513/CW/IN/82

1999 NASA/ASEE SUMMER FACULTY FELLOWSHIP PROGRAM

JOHN F. KENNEDY SPACE CENTER
UNIVERSITY OF CENTRAL FLORIDA

PERFORMANCE MONITORING OF DISTRIBUTED DATA PROCESSING SYSTEMS

Anand K. Ojha
Associate Professor
Department of Engineering Technology
University of Arkansas at Little Rock
KSC Colleague: Craig Jacobson, NN-J2, KSC

ABSTRACT

Test and checkout systems are essential components in ensuring safety and reliability of aircraft and related systems for space missions. A variety of systems, developed over several years, are in use at the NASA/KSC. Many of these systems are configured as distributed data processing systems with the functionality spread over several multiprocessor nodes interconnected through networks. To be cost-effective, a system should take the least amount of resource and perform a given testing task in the least amount of time. There are two aspects of performance evaluation: monitoring and benchmarking. While monitoring is valuable to system administrators in operating and maintaining, benchmarking is important in designing and upgrading computer-based systems. These two aspects of performance evaluation are the foci of this project.

This paper first discusses various issues related to software, hardware, and hybrid performance monitoring as applicable to distributed systems, and specifically to the TCMS (Test Control and Monitoring System.) Next, a comparison of several probing instructions are made to show that the hybrid monitoring technique developed by the NIST (National Institutes for Standards and Technology) is the least intrusive and takes only one-fourth of the time taken by software monitoring probes. In the rest of the paper, issues related to benchmarking a distributed system have been discussed and finally a prescription for developing a micro-benchmark for the TCMS has been provided.

PERFORMANCE MONITORING OF DISTRIBUTED DATA PROCESSING SYSTEMS

Anand K. Ojha

1. INTRODUCTION

The Kennedy Space Center has several test and checkout systems to ensure safety and reliability of spacecraft and equipment for space missions. Some of the commonly used systems are: CITE (Cargo Interface Test Equipment), PPCU (Partial Payload Checkout Unit), TCMS, (Test, Control, and Monitor System), and CMU (Control and Monitor Unit.) These test and checkout systems are essentially distributed data processing systems with the computational functionality distributed over several processors or clusters of processor interconnected through custom or commodity network. Several TCMS units are in use, and because of its interesting and open architecture, the performance of this system is the focus of this project.

System performance is one of the key concerns to designers, administrators, and users of a computer system. Information about system performance can be obtained by two methods: system monitoring and system benchmarking. Monitoring tools help in the operation and maintenance of an existing system and are indispensable tools for computer system administrators. These monitoring tools provide valuable information such as system utilization and workload statistics to help efficiently manage and upgrade the system. Benchmarking, on the other hand, applies a fixed workload to a system to determine latency, throughput, and resource utilization and is helpful in comparing the performance of two different systems under identical workload. While there are several de-facto benchmarking test-suites for CPU, no comprehensive test-suite has achieved this status in the arena of I/O benchmarking. This has been primarily due to the inherently diverse nature of the I/O systems, and has led to the development of system-specific and in-house benchmarking test-suites. A similar undertaking seems to be a viable solution for the TCMS as well. This paper covers the issues in monitoring and benchmarking of distributed systems, especially as they apply to the TCMS.

The next section provides a brief overview of the TCMS. Performance monitoring choices and issues are discussed in Section 3, and the quantitative results presented in Section 4 indicate that hybrid monitoring takes only one-fourth of the time compared to software monitoring. Section 5 provides a synopsis of existing benchmarks, analyzes several benchmarking options, and outlines the details of a comprehensive micro-benchmarking scheme for the TCMS to test not only the CPU but also I/O and operating system.

2. BACKGROUND ON TCMS

TCMS has evolved from PPCU to support the activities at SSPF (Space Station Processing Facility), launch pads, SLF (Shuttle Landing Facility), and the OPF (Orbiter Processing Facility.) It is based on UNIX/X-Windows, and is built around SGI Origin 200 multi-processor computers with the functionality spread over the internal processors and Motorola's MVME167 cards linked though Ethernet. TCMS is essentially a distributed data acquisition, processing, and control system, and supports a mix of inputs including direct discrete and analog measurements and various types of serial and telemetry formats. The TCMS was originally built around Nighthawk computer system, but to benefit from the advances in the technology, the Nighthawks

have been replaced by the Origin 200 containing R10000 dual processors. A detailed schematic of the TCMS is shown in Figure 1 (attached as the end of this paper.) Since several TCMS units are now in use, this is the system chosen for this project.

3. ISSUES IN PERFOMRANCE MONITORING

There are three approaches for implementing a performance monitor: software, hardware, and hybrid. These three techniques and their pros and cons are discussed in this section.

3.1. Software Monitoring

Software monitoring is also known as software instrumentation. In the most basic software monitoring scheme, execution time of a program module is obtained by inserting system-timer calls at appropriate points in the application program. The following three such system calls of interest are available in Unix:

- `time(NULL)`: Wall clock time in seconds since 1/1/1970
- `gettimeofday(&tv, NULL);`

```
Time_seconds = tv.tv_sec;          // Integer number of seconds since 1/1/70
Time_microseconds = tv.tv_usec;    // Fraction seconds in microseconds
                                   // since 1/1/70
```

- `Clock()`: The system call returns the CPU time in ms or μs depending on the implementation. One needs to be very careful in using this system call, because it provides information only about CPU utilization and not the wall clock time. Instructions, such as `sleep(int seconds)`, that make the processor wait for a specified amount of seconds are not accounted for by the `clock()` instruction.

Software monitoring has three obvious limitations. First, the 1-μs resolution of the system timer is not very good for present-day computers whose clock period is on the order of 1-ns. To obtain results with satisfactory resolution, the execution time of a program module (or for that matter of any individual instruction) is determined by executing the module a large number of times and then obtaining the average execution time; clearly, a time-consuming process.

The second problem is a result of what is known as the "probing effect" in which the inserted monitoring instructions add to the execution time of the code under measurement. The probing effect is sometimes tolerable for a standalone uniprocessor system. However, for a multi-user-multi-tasking system that has a node with multiprocessors (such as the Origin 200 used in TCMS), the probing effect may significantly degrade the system performance due to loss of synchronization of the cooperating or multi-threading processes. Probing effects are particularly detrimental to parallel programming environments.

Thirdly, the software approach provides only a partial monitoring capability, because it cannot efficiently monitor cache hits, or events caused by external conditions, such as exceptions, interrupts, etc.

3.2. Hardware Monitoring

Hardware monitoring requires an external dedicated hardware in the form of either a plug-in card or a dedicated box to monitor the signals on the bus at a processing node [1]. The dedicated hardware contains its own local memory and functions like a logic analyzer in that it

records the events of interest based on the signals captured from the address, data, and control bus of the system being monitored. Every event of interest is encoded as a unique bit pattern called an *event token* that uniquely specifies *what* external event occurred and *where* it occurred. An event token is like an ID number of an event. For every event of interest, an event record is created that contains the event token and its timestamp so that the samples could be correlated in time. Event records are first stored in the local memory of the monitoring hardware, and later downloaded on to a host for trace-generation and post-analysis.

On the positive side, hardware monitoring is least intrusive, and in addition to interrupts, it can also monitor designated events during program-execution. However, hardware monitoring is impractical for systems employing virtual memory because while the compiler generates virtual addresses, the monitoring hardware must be designed to be triggered by the physical addresses and signals. This knowledge of this virtual to physical address translation mechanism makes the design of hardware monitoring systems impractical for virtual memory.

3.3. Hybrid Monitoring

Hybrid monitoring combines the desirable features of both software and hardware techniques. The embedded software probes log the events caused by the program itself, while external events are recorded by the hardware. It is generally agreed that hardware monitoring provides an efficient solution to monitoring distributed systems.

Since there were numerous ways to design a hardware monitor, each research group initially came up with its own custom hardware monitor [2]. Later, in an effort to standardize hybrid monitoring, the National Institutes of Standards and Technology (NIST) developed a plug-in board known as the Multikron Interface Board (MIB) for PCI, S-Bus, and VME bus [3, 4]. Universities and research organizations can obtain these MIBs on loan from the NIST. Otherwise, these can be bought from the NIST for about $750 at the time of the writing of this paper in 1999. Salient features of the MIB are described in the following paragraph.

The MIB is a memory-mapped device and can monitor up to eight CPUs sharing a bus. It has a 56-bit timestamp counter with a resolution of 100ns. In contrast, the best resolution obtainable from software system calls is only 1 μs. Two types of samples can be collected by the MIB: *trace* sample and *resource* sample. A 20-byte trace sample contains CPU ID, process ID, user-written identifier, and the timestamp. The 84-byte resource sample, in addition, contains the contents of the16 32-bit counters that can be started, incremented, and stopped under hardware or software control. Hence, these counters can be used to count the occurrence of events such as cache hits, duration of memory bus cycles, etc. The user simply writes an integer identifier value through a single assignment (memory write) statement; all other data are non-intrusively provided or captured by the board itself. The MIB has 16 MB of on-board RAM, and so it can store 800K trace samples in its local memory before needing a download to a host for post-analysis. A RESET signal is available on the MIB to accurately synchronize the timestamp counters of all of counters in a situation where multiple nodes are monitored by as many boards. Thus, the error in time-synchronization is limited only by wire-distance and is far superior to the software time-synchronization. The next section compares the results from MIB with those obtained from software instrumentation.

4. RESULTS FROM MULTIKRON MONITORING SYSTEM

To determine the intrusiveness of various monitoring techniques, a test platform was set up using Linux 2.2-15 (Redhat 6.0) on a 200 MHz Intel Pentium-Pro processor (8KB Instruction cache and 8KB data cache) with 256 KB L2 cache and 96 MB DRAM. GNU's EGCS 1.1 compiler that comes along with Redhat 6.0 Linux, was used. Average execution time from one million iterations are listed in Table 1 below.

Table 1. Average Execution Time of Monitoring Techniques.

Technique	**Instruction**	**Average Execution Time**
Software	time(NULL) gettimeofday(&tv, NULL) clock()	1.77 μs 2.38 μs 2.41 μs
Hybrid	MIB Assignment Statement	0.36 μs

As noted before, gettimeofday(&tv, NULL) has the best resolution of 1 μs among all software instructions, whereas as mentioned before, MIB's timestamp counter has an order of magnitude better resolution. In addition, it can be observed from the above table that in terms of execution time the hybrid monitoring is more than four times better than the best software technique.

5. ISSUES IN BENCHMARKING DISTRIBUTED SYSTEMS

A benchmark suite provides a figure of merit for system-performance. This section provides a synopsis of the issues related to various benchmarks. The performance of a distributed system is impacted by the following components [5].

- CPU
- Memory
- Operating System
- I/O
- Network and Communication

Based on the level of details the test-results contain, there are two classifications of benchmarks: macro and micro [6]. A *macro-benchmark* contains minimum amount of detail in its result; generally, a number indicating how better or worse a given computer system is compared to some other system. Whereas the result from a macro-benchmark is a useful measure in comparing two different computer systems, it does not answer the question as to why a system performs better or worse than the other one. The usefulness of macro-benchmark is limited only to a user in making a decision about which system to use. However, to identify bottlenecks, integrators and designers of computer systems need detailed information. These details provide a

better insight into how the components fit and interact with each other in determining the system performance. Benchmarks that generate such detailed results are known as *micro-benchmarks*.

Another benchmark-classification is based on the system component they test. Earlier benchmarks were developed to test only the CPU, and so they consisted of computation-intensive programs to test the computational prowess of a computer system. Some others were then developed to test the file transfer or I/O capability of a computer system. A few others were primarily intended for transaction processing for a client-server system. However, no benchmark currently exists that can test all of the components of a distributed system listed above. The following sections discuss the issues and tools available to benchmark some of these individual components of a distributed system.

5.1. CPU Benchmarking

CPUs have been the focus of most computer system performance evaluation, and a major concern of such studies has been the interaction between the processor and the memory, which in turn depends on the following factors [5]:

- Instruction Set
- Hardware architecture: caching, pipelining, superscaling
- Clock rate
- Compiler
- Memory Management
- Operating System

A CPU benchmark, as the name suggests, primarily measures the computational prowess of a CPU and its interaction with the memory system. Therefore, CPU benchmark programs tend to be computation-intensive. A number of CPU benchmarks are available but the de-facto standard for CPU benchmarks is SPEC95 which consists of two suites: Cint95 and Cfp95. CInt95 is a collection of eight C programs, while Cfp95 is a collection of 10 Fortran-77 programs [7]. The output of SPEC95 benchmark suites is a number that provides a comparison of the performance of the system under test with the performance of a Digital Equipment Corporation's VAX-11/780 mini-computer of the late 70's.

5.2. File System and I/O Benchmarking

In Unix system, all I/O devices are treated as files. The performance of a computer system that includes I/Os is greatly influenced by the architecture used to interface the three components: CPU, memory, and I/O. In addition, the impact of operating system is very pronounced. Therefore, systems containing I/O devices possess a much higher level of complexity in benchmarking. The confusion in this area is further compounded by the fact that depending on the system and application, there are several performance-measures for systems consisting of I/O devices, and improving one measure might degrade the other. Thus, in contrast to processor performance benchmarking, the performance criterion for systems consisting of I/Os is not as unambiguous. The state of I/O benchmarking is still quite primitive compared to the processor benchmarking and above all is application dependent [5]. This is because of the widely varying schemes as a result of tradeoffs. Some of the notable file system benchmarks that appear to be promising for Unix systems are discussed below for the sake of completeness. Readers may consult the indicated references for further details.

The Andrew benchmark for Unix system is one of the oldest Unix file system benchmarks consisting of a suite of 70 files that take up 200 KB and run in five phases - MakeDir, Copy, ScanDir, ReadAll, and Make [8].

5.3. Operating System Benchmarking

System performance is impacted by the algorithm the OS uses to create, schedule, and switch processes, especially in multi-user or multi-tasking environment. Memory management techniques, if implemented under the operating system, also impact the performance of the system.

The resources for operating system benchmarking are almost non-existent. Nevertheless, the benchmark, LMBench [9], needs mentioning here. LMBench, is a synthetic micro-benchmark to test the operating system as well as the I/O capabilities of a computer system. LMBench relies on determining the execution time of the system and function calls needed to mimic the working of the operating system. For example, consider the problem of determining the system latency in creating a new process and running a new program. Unix performs this task by executing fork() and execve() system calls. Incidentally, this is how the Unix command-interpreter, *shell*, works in the inner loop. Thus, the latency in creating a new process and running a new program can be obtained by determining the execution time of fork()+execve() system calls. Clearly, a better operating system should have smaller execution time..

5.4. Prescription for a Comprehensive Micro-Benchmark Suite for TCMS

A test and checkout system is a complex distributed data processing system, and in addition to the components discussed above, consists of heterogeneous analog and discrete signals, multiplexers/demultiplexers, and networked systems with multiprocessors.

It should nevertheless be observed from the discussion in the preceding sections that due to the specific application and often competing requirements, a macro-benchmark is impractical in evaluating distributed systems. Instead, a better insight into all these issues can be obtained by quantifying the parameter that show how the components belonging to the system fit and interact with each other in determining the system performance. In other words, a micro-benchmark test suite that can test the components that impact the performance of a distributed system seems to be the best alternative. The idea here is to have a benchmark that can help the design and upgrade of the system. For example, consider the upgrade of the R10000 processors in Origin 200 by a later version. The benchmark should not only provide the gains for computation intensive tasks, it should also shed light on the new performance results for operating system and I/O operations as well. As a second example, consider replacing the optical drives of the Origin 200 system by faster drives. One would expect the file-transfers to be faster, but a CPU-bound system might not show a commensurate improvement in the file transfer rate because of the processor's inability to cope up with the more frequent interaction with the faster drives. These examples highlight the significance of a micro-benchmark in system design and upgrade. A comprehensive micro-benchmark being proposed here should have the following capabilities.

5.4.1 Computational Capability: This should consist of a set of computation–intensive programs to determine the computational capability of the host. In a multiprocessor system, the benchmark should provide information on the utilization of individual processors. A set of computation-intensive programs, such as LINPACK, should suffice [6].

5.4.2 Memory Performance: The following tests should reveal performance of cache and DRAM.

- Reading a block of data from resident memory (Test for different block sizes)
- Writing a block of data to resident memory(Test for different block sizes)
- Reading a block of resident memory and writing to a different block(Test for different block sizes)

5.4.3 File System and I/O Capability: Since I/O devices are treated like files in Unix, the following tests can benchmark both file systems and I/O devices.

- Time taken in creating/deleting files
- Time taken in writing to a file using *putc()*
- Time taken in reading from a file using *get()*
- Time taken in sequentially writing a block of data to a file or I/O device (Test for different file sizes): write(int *handle*, void **buf*, unsigned *len*)
- Time taken in sequentially reading a block of data from a file or I/O device (Test for different file sizes): read(int *handle*, void **buf*, unsigned *len*)
- Time taken in reading/writing random files: In this case, the buffer pointer is randomized by rand() in the function long lseek(int *handle*, long *offset*, int *fromwhere*). It should be noted that this randomized I/O function can be used to determine the time it takes for a disk drive to position its head at a random location.

5.4.4 Operating System: The following tests should reveal how efficiently the OS can create, schedule, and switch processes. Networking as well as the operation of many other tasks can be synthesized and the system benchmarked.

- Time to create a process
- The latency in context switching
- Data transfer rate through a Unix pipe
- Opening and closing TCP/IP connection

5.4.5 Network and Communication: It should be observed from the schematic of the TCMS that the user interacts with the AP or ARS through a number of X-Terminals connected to a common Ethernet. Since the user interaction is random, it is not realistic to obtain a deterministic benchmark value or a figure of merit. A possible approach would be the installation of a network monitor capable of providing important network statistics such as number of packets transferred, number of collisions, throughput, etc. Inexpensive COTS (commercial-off-the-shelf) network monitors are available. Conclusion about how the network is performing can then be drawn from these statistics. For example, if the number of collisions is high, the network is saturated either because of increased user requests or due to the bandwidth limitation of the network. The same strategy may be applied to the other Ethernet link that interconnects the FEPs (Front End Processor) of the TCMS to the AP and ARS in the upper half of the figure.

Unfortunately, the X-terminals are not intellligent terminals; they have display capabilities but they do not have any kernel. If they had a kernel, a benchmark program running on either the AP or the ARS could have been designed to emulate the user interaction by implementing a client-server model to account for the communication between AP/ARS and the user terminals. This aspect, however, needs more investigation.

The micro-benchmark suggested here would not only be useful for the test and checkout systems, but would also have the capability to test the CPU, operating system, and I/O capability of any generic computer system.

6. CONCLUSION

This paper discussed monitoring and benchmarking issues for performance evaluation of distributed systems with special emphasis on the test and checkout systems. Details of a prototype Linux-based hybrid monitoring system, that employs NIST's PCI-MultiKron card, have been provided. Results obtained from the monitoring system indicate that compared to software monitoring, the hybrid monitoring reduces probing effect by at least a factor of four. Finally, issues related to benchmarking a distributed system have been discussed and the details on how to develop a micro-benchmark for the TCMS have been provided.

7. ACKNOWLEDGEMENT

Thanks to Craig Jacobson and Robert Yascovic from the NASA Kennedy Space Center for their support. Thanks also to Wayne Salamon, NIST, Gaithersburg, MD, for his assistance in upgrading the Multikron code and for his prompt clarifications on several operational aspects of the Multikron interface board.

8. REFERENCES

[1] R. Hoffman, et al., "Distributed Performance Monitoring: Methods, Tools, and Applications," IEEE Trans. Parallel Distributed Systems, Vol. 5, No. 6, pp. 585-597, 1994.

[2] J. C. Harden, D. S. Reese, M. B. Evans, S. Kadambi, G. J. Henley, C. E. Hudnall, and C. Alexander, "In Search of a Standards-Based Approach to Hybrid Performance Monitoring," IEEE Parallel and Distributed Technology, Vol. 3, No. 4, pp. 61-71, Winter, 1995.

[3] A. Mink and W. Salamon, *Operating Principles of the PCI Bus MultiKron Interface Board*, NIST Report No. NISTIR 5993, U S Department of Commerce, Gaithersburg, MD, March 1997.

[4] A. Mink, et al, "Performance Measurement using Low Perturbation and High Precision Hardware Assists," *Proc IEEE Real-Time Systems Symposium, Madrid, Spain, pp 379-388, December 1998.*

[5] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Fifth Edition, Morgan Kauffman, San Francisco, CA, 1998

[6] K Hwang, and Z. Xu, *Scalable Parallel Computing*, pp. 2631, WCB McGraw Hill, Boston, MA, 1998.

[7] SPEC95 Benchmark, SPEC, Santa Clara, CA. (http://www.spec.org/osg/cpu95/).

[8] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems*, 6(1), February 1988.

[9] L McVoy and C. Staelin, "lmbench: Portable Tools for Performance Analysis," *Proc. Of the 1996 USENIX Technical Conference*, pp. 279-295, San Diego, CA, January 1996. (http://www.bitmover.com/lmbench/)

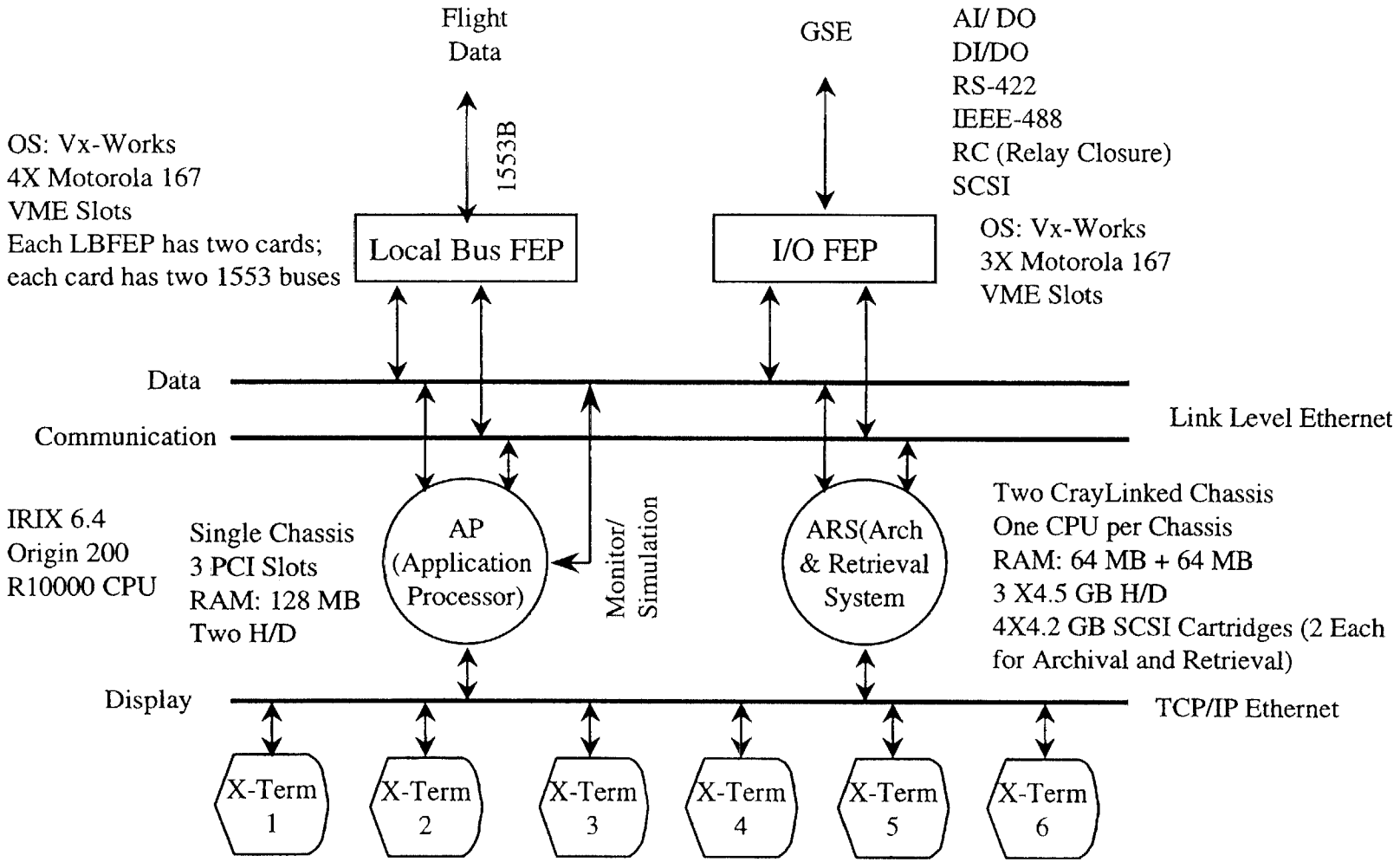

Figure 1. Schematic of TCMS