3P−5

A Lock Monitor for a Shared Memory Multiprocessor Operating System

Joe Uemura, Katsuhide Takahashi, Takashi Kan
Mitsubishi Electric Corporation
Computer & Information Systems Laboratory

1. Introduction

Harnessing the power provided by shared-memory multiprocessors requires an operating system to service multiple simultaneous user requests. Operating systems with this capability are said to support symmetric multiprocessing execution. Symmetric execution demands the operating system to synchronize accesses to its data structures. This synchronization is often implemented using multiprocessor locks. Contention for locks and the delays associated with lock events are potential performance bottlenecks. Tools which help gain insight into the execution characteristics of locks play an important role in the analysis and tuning of symmetric multiprocessor operating systems.

In this paper, we present a facility implemented to monitor multiprocessor locks. The target system of this implementation is a RISC based shared-memory multiprocessor running the OSF/1[1] operating system. Although the implementation takes into consideration features of OSF/1, we believe that most of the descriptions about our work can be applied to monitoring locks on other symmetric operating systems as well. We are currently using this monitor to analyse the OSF/1 kernel.

2. Design Considerations

2.1. Lock events

The lock attributes which we chose to investigate were *lock activity, contention, latency,* and *granularity.* Previous lock analysis work, including our own [Uemura91], and the one of Campbell et.al. [Campbell91] also focus on a similar set of attributes. An overview of these attributes follows:

1) ***lock activity:*** the number of times a lock is accessed during a given load, giving an estimate of the overhead incurred acquiring and releasing locks.
2) ***lock contention:*** taken every time a lock cannot be acquired. This information is of particular interest for performance tuning; a lock with a high ratio of contention/acquisitions indicate a potential performance problem.
3) ***lock latency***: the amount of time spent waiting when a lock contention occurs. With spin locks, the number of spins is used; for blocking locks, the elapsed wall clock time is recorded. Locks with high latency measurements also suggest the existence of potential performance bottlenecks.
4) ***lock granularity:*** a fine grain lock is one which protects a small amount of shared data, whereas a coarse grain lock protects several data structures with a single lock. A fine grain lock will often have a higher activity rate, but lower latency. On the other hand, a coarse grain lock might require fewer accesses, thus minimizing access overhead.

[1]OSF/1 is a trademark of the Open Software Foundation, Inc.

Performance analysis and tuning requires finding the right tradeoffs among these lock attributes.

2.2. Global Measurements

As one of our goals, we were interested in characterizing the overall behavior of the system. Particularly, we focused on collecting data meaningful for global analysis. To address this concern, we decided to collect data by *lock classes* instead of by lock object. We grouped locks into *lock classes* according to the data type associated with the lock. We used these classes when monitoring lock events. A global table was used to hold the statistics associated with the lock classes. A pointer to the corresponding entry in this table was added to each lock object. This greatly simplified gathering the data for analysis since a single entry is allocated for each lock class. This entry is used to hold the collected statistics for all locks in the class, and it persists even after the lock objects belonging to the class have been deallocated. This table is statically allocated and thus resides at a well-known location in the kernel address space. Collecting the monitored data by user programs simply involves reading this table. Furthermore, the data associated with the lock classes presents a summary of the locks execution patterns, particularly useful for global analysis. Figure 1 illustrates lock classes.

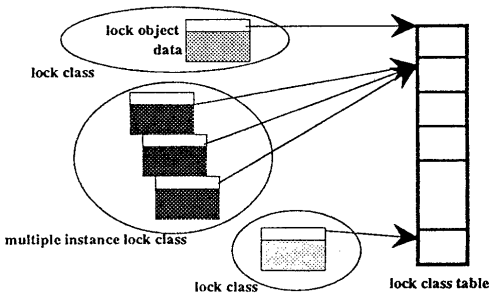

Figure 1.
Lock class implementation.

2.3. Perturbation Effects

Software monitoring tools, though flexible in nature, are intrusive, and often introduce perturbation effects which affect the results of the computation being monitored. Minimizing these effects is important, not only for approximating correct execution, but also for wide acceptance of the tool. Ideally, the perturbation effects should be so minimal that users would constantly monitor the system regardless of the overhead produced by monitoring. We offer several levels of control which the user can choose to exercise.

1) ***configuration parameter***: at build time, the kernel developer can specify if the monitor code should be

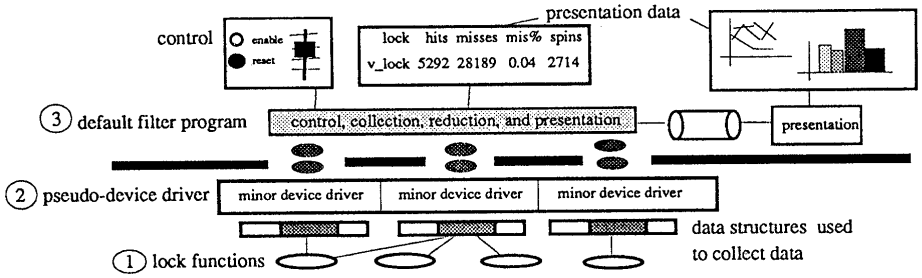

Figure 3.
Main components of the lock monitor facility are (1) lock functions, (2) pseudo device driver and (3) default filter program.

compiled with the rest of the kernel. This is accomplished by simply toggling a built time option.

2) ***user control***: for kernels built with the lock monitor, users can enable/disable monitoring at execution time giving them control of when to incur the monitoring costs. Options to enable different types of monitoring are also available. Users can balance the amount of information collected against the cost of obtaining it.

3) ***sampling rates:*** to support finer control over the amount of perturbation, a sampling rate parameter is provided. Users can, at run time, specify the sampling rates of the events being recorded. A sample rate of ten means that one out of ten events is actually monitored. Figure 2 illustrates how different rates help minimize the performance degradation due to monitoring.

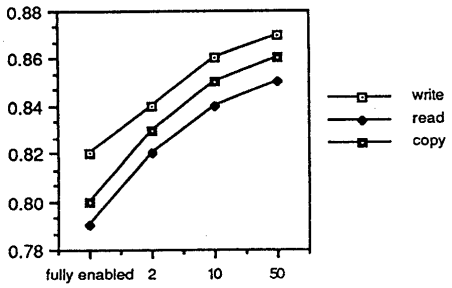

Figure 2.
The horizontal axis displays the sample rates. A fully enabled monitor has a sample rate of one (every event is monitored). The vertical axis displays the relative performance compared to running the same load with the monitor disabled. We ran a file I/O program which reads, writes, and copy a file with various sampling rates. The results show that higher sampling rates can reduce the pertubation effects by several percentage points.

3. Monitor Structure

The overall structure of the lock monitor facility is shown in figure 3. The major components are:

• ***modified lock functions***: existing functions have been modified to acquire lock data. These functions store the data in a set of buffers which are read and updated by the pseudo device driver.

• ***pseudo device driver***: a pseudo device driver is used as the interface between user programs and the monitor. The driver also allows users to control how monitoring takes place. User programs, using the driver, can enable/disable monitoring, reset internal buffers, and specify various thresholds. The pseudo driver has several minor devices. Extracting data and controlling the monitor is accomplished by accessing the associated minor device. For example, we currently associated one minor device with spin locks, and another with blocking locks.

• ***default filter program***: this program runs in user mode and is used to control the monitor, and collect, reduce, and present data. Various forms of data reduction are supported such as sorting and filtering of uninteresting data. Other data manipulations are also performed at this level such as computing ratios between various elements of the monitored data. This program can also be used to pass data to other more specialized filters. These filters can present data in different forms such as graphics outputs, or in even more specialized views of the data.

4. Conclusion

The events and delays associated with locks used to guarantee correct execution of multiprocessor operating systems play an important role in the analysis and tuning of these systems. Tools which help developers gain insight into the execution characteristics of locks are highly helpful. Addressing this need, we have developed a lock monitor tool. Making this tool fulfill its potential required us to design it with several considerations in mind. To present a summary of the locks execution pattern, we classified the existing locks, and used these classes to summarize the collected data. Minimizing pertubation effects required us to design an implementation with several levels of user control. Code portability requirements were also met by localizing changes. Finally, flexible and expandable user interfaces were implemented.

Several developers are currently using the monitor. Besides analysing the various interactions of locks, other current usages include evaluating the overhead incurred acquiring and releasing locks. Under certain loads, this overhead seems higher than originally expected. Several developers are using the monitor to find the reasons why this occurs.

References

[Accetta86] M. Acceta, R. Baron, D. Gollub, R. Rashid, A. Tevanian, M. Young. MACH: A New Kernel Foundation for UNIX Development. *Proc. Summer 1986 Usenix Technical Conference.* Usenix Association, 1986.

[Campbell91] M. Campbell, R. Holt, J. Slice. Lock Granularity Tuning Mechanisms in SVR4/MP. *Proc. of 2nd. Distributed & Multiprocessor Systems Workshop.* Usenix Association, 1991.

[Uemura91] J. Uemura, T. Sakakura, T. Kan. An Empirical Investigation of Multiprocessor Synchronization Mechanisms in the MACH Kernel. *Proc. of SWooP Ohnuma 91.* 1991.