{-}in}$ being less than $T_{sensors\text{-}off}$. Actually, disconnecting the µVax caused some network queries from the system to timeout, producing periodic outliers in the collection of time-stamps. This periodicity was so regular and the outliers so extreme, however, that those data points were easily detected and discounted.

### 5.4 Data Analysis

The assumptions and calculations used to generate the numbers in Tables 2 and 3 are not of central importance in this report. For those not concerned with the specifics, this section can be skipped.

Most of the data listed the two tables was calculated using ordinary statistical formulas. The exceptions are $N_{sensors-off}$ and $\sigma_{sensors-fire}$ and $\sigma_{const-overhead}$, the standard deviations of the sensor firing time and constant overhead per context-switch. The mean number of context-switches occurring when the sensors were off is an estimate based on three considerations:

1. Aside from the firing of the sensors, the loops are identical.
2. The loop index limit was 1200, limiting the number of intentionally *induced* context-switches to 2400. The remaining switches resulted from other scheduling behavior.
3. Assuming this "other" scheduling behavior is roughly periodic, there will be a slightly lower mean number of context-switches when the sensors are off than when the sensors are on because the loop times are shorter.

The problem of not being able to get a count of loop context-switches in the unmonitored tests, required an approximation in the calculation of the standard deviation (and, thus, confidence interval) of the estimated sensor firing time.

The variance ${\sigma_{sensors-fire}}^2$ is approximated by:

$$ {\sigma_{sensors-fire}}^2 \approx \left(\frac{1}{N_{loops}-1}\right)\sum_{i=1}^{I} \left(\frac{T_{sensors-on_i} - \overline{T}_{sensors-off}}{N_{sensors-on_i}} - \overline{T}_{sensors-fire}\right)^2 $$

Ideally, of course, it would be desirable to have a unique $T_{sensors-off_i}$ for each $T_{sensors-on_i}$ and $N_{sensors-on_i}$. But, the inability to retrieve an individual context-switch count with each unmonitored loop time (eg. $T_{sensors-off}$) necessitates using an approximate method to calculate the variance of $T_{sensors-fire}$. If the standard deviations for $T_{sensors-on}$ and $T_{sensors-off}$ were large and different, then it would not be likely that the approximation reflects a reasonable estimate of a hypothetical sum containing unique $T_{sensors-off_i}$. But, because the standard deviations for $T_{sensors-on}$ and $T_{sensors-off}$ are small and close, this approximation yields a viable estimate for ${\sigma_{sensors-fire}}^2$. An estimate for ${\sigma_{sensors-out}}^2$ is obtained in a similar way. The value for ${\bar{\sigma}_{const-overhead}}^2$ is then computed using ordinary techniques.

### 5.5 Measurement Methodology

Measurements the constant overhead of the kernel-monitor as well as sensor firing times were done using programs modeled after the two outlined in Figures 12. The programs were executed on a μVax booted in multi-user mode because certain system services they used were unavailable in single-user mode. In the experiments to determine the constant overhead, the μVax's ethernet cable disconnected in order to reduce competition for the processor. Disconnection from the ethernet caused some servers to cycle longer, but these periods were predictable and easily discounted.

### 5.5.1 Measuring Sensor Firing Times

Measurement of the firing times was done using a program like that shown in Figure 12. Implicit in its design is that identical loops will, on mean, have the same number of context switches. As can be seen, one instance of the loop contained in `send_receive_messages_and_time_them()`, was monitored and timed, the other was just timed. By subtracting the time difference and dividing by the number of context-switches, an estimate of the sensor-firing times could be obtained.

```
main()
{

  declarations;

  if (fork()) {
    while (child_done != CHILD_DONE){
        receive_child_mesg();
        send_child_mesg();
    }
  } else {
    create_and_start_monitor();
    for (i= 0; i < SOME_CONSTANT; i ++) {
        enable_thread_for_monitoring();
            send_receive_messages_and_time_them();
        disable_thread_for_monitoring();
        get_and_count_context_switches();

        send_receive_messages_and_time_them();
    }
    terminate_monitoring();
  }
}
```

**Figure 12:** Code for measuring kernel sensor firing times

### 5.5.2 Measuring Constant Overhead

The second measurement program shown in Figure 13 was run on two versions of the "same" kernel. The only difference between the two kernels is that one contained sensors and the other did not. The loop-limits were the same as in the first program so that it could be assumed that the number of context-switches were the same. By subtracting the shorter time from the longer and dividing by the expected number of context-switches, an estimate of the constant overhead was obtained.

## 6 Work Pending

This paper has shown how the marriage of a context-switch kernel-monitor and the visualization graphics of a programming environment yields penetrating glimpses into scheduler performance. The example showed that visually ordering performance data is a powerful technique for studying experiment results. The PIEscope views chronicled the context-switch behavior, clearly illuminating several salient differences between the two schedulers. They provided an intuitively appealing representation of context-switching performance, permitting a designer to quickly analyze the performance of scheduling policies for both sequential and parallel machines.

Although MKM and PIE are a powerful synergy, they both can be improved and extended in a number of ways. For example, the current version of MKM only monitors context-switches. Other monitoring work

```
main()
{

  declarations;

  if (fork()) {
    while (child_done != CHILD_DONE){
        receive_child_mesg();
        send_child_mesg();
    }
  } else {
    for (i= 0; i < SOME_CONSTANT; i ++) {
       send_receive_messages_and_time_them();
    }
  }
}
```

**Figure 13:** Code for measuring kernel monitor overhead

consists of tracing file system management [9] or inter-thread actions as in METRIC [6] and DPM [8]. Both METRIC and DPM include a monitoring facility for distributed programs which traces actions between the distributed processes of a computation. The monitoring provides data which, when analyzed, not only yields basic communication statistics but also rudimentary information about execution histories.

## 6.1 Monitor Enhancements

As just noted in our discussion of alternative monitoring, there are other interesting kernel-level actions that can be monitored and visualized besides context-switches. Being able to visualize disk accesses or page fault behavior, for example, would be useful for a designer investigating various memory management policies. One of our long-range goals is to develop a work-station computational laboratory based on a kernel-monitor which observes a sufficient variety of actions and a visualization environment which offers flexible views of performance data. In addition to being able to measure sequential and tightly-coupled parallel systems, such a kernel-monitor would have to be sensitive to actions peculiar to distributed environments as well. Although MKM is extensible to a distributed environment, it is not clear how the additional behavior should be visually represented by PIE. Although the actions measured by MKM for distributed programs will be similar to those monitored by systems like DPM, the data describing them will be targeted for visualization formats like those provided by PIE.

## 6.2 PIE Enhancements

From the beginning, PIE has included a user-level run-time monitoring facility as a part of its environment. It was recognized early on, however, that in order to disambiguate user-level performance data, kernel-level information is needed. The addition of kernel-monitoring, however, has been only one of several improvements made to PIE since its inception. For example, work is continuously underway to enhance the visualization formats and performance analysis tools offered by PIE in order to speed up the performance improvement process. IPS [7], for example, includes automatic assistance for peformance analysis. There is, of course, work on other aspects of performance monitoring such as making certain that the data collected about a computation is an accurate portrait of what it did. It is important to understand how a computation is perturbed by a monitor. Because total elimination of monitor perturbation is impractical, work on perturbation has centered on measuring monitor overhead instead of eliminating it. Although this work will undoubtably improve performance analysis by PIE, the synergy of the kernel performance monitor and the visualization tools already available in PIE yields a powerful performance analysis tool for a variety of applications.

## 7 Conclusions

The contribution of this research is not only proving that an integrated user-level/kernel-level visualization system is conceptually and practically feasible, but that such system could be fully implemented and evaluated. MKM as integrated with PIE is fully operational on a number of types of workstations and parallel computers. In fact, we are fully expecting that MKM/PIE will be operational on any machine running Mach.

MKM is providing valuable insights into the design of the Mach scheduler as well as in the design of concurrent and parallel applications. The evaluation of MKM concluded that the intrusiveness is predictable in well defined limits and for a certain set of applications the overhead incurred is either acceptable or compensable. This result serves to highlight the attractiveness of the software based monitoring used in MKM when contrasted with potentially less portable, less intrusive hardware based monitoring techniques. Further research is needed and planned in the above area as well as in how to use MKM as a basis for a general purpose parallel and distributed functional and performance debugger.

## References

[1] M. Accetta, R. Baron, W. Bolosky, D. Bolub, R. Rashid, A. Tevanian, M. Young.
Mach: A New Kernel Foundation for Unix Development.
In *Proceedings of USENIX 1986 Summer Conference*, pages 93 - 112. Computer Science Department, Carnegie Mellon University, Summer, 1986.

[2] R. Fitzgerald, R. F. Rashid.
The Integration of Virtual Memory Management and Interprocess Communication in Accent.
*ACM Transactions on Computer Systems* 4(2), May , 1986.

[3] Francesco Gregoretti, Zary Segall.
*Programming for Observability Support in a Parallel Programming Environment.*
Technical Report CMU-CS-85-176, Computer Science Department, Carnegie Mellon University, November, 1985.

[4] M. B. Jones, R. F. Rashid, M. Thompson.
MatchMaker: An Interprocess Specification Language.
In ACM (editor), *ACM Conference on Principles of Programming Languages.* Computer Science Department, January, 1985.

[5] Makoto Kobayashi.
An Empirical Study of Task Switching Locality in MVS.
*IEEE Transactions on Computers* C-35(8):720 - 731, August, 1986.

[6] Gene McDaniel.
METRIC: a kernel instrumentation system for distributed environments.
In *Proceedings of the Sixth ACM Symposium on Operating System Principles*, pages 93 - 99. Xerox Palo Alto Research Center (PARC), November, 1977.

[7] Baron P. Miller, Cui-Quing Yang.
IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs.
In *Seventh International Conference on Distributed Computing Systems*, pages 482 - 489. Computer Sciences Department, University of Wisconsin, September, 1987.

[8] B.P. Miller.
DPM: A Measurement System for Distributed Programs.
*IEEE Transactions on Computers* 37(2):243 - 247, February, 1988.

[9] J.K. Ousterhout, H. Da Costa, D. Harrison, J.A. Kunze, M. Kupfer and J.G. Thompson.
A Trace-Driven Analysis of the UNIX 4.2 BSD File System.
In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 15 - 24. Computer Science Division, Electrical Engineering and Computer Sciences, UC Berkeley, December, 1985.

[10] Richard Rashid, Avadis Tevanian, Michael Young, David Young, Robert Baron, David Black, William Bolosky and Jonathan Chew.
Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures.
*IEEE Transactions on Computers* 37(8), August , 1988.

[11] Zary Segall, Larry Rudolph.
*PIE - A Programming and Instrumentation Environment for Parallel Processing.*
Technical Report CMU-CS-85-128, Computer Science Department, Carnegie Mellon University, April, 1985.

[12] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, R. Baron.
The Duality of Memory and communication in the Implementation of a Multiprocessor Operating System.
In *Proceedings of the Symposium on Operating System Principles.* School of Computer Science, Carnegie Mellon University, November, 1987.