NASA-TM-104,223

NASA Technical Memorandum 104223 NASA-TM-104223 19920010422

COMPUTER OPTIMIZATION TECHNIQUES FOR NASA LANGLEY'S CSI EVOLUTIONARY MODEL'S REAL-TIME CONTROL SYSTEM

Kenny B. Elliott, Roberto Ugoletti, and Jeff Sulla

February 1992

NASA
National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665

COMPUTER OPTIMIZATION TECHNIQUES FOR NASA LANGLEY'S CSI EVOLUTIONARY MODEL'S REAL-TIME CONTROL SYSTEM

KENNY B. ELLIOTT
NASA Langley Research Center
Hampton, VA 23665

ROBERTO UGOLETTI AND JEFF SULLA
Lockheed Engineering and Science Company
Hampton, VA 23666

KEYWORDS

Digital Control Systems, Integrated Control Systems

ABSTRACT

The evolution and optimization of a real-time digital control system is presented. The control system is part of a testbed used to perform focused technology research on the interactions of spacecraft platform and instrument controllers with the flexible-body dynamics of the platform and platform appendages. The control system consists of CAMAC standard data acquisition equipment interfaced to a workstation computer. The goal of this work is to optimize the control system's performance to support controls research using controllers with up to 50 states and frame rates above 200 Hz. The original system could support a 16-state controller operating at a rate of 150 Hz. By using simple yet effective software improvements, I/O latencies and contention problems are reduced or eliminated in the control system. The final configuration can support a 16-state controller operating at 475 Hz. Effectively the control system's performance was increased by a factor of 3.

INTRODUCTION

As space-science data quality requirements become more stringent, the performance of space platforms become more critical. One of the key elements of a space platform's performance is the interaction of the platform/instrument controller with the flexible-body dynamics of the platform and it's appendages. This controls/structure interaction (CSI) is the topic of a NASA sponsored focused technology research program. The objective of this focus program is to develop technology to enhance the CSI performance and design of future spacecraft structures.

The CSI program concentrates on three complementary areas: spacecraft design, ground testing, and flight verification. The program has lead to the development of several testbeds for testing and demonstrating CSI technology. At the NASA Langley Research Center, a testbed has been developed which simulates realistic structural characteristics and control hardware of a generic large space platform (1). The testbed is being used to develop test methods, evaluate flexible structure control algorithm performance, evaluate sensor and actuator technology, evaluate CSI design concepts, and experimentally assess the level of confidence with which CSI technology can be applied to space platforms.

The testbed, shown in Figure 1, is called the CSI Evolutionary Model (CEM). The CEM can be subdivided into three major components: the structure, the instrumentation, and the real-time digital control system. This break down is shown in Figure 2. The structure has been designed to possess the dynamic properties of a typical future spacecraft. It consists of a 55 foot long truss-bus with several appendages which possess varying degrees of flexibility. The structure is modular in design, and it's configuration can change in order to support program needs. The CEM can be adapted to carry almost any type of sensor/actuator, such as CMGs, gimbals, or IMUs. Currently, the instrumentation is limited to inertial sensors (accelerometers and rate sensors), a laser-based line-of-sight sensor, and cold-gas thrusters acting as actuators. The real-time control system is a combination of commercial and custom components. The system performs system monitoring, system safety, signal conditioning, and serves as a host platform for the control algorithm.

One of the strengths of this testbed is it's flexibility to support current as well as future research needs. Each part of the CEM is equally important to the testbed's evolution, and each part will evolve as the CSI program develops.

This paper details the current evolution and computer optimization techniques used to enhance the CEM's real-time control performance.

BACKGROUND

The control system is a digital computer which reads information from the sensors, processes this information through a control algorithm, and sends control commands to the actuators. Specifically the system performs, in real-time, its own data acquisition, signal conditioning, data management, computations of state, control, and safety limit checking, and provides a user interface.

Originally, the CEM's control system was a carry-over from a previous program; the CSI Mini-Mast program (2). Mini-Mast's control system was part of a system which was developed to support real-time flight simulation research. This system is call the Advanced Real-Time Simulation (ARTS) system (3,4). During the Mini-Mast program the ARTS system was adopted for real-time controls research (5). The system's configuration is shown in Figure 3. The system has six major components: the data acquisition equipment located near the structure; the network switch and fiber-optic highways; the real-time computer located at a central computing complex; a real-time clock/timer subsystem, and software to run the system. A brief description of the system follows; however, a detailed description of the ARTS system can be found in Ref. 3 and 4. The data acquisition subsystem is based on the CAMAC[*] standard (5). Analog-to-digital (ADC) and digital-to-analog (DAC) signal converters are contained on modules which connect into a chassis called a crate with a backplane bus called a dataway. The crate functions are controlled by a crate controller which in turn is controlled by a mainframe computer; a CDC CYBER 175. The crate controller is connected to the mainframe computer through a ring master/slave fiber-optic network called a highway. This network includes a network switch for dynamically switching between different remote sites. A clock subsystem is provided for system synchronization and real-time clock services. The system is controlled through a console subsystem. The operating

* Computer Automated Measurement and Control.

system is NOS which has been modified to support real-time operations. The control software is a FORTRAN program built around a linear generic control law (GCL) subroutine. A description of the control software can be found in Ref. 6.

The performance of this system is shown in Figure 4. This figure shows the available computation time (the time available to perform control computations) as a function of frame rate. The frame rate is the inverse of the time it takes to acquire sensor data, execute the control law and safety checks, and transmit the actuator data. Practically, this system is limited in two fundamental ways. First, the minimum clock resolution, at this time, is 5 msec, resulting in a maximum frame rate of 200 Hz. Currently, the control law implementation requires over-sampling techniques which require frame rates from 150 to 500 Hz. The frame rate must encompass the time required to perform all necessary control computations. As the frame rate increases, the time available for computations decreases. Therefore, high frame rates handicap the controller's complexity. For example, if a 16-state linear time-invariant controller is run on this system, then the maximum system frame rate drops to 125 Hz. The computation time of the controller extends the frame size. This condition is called "compute bound." Currently, controllers with up to 100 states are being considered. The second limitation is scheduling. The ARTS system is a field-wide resource. As such, the system must be time managed. Session time of 2 to 4 hours are usually scheduled 2 weeks in advance of the planned test. This severely limits the spontaneity of research.

SYSTEM DESCRIPTION

Shortly after the CEM became operational, an effort was undertaken to evaluate methods of increasing the performance of the controller system. The goal was to arrive at a system which would allow daily unscheduled system use with controllers of a moderate size, up to 50 states, at a speed which meets or exceeds those of the existing ARTS system. The system needed to be developed at a low capital cost and fit into the existing computer environment. These constraints led to the use of an existing local workstation, the existing CAMAC equipment, and the GCL software to develop a local control system.

The base-line system under consideration is shown in Figure 5. This configuration is accomplished by adding a second crate controller which communicates to a workstation. The crate controller communicates over a parallel bus to the Q-bus backplane of a DEC VAXstation 3200 workstation. The workstation is configured with a color graphics terminal, 16 MB of memory, a real-time clock card with a 1 μsec resolution, and 300 MB of disk space. The computer operates using the VAX/VMS operating system. This configuration is supported by the CAMAC equipment manufacturer. Therefore, no custom equipment needed to be developed.

The initial CEM control program for this configuration was designed and developed to operate as similar as possible to the ARTS system's GCL code. Due to different operating systems, different data acquisition calling methods, the large software overhead required by the ARTS system, and different versions of FORTRAN, most of the GCL software was rewritten. However, the fundamental control algorithms and their implementation were not changed. Basically, the program provides signal generation, open and closed-loop modes of testing, digital filtering, software safety checks, test logging, and feed-back control. The control law is implemented using the following equations:

$$U_{k+1}=CX_{k+1}+DU_k$$

$$X_{k+1}=AX_k+BU_k$$

where A, B, C, and D are constant matrices of appropriate dimension,

X_k	=	column vector of controller states at time k,
X_{k+1}	=	column vector of controller states at time k+1,
U_k	=	column vector of actuator commands at time k,
U_{k+1}	=	column vector of actuator commands at time k+1, and
k	=	time index.

The main section of the control code consists of the following steps:

1. read the analog sensors through the CAMAC crate,
2. remove sensor bias and limit check the sensor inputs,
3. compute actuator commands,
4. limit check the new actuator commands,
5. output the actuator commands through the CAMAC crate,
6. update the controller state equations (closed-loop),
7. save current sensor data and actuator commands in a memory array,
8. repeat the above sequence for a number of iterations based on the delta loop time to achieve the total amount of control test time required,
9. after all loops are done, zero the final actuator commands,
10. write the final array of accumulated sensor data and actuator commands.

The initial version of the control program on the VAXstation 3200 real-time control system ran and performed all of the above functions. Various control laws could now be tested locally without having to schedule time on the ARTS system. The performance of the control system, shown in Figure 6** in terms of controller states and frame rates, approached very close to that of the ARTS system. A 16-state control law executed at a frame rate of 125 Hz. However, the frame rates were not stable and were highly affected by background system activity.

In order to quantify the stability of the system, two variables were added to the control program; clock overrun and frame overrun. A clock overrun occurs when the system takes a longer time than specified to complete one frame. A frame overrun occurs when a frame, or one cycle, has been skipped. The program detects these events by monitoring the clock tick generated by the real-time clock. A frame is the time between two clock ticks. The start of a frame occurs at the next clock tick. When the control computations are complete, the program loop polls the clock register and idles until the next clock tick occurs. If the program detects the clock tick already set before it idles, it knows that it took too long to execute and exceeded its allotted frame time; i.e., it overran the clock tick (clock overrun). Also, if the clock hardware rolled over twice without a reset of the first clock tick, it will flag another bit in the status register to inform of a completely missed clock tick duration (frame overrun).

OPTIMIZATION TECHNIQUES

A post-analysis of the system's performance was performed. This analysis consisted of analyzing clock and frame overruns to determine when and how they occurred. Also, timing analyses were performed on individual and groups of hardware and software components. The analysis revealed several inefficiencies. Specifically, I/O was slow;

** The number of controller states was determined to be a more meaningful measure of system performance instead of available computational time. The number of states, which refers to the size of matrix A in Eq. 1, combined with the number of inputs and outputs, define the computational load of the system. The number of inputs and outputs are held constant at 8 each.

virtual memory created delays; background system activity caused delays, and the generic software itself was inefficient. By addressing these problems, improvements in controller performance could be made if certain simple software methods are applied to the control program. These methods would involve:

1. getting more control over the CPU,
2. reducing background system activity,
3. defeating virtual memory,
4. reducing I/O bottlenecks,
5. implementing a real-time operating system, and
6. optimizing the controller software algorithm.

Each of these methods will be addressed below.

CPU AND BACKGROUND ACTIVITY CONTROL

Getting full control of the CPU means to use all the available CPU time that is present and not share time with other users or background activities. CPU usage is directly attributable to the running of tasks and interrupts, and the switching between them. A description of task priorities, context switching, interrupts, and their effect on the control program are explained below. Solutions to these problems that were applied are also discussed.

In a multi-tasking system, tasks can be directly related to users as well as background activity which is providing network services or workstation windowing functions. The way tasks are processed is directly affected by task priorities. Task priorities are operating system values assigned to a program or task that allow them to have a more or less favorable status as to which task should execute next. All multi-tasking operating systems have time-sharing task priorities. At these levels, the scheduler allows a task to run for a maximum fixed amount of time before switching to the next task of highest priority. Minor adjustments of task priority are automatically made by the scheduler to permit improved interactive and I/O performance over CPU-bound tasks. Time-sharing task scheduling cannot guarantee sufficient CPU time for a control program to run at a regular rate. Priorities will fluctuate allowing background processes such as networking and screen windowing to affect control times even on a single user system. A limited number of multi-tasking operating systems have an additional range of task priorities referred to as real-time. This priority range is always higher than the time-sharing range of priorities. At real-time priority levels, a task is allowed to run as long as it needs to and is the highest priority task ready to run.

When the scheduler switches between tasks (context switch), time is taken to perform the switch. The actual switch involves saving all current CPU registers such as program counter, processor status, accumulators, stack pointers, floating point processor registers, etc. and loading the saved registers for the new task into CPU to execute it. The scheduler typically performs a context switch at a periodic rate usually determined by the system clock rate (10 ms for a VAX/VMS system) and at the request of a task which wants to start/stop another task or itself. Some systems also look for a possible context switch after every I/O completion event. Context switching becomes a concern to a real-time control program when it must run at a periodic rate that's faster than the scheduler rate or not a multiple of it. A real-time clock may signal to the operating system to start a real-time task at an exact time, but the scheduler may wait until its next execution period before actually performing the context switch. Any real-time program that cannot tolerate delays caused by context switching or must guarantee exact timing, and does not have a sufficiently robust scheduler or I/O event signaling, should not relinquish the CPU by hibernating or sleeping.

CPU interrupts and their service routines could also preempt the execution of the control program and cause delays. Interrupts are events that enable a CPU to respond asynchronously and immediately to an event either internal or external to the CPU. Since interrupts are usually urgent and must be serviced immediately, they preempt any and all tasks that are currently running.

A very simple program was written to test the operating system for background activity that would impact a real-time control task. A continuously running system clock is read repeatedly for the duration of the test (typically 5-10 minutes is sufficient). A delta time is calculated between each successive read. The minimum delta is the clock resolution and the maximum delta is the worst case background activity that occurred during the test period. The test may be run at various priority levels, simultaneous user, network, and peripheral device activities. The lower the maximum time, the quieter the operating system is or the higher the test program's task priority is over all other activities. The affect of various background activities on this system's performance is shown in Table 1. The tests showed at worst-case, a control program, running at normal priority, could be interrupted and delayed by up to 320 ms. This delay is the result of background system activity due to networking, interrupts, and task contention with other processes.

The control program was changed to handle task priorities dynamically. This was implemented by using a system service call in the control program to raise and lower task priority as needed. The control program was modified to start at normal priority, perform user interaction and initialization, then raise its priority to a real-time task level during the actual control period. When the control period is completed, priority is restored to normal to perform final computations and data storage as required. This eliminated almost 78% of the interruption and delay, but a 70 ms delay still occurred.

There are only two areas where the program had an opportunity to be affected by context switching. One area was at idle time waiting for the end of a frame clock tick. However, the program polls for the occurrence of the clock tick status bit which requires no intervention of the operating system and therefore the possibility of a context switch occurring. The second area of concern was the system I/O calls to CAMAC hardware. It was known that the I/O call would relinquish CPU control to a lower priority task through a context switch, but unknown if it could regain it back at I/O completion right away. Timing tests showed no odd skewing of I/O times; therefore, the completion of the I/O was causing a reschedule of tasks to occur and the higher priority control task would immediately resume execution. No changes to the control program were required to protect against context switching.

In the test, the source for interrupts was the simple moving of the workstation mouse which would send a series of position change interrupts to the workstation. Only when all windowing and mouse movement activity were halted, did background system activity decrease below detectable levels (10 ms which was the same as the system clock resolution). The VAX/VMS operating system provided no way for a user program to lock out or inhibit specific interrupts from occurring. The only resolution was to not move the mouse and to stop running certain network services that were trying to respond to network broadcast messages and generate periodic status information.

These changes to running the control program gave much more consistent execution times. A specified total controller run time of 30.0 seconds now executed in 30.0 to 30.1 seconds and not an erratic longer execution time of 30.1 to 31.5 seconds.

DEFEATING VIRTUAL MEMORY

Virtual memory is the ability for a program to reference more memory than is physically available. Virtual memory is typically implemented on a computer system by using random access mass storage devices (hard disks) to store the virtual program code, data, stack, and operating environment. Either small segments (paging) or the entire program (swapping) may be moved to/from physical memory and mass storage. This is a definite advantage for multi-user systems which are trying to conserve memory resources but detrimental to real-time programs which must wait huge amounts of time for page faults and the resulting disk I/O to complete before continuing to execute. To insure that the control program was not incurring page faults during its controller execution, certain steps had to be performed to keep the program and its data in physical memory. The program needed to be loaded and locked into physical memory just after all initialization functions were done and before the real-time control phase.

The method of forcing a program into physical memory on a VAX/VMS system involves locking pages into the working set. The working set is a representation of the physical memory that a process or program is using. It is used by the operating system and the memory management hardware to track which virtual address pages of the program actually point to physical memory. If excessive page faults occur, the operating system will increase the working set size to allow more physical memory to be used. Before a program can lock all of its virtual pages into the working set and have a one-to-one correspondence between virtual pages and physical pages, it must have a sufficiently large working set to hold the entire program at once. Once user account quotas are raised to sufficient levels, a system service routine is called to lock a range of program virtual address pages into the working set. If any pages are not currently in the working set, they are brought in by page faulting and then locked so that they will not be moved back out to mass storage and the actual physical memory reused by another program.

With the program code and data locked into memory, the process header was the last unprotected part of the program that could incur a page fault. A second system service call was added to the control program to prevent the process from being swapped out of the list of memory resident processes. The entire program now had a physical page associated with each virtual page and would not incur any additional page faults during any part of its control phase execution.

Tests performed showed that no page faults were occurring in the control phase regardless of whether the program was locked into the working set or not. Page faults did not occur because the data arrays, which incur the page faults, were accessed previous to the control phase of the software. However, as a precaution, locking of pages into the working set is still recommended and performed.

REDUCING I/O BOTTLENECKS

Once the control program's execution was stabilized by controlling the CPU and defeating virtual memory, the first detailed timing test could be performed. These tests showed that the existing control system, running a 16-state controller at 125 HZ (8.0 ms), was using 2.8 ms to read analog sensor channels and 2.2 ms to write actuator command channels. This translated into a total I/O time of 5.0 ms or 63% of the control loop time. These timing tests clearly showed that the majority of time was spent performing I/O (an I/O bound problem). Generally, there are two ways to improve most I/O times; eliminate or combine I/O requests, and reduce the amount of overhead associated with each I/O request.

System I/O calls on a typical VAX workstation (VS3200) are time expensive, consuming anywhere from one to three milliseconds. Any way to reduce the number of I/O calls would improve performance. In this CAMAC configuration, with standard CAMAC commands, there is no problem combining read and write commands. This allowed a single list of CAMAC read and write commands to be created and initiated with just one I/O call instead of two. This simple change improved CAMAC I/O performance from 5.0 milliseconds to 3.4 milliseconds. This was a 32% improvement in I/O performance and allowed a 16-state controller to increase from 125 Hz to 150 Hz for a 16% improvement in overall controller performance.

To deal with the overhead associated with an I/O call, an understanding of the call process is needed. System I/O calls are requests made to the operating system to perform an input, output, or control to a device. After checking some parameters, the request is typically queued to a device driver. Device drivers are the last layer of software that communicates directly with the device hardware by reading and writing to special registers on the interface card or chip. I/O requests that can not be completed immediately require a wait period for the device to perform the operation. The device driver will perform these operations and wait for notification by the device of a step or operation completion. Although highly useful and functional, device drivers add an extra layer of software between a program and hardware. With a single dedicated use of the CAMAC interface (no multiuser arbitration is required) and a low level of device register complexity, it is fairly easy to bypass the device driver and write directly to device control registers themselves.

The first attempt to improve I/O performance was to use the 24-bit programmed transfer mode for CAMAC I/O, and read/write the CAMAC registers directly in the main control program rather than performing system I/O calls to the device driver. Programmed transfer mode requires the CPU to read and write every word between device and physical memory rather than having a dedicated controller chip directly transferring the data to/from memory (DMA). Setting up DMA transfers of data into physical memory is slightly more complicated and for small channel counts is not that much faster.

From a programing point of view, reading and writing directly to device registers from a program after all necessary system service calls are made is as simple as reading and writing to local program variables and arrays in memory, hence the name memory-mapped I/O. The additional requirement for a program operating in a virtual memory environment is to map a portion of its virtual address space into the physical address space where the particular device is configured to appear/respond at.

The new memory-mapped I/O technique was implemented and timed. The CAMAC I/O performance improved from 3.4 milliseconds to 0.8 milliseconds. This was an additional improvement in I/O performance of 325% and allowed a 16-state controller to increase from 150 Hz to 270 Hz for a 80% improvement in overall controller performance.

IMPLEMENTING A REAL-TIME OPERATING SYSTEM

A concern still remaining was that, for a fixed controller size, the frame rate could be increased from the point where no clock overruns were occurring to a point where the overruns exponentially increased until finally a frame overrun occurred, as shown in Figure 7. The reason that there existed a rate difference between one clock overrun and one frame overrun was that there was additional background system activity at a very high level that could not be eliminated.

A real-time operating system would allow a much greater control of the CPU and eliminate more of the background system activity. The most compatible real-time operating system with the VAXstation 3200 and VAX/VMS was VAXeln from DEC. VAXeln could run on the current VAXstation without any hardware modifications. Since it is a minimal operating system and fully configurable for the amount of system services needed, it will have less background tasks and peripherals consuming valuable CPU time. The real-time operating system has only real-time task level priorities. It also implements most of the device driver functions at the task level rather than at an interrupt level. This means it is possible to run a control program above the activity generated in servicing a particular peripheral or network service. The real-time operating system also provides the capability to perform from a program level, the starting and stopping of network functions, and the selective disabling of interrupts.

The configuration of VAXeln allowed selection of a minimal amount of device drivers to support the exact peripherals used. The disk device driver and file I/O allowed VAXeln to access the existing disk drives on the workstation and read/write the same disk file format. This meant that no changes were required to the control program so that it could read the same data files that were used for initialization and write the same data files for saving sensor and thruster commands.

Program development used the exact same compilers and linkers that were currently used for the CEM control program; both VAX FORTRAN and VAX "C". Only a different run-time library had to be linked with the compiled code to create an executable that would run under VAXeln. The only changes required to be made to the control program were those that performed system service calls. No locking of pages in memory was required since VAXeln was a real-time operating system that did not support virtual memory. All programs are required by VAXeln to fit into physical memory due to real-time constraints. The system calls to perform memory mapped I/O, and raise/lower task priority were different and required a small change. These were the only changes necessary to make the CEM control program run under the VAXeln real-time operating system.

The new real-time operating system was implemented and timed. The performance of a 16-state controller improved from 270 Hz to 357 Hz for a 32% improvement. The elimination of background activity is shown in Figure 7.

OPTIMIZING THE CONTROL ALGORITHM

The controller designs developed by researchers for the CEM varied considerably in their complexity. Some controllers use a low order, sparsely populated, controller A matrix while other controllers use a high order, fully populated, controller A matrix. As the order of the controller increased, the computation time associated with the $A*X_k$ multiply increased dramatically. The increase in computation time severely limits the digital frame rate at which the higher order controllers could be run.

The first approach to optimizing the state vector calculation was to take advantage of the structure of the A matrix in second-order controllers. For second-order controllers, the A matrix is in a block-diagonal form. Therefore, the state calculation routine was modified to operate only on the 2x2 block diagonal elements of the controller A matrix (all other elements of the A matrix are zero). This method was successful in increasing the allowable frame rate for this type of controllers.

The success of this procedure motivated the development of a procedure for block-diagonalizing the more complex controllers. The controller A matrix was block diagonalized using an eigenvector transformation process. Because the diagonalized matrix could include real and complex poles, the state calculation routine was modified to perform the matrix multiply using a tri-diagonal form of the controller A matrix.

The control program also checks for the D matrix upon initialization. If a zero D matrix is present, the controller output equation skips the $D*U_k$ calculation.

The performance of the 16-state controller is increased 33% from 357 Hz to 475 Hz. However, the real advantage of these techniques are realized for large controllers. A 38 state controller will run at frame rate of 140 Hz before optimization, and after optimization the controller's performance is increased to 280 Hz; a 100% improvement.

OVERALL SYSTEM PERFORMANCE

The overall performance evolution of the system is shown in Figure 8. The performance is referenced to the original system's performance which is originally shown in Figure 6. Methods which dealt with the increasing the performance of the hardware have a greater affect over I/O bound problems (high frame rates). However, improving the computational efficiency (diagonalization) has the greater affect on compute bound problems (high number of states). The goals of the system were exceeded. The system functions locally with no scheduling. The system can be used to run control laws with over 40 states at a moderate rate, and the system outperforms the ARTS system by a factor of 3 to 4 for medium size controllers.

SUMMARY

The evolution and optimization of a real-time digital control system was presented. The goal of this work was to optimize the control system's performance to support controls research using controllers with up to 50 states and frame rates above 200 Hz. The original system could support a 16-state controller operating at a rate of 150 Hz. The issues addressed were CPU control, reduction of background activity, defeating virtual memory, reducing I/O bottlenecks, implementing a real-time operating system, and optimizing the controller software algorithm. By using simple yet effective software improvements, I/O latencies and contention problems are reduced or eliminated in the control system. The final configuration could support a 16-state controller operating at 475 Hz. Effectively the control system's performance was increased by a factor of 3 to 4.

REFERENCES

1. Belvin, W. B., et. al, "Langley's CSI Evolutionary Model: Phase 0," TM 104165, NASA Langley Research Center, Hampton, VA, September 1991.

2. Tanner, C. E., et. al.,"Mini-Mast CSI Testbed User's Guide," TM 102630, NASA Langley Research Center, Hampton, VA, September 1991.

3. Crawford, D. J. and J. I. Cleveland, III, "The New Langley Research Center Advanced Real-Time Simulation (ARTS) System," AIAA-86-2680, Presented at the AIAA/AHS/ASEE Aircraft System Design and Technology Meeting, Dayton, Ohio, Oct. 20-22, 1986.

4. Crawford, D. J., J. I. Cleveland, III, and R. O. Staib, "The Langley Advanced Real-Time Simulation (ARTS) System Status Report," AIAA-88-4595-CP, Presented at the AIAA Flight Simulation Technologies Conference, Atlanta, GA, Sept. 7-9, 1988.

5. Anonymous, CAMAC Instrument and Interface Standards, The Institute of Electrical and Electronics Engineers, Inc., New York, New York, 1982.

6. Wood, D. V., D. W. Geyer, and J. Sulla, "Real-Time Control System for Mini-Mast Using the Advanced Real-Time Simulation System at NASA Langley Research Center," 61st Shock and Vibration Symposium, Pasadena, CA, Oct. 1990.

TABLE 1. Background Activity

TEST CONFIGURATION					Minimum Noise Resolution	Maximum Noise Resolution
Network Active	Task Priority	Terminal Device	No. Of Users	Mouse Activity		
Yes	High	VT 220	1	-	10 msec	10 msec
Yes	Normal	VT 220	1	-	10 msec	10 msec
Yes	Normal	Workstation	1	No	10 msec	10 msec
Yes	High	Workstation	1	Yes	10 msec	70 msec
Yes	Normal	Workstation	1	Yes	10 msec	320 msec

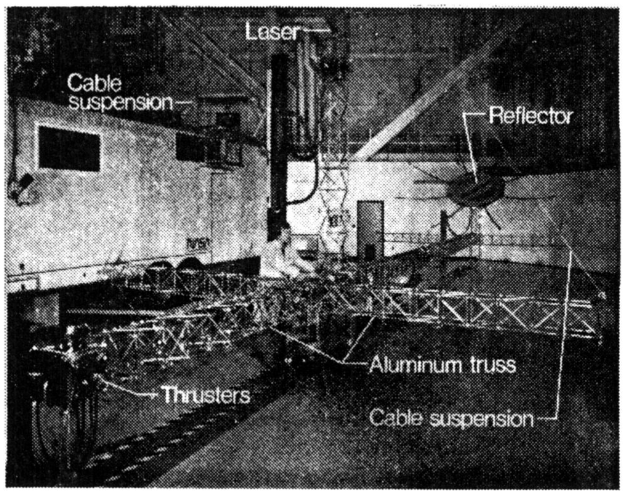


FIGURE 1. Photograph of the CEM Testbed

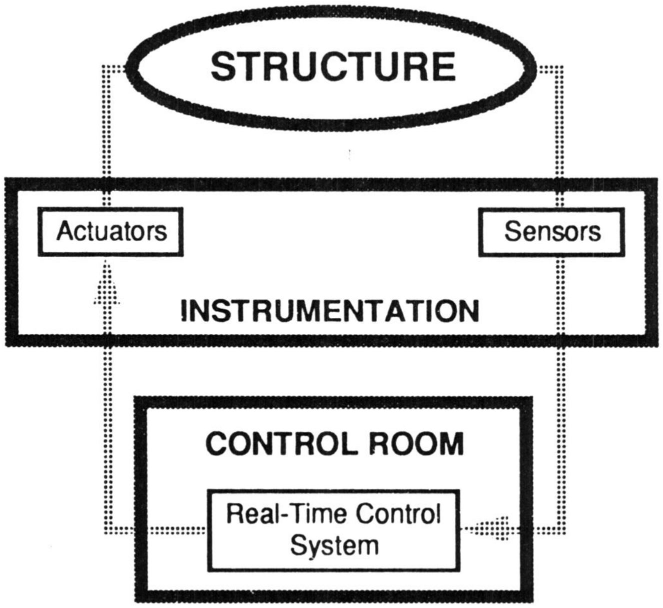


FIGURE 2. CEM Testbed Components

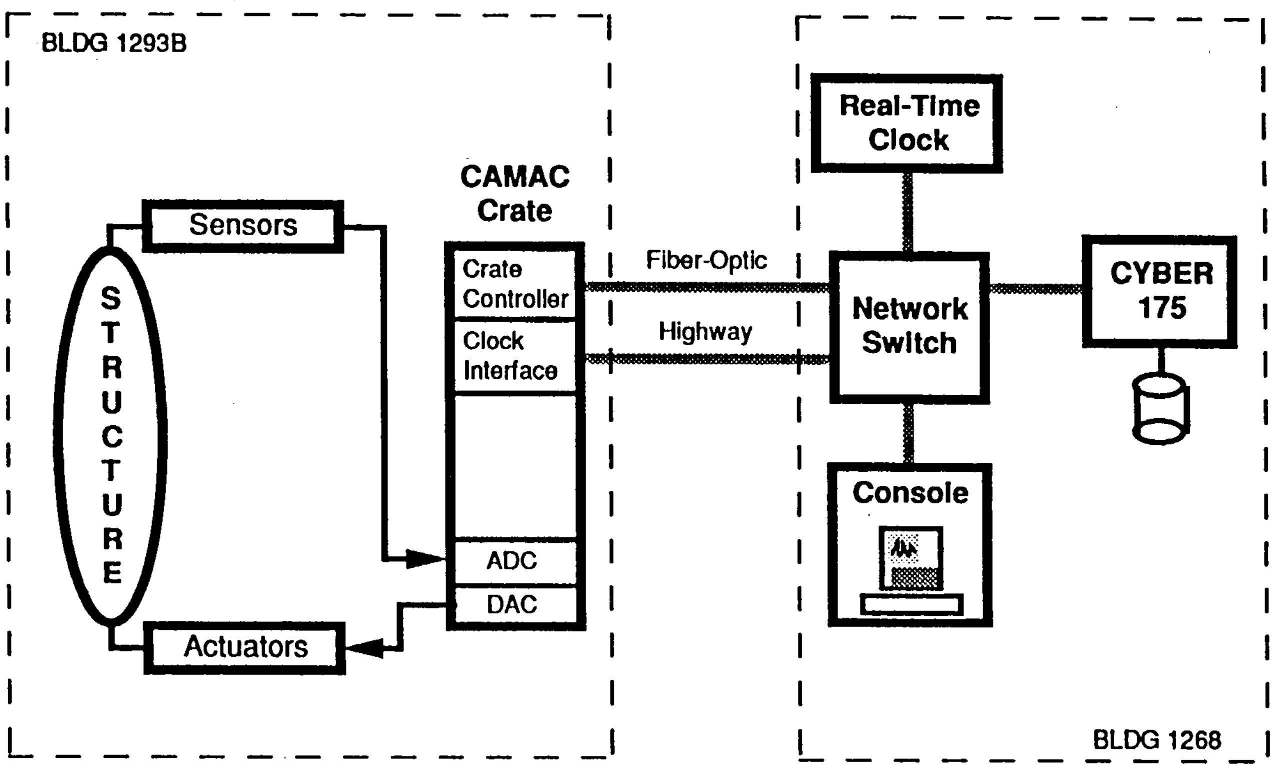


FIGURE 3. ARTS System Configuration

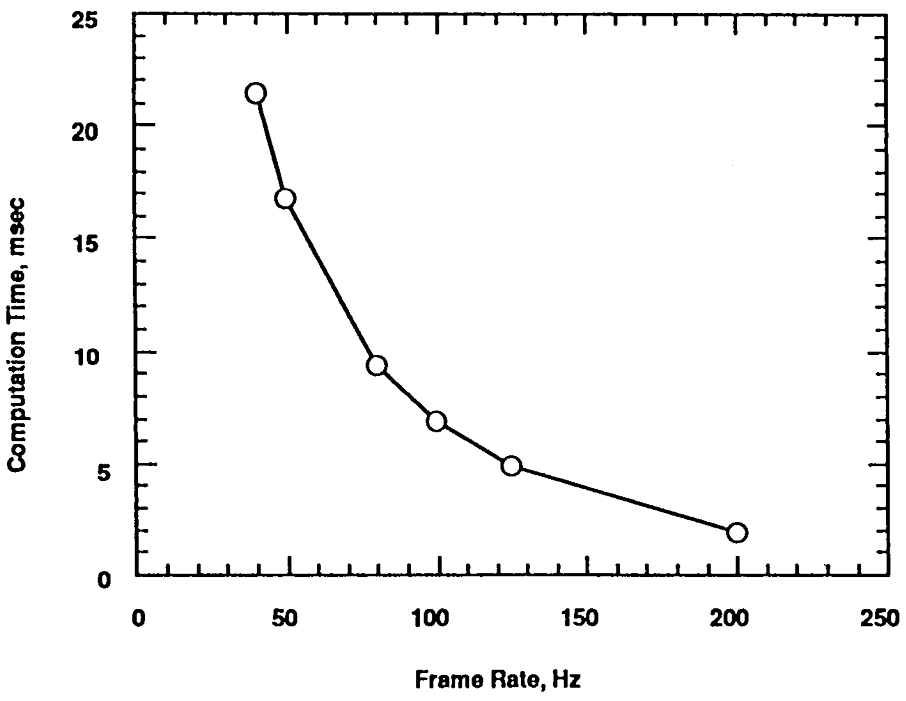
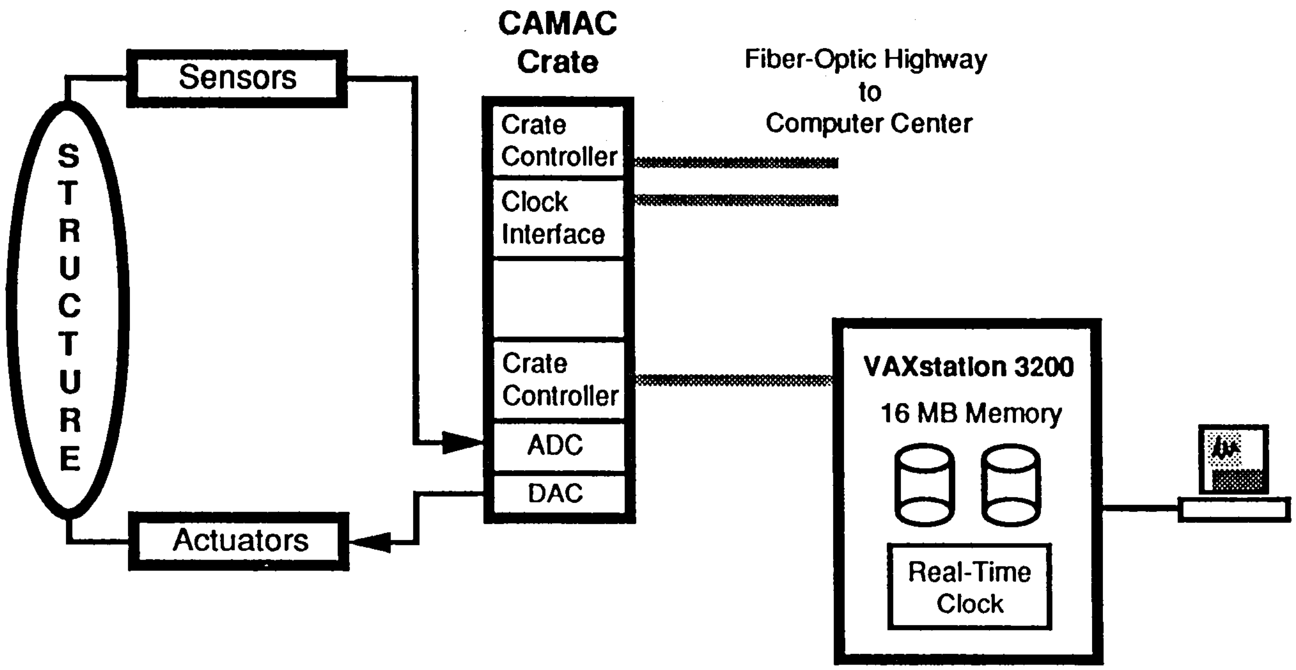


FIGURE 4. ARTS System Available Computational Time



FIGUE 5. CEM Control System

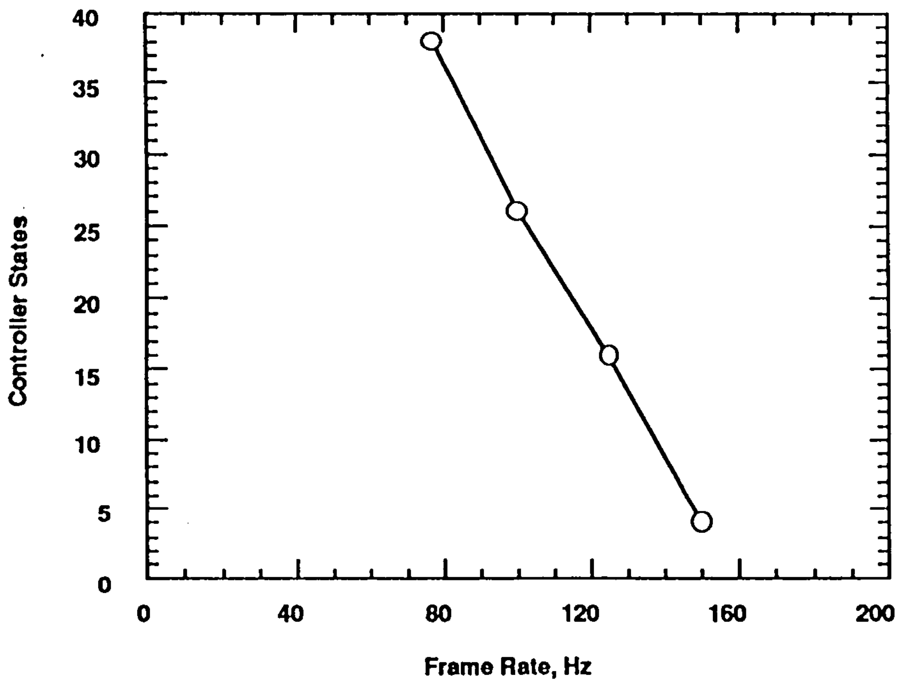


FIGURE 6. Initial CEM Controller Performance

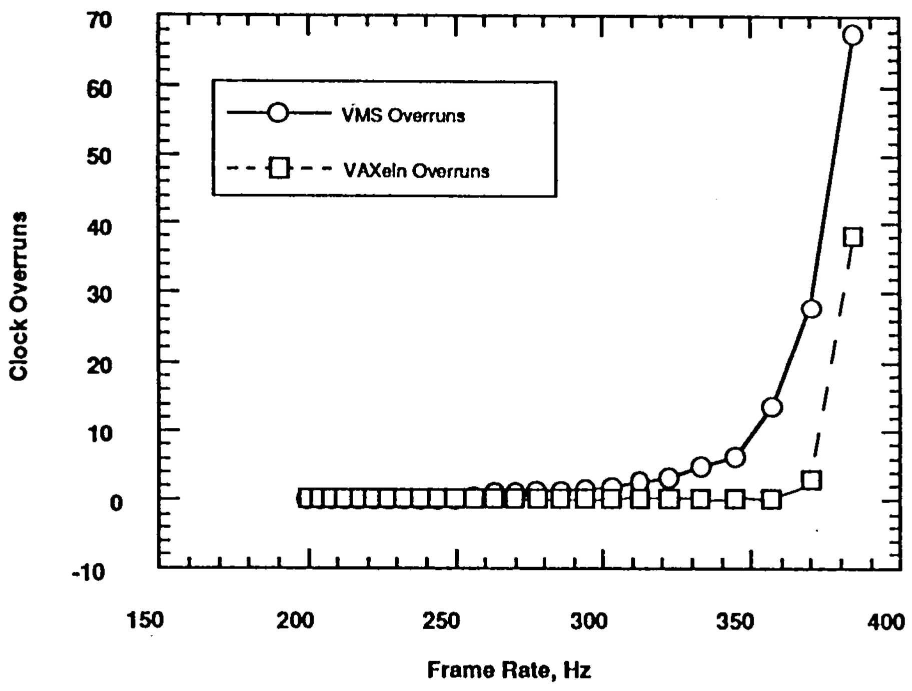


FIGURE 7. 16 State Controller Clock

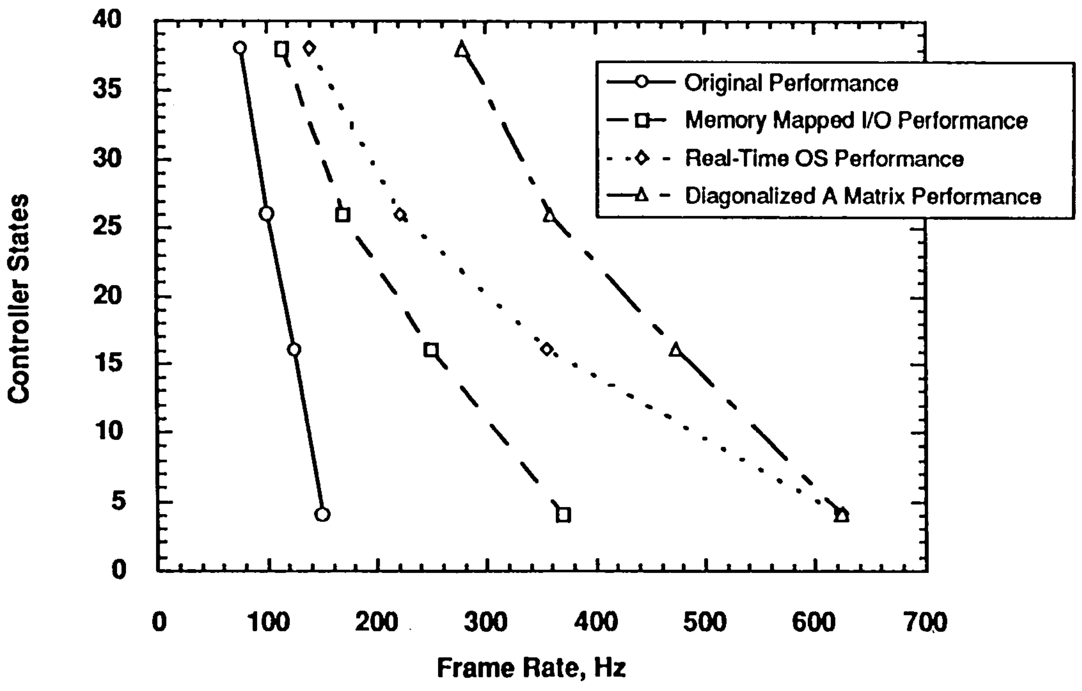


FIGURE 8. Progressive Controller Performance

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE February 1992	3. REPORT TYPE AND DATES COVERED Technical Memorandum

4. TITLE AND SUBTITLE Computer Optimization Techniques for NASA Langley's CSI Evolutionary Model's Real-Time Control System	5. FUNDING NUMBERS WU 590-14-61-01
6. AUTHOR(S) Kenny B. Elliott, Roberto Ugoletti, and Jeff Sulla	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NASA Langley Research Center Hampton, VA 23665-5225	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-001	10. SPONSORING/MONITORING AGENCY REPORT NUMBER NASA TM-104223

11. SUPPLEMENTARY NOTES
Elliott: Langley Research Center, Hampton, VA.
Ugoletti and Sulla: Lockheed Engineering & Sciences, Co., Hampton, VA.
Presented at the 38th International Instrumentation Symposium, Instrument Society of America, Las Vegas, NV, April 26-30, 1992.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified-Unlimited Subject Category 39	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The evolution and optimization of a real-time digital control system is presented. The control system is part of a testbed used to perform focused technology research on the interactions of spacecraft platform and instrument controllers with the flexible-body dynamics of the platform and platform appendages. The control system consists of CAMAC standard data acquisition equipment interfaced to a workstation computer. The goal of this work is to optimize the control system's performance to support controls research using controllers with up to 50 states and frame rates above 200 Hz. The original system could support a 16-state controller operating at a rate of 150 Hz. By using simple yet effective software improvements, I/O latencies and contention problems are reduced or eliminated in the control system. The final configuration can support a 16-state controller operating at 475 Hz. Effectively the control system's performance was increased by a factor of 3.

14. SUBJECT TERMS Digital Control Systems Integrated Control Systems	15. NUMBER OF PAGES 15
	16. PRICE CODE A03

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std Z39-18
298-102