Distributed Hartstone Real-Time Benchmark Suite

Clifford W. Mercer*, Yutaka Ishikawa** and Hideyuki Tokuda*

March 13, 1990
CMU-CS-90-110 3

School of Computer Science*
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213 USA
cwm@cs.cmu.edu, hxt@cs.cmu.edu

Electrotechnical Laboratory**
1-1-4 Umezono, Tsukuba-shi
Ibaraki 305 JAPAN
yisikawa@etl.go.jp

Condensed version appeared in *Proceedings of the 10th International Conference on Distributed Computing Systems*, May 28 - April 1, 1990, Paris, France.

Keywords: Real-Time Computing, Distributed Computing, Real-Time Operating Systems, Real-Time Benchmark, Real-Time Schedulability, Priority Inversion, Priority Inheritance.

This research was supported in part by the U.S. Naval Ocean Systems Center under contract number N66001-87-C-0155, by the Office of Naval Research under contract number N00014-84-K-0734, by the Federal Systems Division of IBM Corporation under University Agreement YA-278067, and by the SONY Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of NOSC, ONR, IBM, SONY, or the U.S. Government.

Keywords: Real-Time Computing, Distributed Computing, Real-Time Operating Systems, Real-Time Benchmark, Real-Time Schedulability, Priority Inversion, Priority Inheritance

Table of Contents

List of Figures

Abstract

In our definition of the Distributed Hartstone (DHS) benchmark, we extend the Hartstone real-time benchmark [14] to the distributed environment where communication and end-to-end scheduling become important. Traditional performance metrics for real-time systems do not tell the whole story, so we need a benchmark to gauge the performance for different areas of the operating system. Furthermore, we give a detailed description of the motivation and formulation of the benchmark to enable practitioners to implement the benchmark on a wide variety of systems. The benchmark is defined to be easily implemented on any system with basic real-time capability such as prioritized, preemptive scheduling and a real-time clock. Also, the tests are open-ended to allow comparisons on a wide variety of hardware.

The benchmark has been implemented on the ARTS Kernel [9], a distributed real-time testbed we have developed in the ART Project at Carnegie Mellon University. We give the benchmark results from the ARTS Kernel for different types of processor and communication scheduling algorithms. These results verify the usefulness of the benchmark suite in distinguishing between performance characteristics of these algorithms.

1. Introduction and Motivation

Synthetic benchmark programs have become popular for the measurement and comparison of different computer architectures and the compilers which generate code for them. The Whetstone [2] and Dhrystone [12] benchmarks are intended to measure performance for scientific calculations and systems programming. Recently, there has been a growing interest in defining a standard benchmark for real-time computer systems. The Rhealstone performance metrics [4] and the Hartstone benchmark [14] are two examples of this effort. The Rhealstone metrics concentrate on fine-grained real-time measures such as context switch time and interrupt latency as well as scheduling delay due to synchronization, and the Hartstone emphasizes total system performance.

Benchmarks provide a means to compare the performance characteristics of different computer systems, and we can classify benchmarks by the level at which they measure the system. We consider three levels: the computer architecture, the operating system primitives, and high-level operating system services (such as inter-process communication). The performance at each level depends not only on the performance of the lower levels, but also on the coordination of the lower levels in the current level. For instance, we show that even if the protocol engine [1] for a real-time protocol costs more in terms of overhead than a non-real-time protocol engine, the improved coordination of message traffic can result in a better higher-level performance measure (such as better schedulability) [3, 10]. Different applications designers might be interested in different levels of the system. For instance, a programmer using a workstation for scientific computing might be interested mostly in the performance of the architecture and of the compiler that is tuned to that architecture. The designer of a transaction system might be interested in the architecture level and also the performance of low-level operating system primitives for the file system. The designer of a robotics application might be interested in all three levels, particularly since effective operating system coordination of high-level services such as communication and synchronization will be critical to the timing correctness of the robotic system.

Current benchmark test programs are typically geared toward a particular class of applications such as scientific computing or systems programming. For example, the Whetstone [2] benchmark is a program which is meant to be representative of applications in scientific computing. The Dhrystone [12, 13] benchmark was subsequently defined to address the need for measuring the performance of systems programs which use records and pointers more than floating point calculations. Each of these benchmarks contain a mix of statements which are meant to represent the average application program. These benchmarks provide a good way of measuring the performance of the computer architecture and the compiler.

However, the performance of the operating system level is more difficult to quantify. Typical measures of operating system performance include the context-switch time, average throughput for tasks and messages, and average latency for tasks and messages. In real-time operating systems, the issue is further complicated. Additional

[1]"Protocol engine" refers here to the software module responsible for implementing the network protocols.

characteristics such as interrupt latency, worst case execution time for system primitives, and the scheduling delay due to resource contention and communication synchronization become important. Real-time systems depend on the operating system to support the predictability of the application and the schedulability of the system. Certain operating system constructs promote predictability, and we would like to be able to quantify these features of the system. The Rhealstone [4] standard lists several measures of operating system performance which are traditionally important for commercial, uniprocessor real-time operating systems. The Rhealstone measures include context-switch time, preemption time, interrupt latency, maximum scheduling delay due to synchronization, deadlock breaking time, and datagram throughput.

The Hartstone benchmark [14], developed as a joint effort between the ART Project and the Software Engineering Institute at CMU, is a further attempt to gauge the performance of real-time operating systems (or Ada runtimes) which must schedule harmonic and non-harmonic periodic activities and aperiodic activities. Also, the ability of the system to handle priority inversion [10] among synchronizing tasks is measured by the Hartstone suite. The synthetic workload for Hartstone is based on the Whetstone benchmark, so the architecture level performance is automatically taken into account. Also, since real-time systems tend to be more computationally oriented, the use of the Whetstone is appropriate.

None of these benchmarks, however, addresses the distributed real-time environment. In fact, many real-time applications are so computationally demanding that no single CPU will suffice, and we must connect multiple CPU's to handle the intense workload. Also, some applications require physically distributed processing, and so the ability of a system to perform in a distributed environment becomes very important. The Distributed Hartstone is an extension of the Hartstone work for the distributed real-time computing environment. We define real-time task sets which are designed to stress specific areas of the operating system such as communication latency, communication bandwidth, prioritized message handling at the transport level as well as the media access level, and preemptability of the communication processing. These characteristics are critically important to the performance and especially predictability of the real-time system. The performance of real-time systems is often measured only in terms of the time for various low-level operations. We contend that these measurements do not adequately describe the system. The performance of the real-time processor scheduling and communication network scheduling and the coordination between these scheduling domains largely determines how well the system can perform. And since more sophisticated scheduling algorithms may require more overhead for low-level operations, a system which offers better schedulability for its applications and thus better overall performance may not have the best times for low-level operations. A system which is leaner and faster in terms of low-level operations may not be capable of scheduling a task set to meet all of its deadlines. So we would like to factor all of these attributes into our overall measure of the system.

In Section 2 we discuss the areas of the system which are particularly important for distributed, real-time computing. In Section 3, we define several task sets which are intended to stress each of the areas identified in Section 2. And in Section 4, we give the performance of the ARTS Kernel on the Distributed Hartstone benchmark. In Section 5 we summarize our work.

2. Requirements for a Distributed Real-Time Benchmark

The objective of a real-time benchmark suite is to evaluate the performance of the system as a whole. The timing characteristics of primitive operations in the operating system are important, but so are the timing characteristics of the operations which depend on various subsystems working in harmony. With our Distributed Hartstone benchmark, we define task sets which attempt to stress areas of the system that we have identified as being critical for the predictable execution of real-time tasks. For instance, the coordination between scheduling in the processor domain and scheduling in the communication domain is critical in a distributed real-time operating system [10], and we concentrate on this area in several of our benchmark tests. In Section 4, we show that the Distributed Hartstone achieves these objectives by comparing the performance of different processor and communication scheduling algorithms using the ARTS Kernel.

2.1. Message Priority for Queueing

It is important to carry priority information with messages so that the processor scheduler of the receiving host can properly schedule the activity [10, 9, 11]. We would like to control the scheduling of the message traffic at each level of the protocol stack [5], and some sort of priority information is necessary. Also, the queueing of the messages between layers of the communication subsystem depends on priority information to avoid priority inversion [11]. Traditional time-sharing systems use FIFO queues to promote fairness and avoid starvation, but in real-time systems, service should be provided based on the time constraints of the activity[2]. A high frequency activity cannot afford to wait for arbitrarily many lower frequency or non-real-time activities to conclude their message activity, so the system must facilitate the timely delivery of such high frequency message traffic by promoting the high priority message to the head of the priority queue. This is an important area of the system to stress with a benchmark task set.

2.2. Preemptability of Protocol Engines

Preemptability is a key issue in analyzing the schedulability of a communicating or synchronizing real-time task set. Many real-time scheduling algorithms depend upon the preemptability of the task set, and the number and size of the critical regions must be kept to a minimum; otherwise the predictability of the task set is severely limited. We consider preemptability to be a multi-valued measure of the number and size of the critical regions in a computation rather than a binary feature. We have identified the communication protocol engines as an area where the computation can be quite time consuming and where the latency of high priority activities may be adversely affected if the protocol computation is not preemptable. If a low priority message is being handled by a protocol engine and a high priority message comes into the system, we are concerned about how long it will take to switch the protocol processing resources from the low priority message to the higher priority message. A real-time benchmark should be able to differentiate between systems with varying degrees of protocol engine preemptability.

2.3. Communication Latency

The end-to-end communication latency is an important characteristic of the distributed system, and so we should be able to distinguish between different systems in this area. This issue involves not only the latency due to the communication media but also time required for the other communication layers and the contention for resources including processor cycles. The analysis of real-time systems depends on the predictability of the underlying operating system. We must be able to bound the execution time, protocol handling, and scheduling delay for each task. A good system should be able to schedule activities including protocol processing and media access in a way that provides the smallest bounded communication latency to the highest priority tasks, and the benchmark should measure how well a system is able to coordinate these requirements over the entire task set.

2.4. Communication Bandwidth

Another important system attribute related to the communication performance is the bandwidth of the network. This is related to communication latency, but here we are interested in how much message traffic the communication system can handle instead of long it takes for a message to travel through the network. We include in this area the raw network bandwidth as well as the software processing needed to drive the network. As the network media become faster, the bottleneck for communication is increasingly in the operating system software and software protocol processing, and we need to measure how much overhead this processing consumes in the system.

2.5. Packet Priority for Network Contention

Using prioritized packets in a real-time network should allow for proper scheduling of the communication traffic. A real-time communication system should avoid priority inversion at the media access level, and frame level priority is one method which has been used to do this. Unfortunately, some systems do not use such sophisticated

[2]Note that if all real-time deadlines are met, then there is no deadlock or starvation

communication media, and the implementations of other networks with sophisticated designs sometimes artificially limit the perfomance of the contention algorithms [11]. The TI TMS380 chipset for the 802.5 token ring, for instance, reduces the number of frame priority levels from eight to four. This lack of priority granularity at the media access level exacerbates the problem of mapping from the processor priority levels to the frame-level priorities [7]. Also, this implementation uses FIFO queueing in the hardware, and these two factors affect the ability of the operating system to provide a prioritized message service. The benchmark should be able to detect these types of differences in the contention algorithms for the network media.

3. Distributed Hartstone Benchmark

Based on the requirements above which specify various areas of the system that a distributed real-time benchmark must stress, we have developed a series of task sets in the style of the Hartstone benchmark [14]. Each task set addresses a particular feature important for real-time operating systems. Most of the task sets for the distributed benchmark require two host machines, but one test has the number of host machines as a variable parameter. We will discuss the requirements for this particular test in more detail in Section 3.5. The distributed system should have a communication network with some end-to-end protocol available at the user level. The system should also have a real-time clock to verify the timing constraints for the task sets are met. We use the kilo-Whetstone (KWS) as a unit of computation derived from the popular Whetstone benchmark. A kilo-Whetstone is one thousandth of the one million cycle Whetstone benchmark. For more information on the kilo-Whetstone including the C code, see Appendix I. The Whetstone provides an appropriate synthetic load for real-time applications which tend to rely heavily on floating point computation.

We also designed our benchmark tests to be open-ended, as in the Hartstone suite. In each task set, we hold all factors constant except for one parameter which we vary. This parameter is typically a server execution time or the number of tasks of a particular category (such as low frequency tasks or high frequency tasks). These values can be increased almost indefinitely (within the resource requirements for new tasks in the case where the variable is number of tasks), so no matter how well a particular system perfoms, eventually, the breaking point at which deadlines are first missed should be reached yielding a Distributed Hartstone performance measure.

The task sets that we define to comprise the benchmark specify only the timing requirements for the task set and the unit of workload. This degree of generality enables implementation of the task sets on a wide variety of systems. The benchmark does not rely on features of any particular language or operating system. We have chosen timing constraints and the synthetic workload to be representative of the current requirements of typical real-time systems running on current microprocessor technology such as the Motorola MC68030 and the Intel i386[3].

In our specification of some task sets, we have indicated the priorities that might be assigned to the tasks in a rate monotonic framework [6], but if the scheduling algorithm under test is based on a dynamic priority scheme like the earliest deadline first policy or the least slack time first policy, this static priority assignment should be ignored. The priorities are given to make the intention of the task set transparent; they serve to focus attention on the potential problem each task set addresses. The particular priorities shown are relative priorities with no special significance in the choice of the numbers except that smaller numbers are higher priorities. Many real-time operating systems use a preemptive, prioritized scheduling model, so our priority assignment will enhance the consistency of the performance measurement among these types of systems. Of course, one of the primary objectives of research in real-time operating systems is to learn how to structure the contention for resources and how to manage this contention, so if a system is able to determine a better (given the individual system) assignment of priorities which enables the system to achieve high performance, then we take the results of these tests as the Distributed Hartstone values for the system. We do not intend to handicap a system by requiring the use of the current priority assignment; the timing constraints of the task set are the important attributes of the benchmark. In the following sections, we give the specifications for the task sets for each Distributed Hartstone series.

[3]The open-endedness of the benchmark suite should facilitate the benchmarking of machines with differing raw performance characteristics.

3.1. DSHcl Series: Communication Latency

The DSHcl Series is a (D)istributed, (S)ynchronized, and (H)armonic task set which tests the communication latency of the system. The base task set is patterned after the periodic, harmonic task set in the original Hartstone benchmark. However, the task set contains an additional remote server, and each of the tasks τ_1, ..., τ_5 sends a request to the server before consuming its own computation time. The server is non-preemptable, so a high frequency message may have to wait for the service of a lower frequency message to complete.

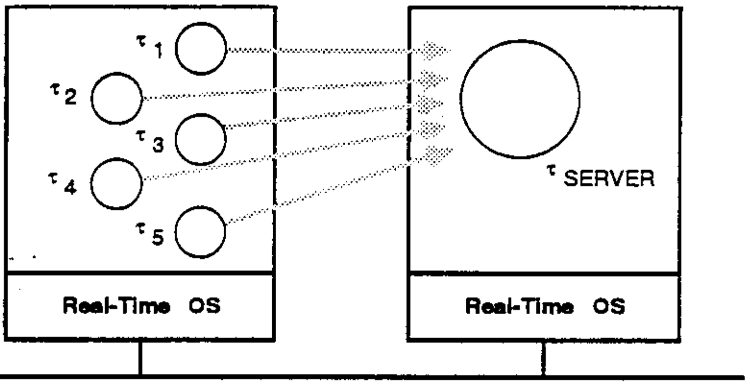

Figure 3-1: Five Clients with Single Server

Figure 3-1 shows the structure of the DSHcl Series task set. Communication invocations are illustrated by the arrows. The table specifies the timing requirements of the task set.

DSHcl Series Task Set		
Task	Workload	Period
τ_1	1 KWS	80 ms
τ_2	1 KWS	160 ms
τ_3	2 KWS	320 ms
τ_4	2 KWS	640 ms
τ_5	8 KWS	1280 ms
τ_{SERVER} (remote)	variable	N/A

We vary the computation time of the server to gradually squeeze the tasks until the size of the server combined with the time the request message is in transit causes deadlines to be missed. Figure 3-2 shows how an increase in the server execution time will, for a large enough execution time, cause the client to miss the deadline by increasing the delay associated with the contention for server time among the clients (called SS in the figure). If the scheduling is handled properly (by using priority inheritance in the communication and processor scheduling domains, for instance), a high frequency task will have SS $\leq$ (1+N) * ST where N is the number of high frequency tasks which may have service requests pending and ST is the Service Time. In the case of the above task set, N = 1 since we have only one high frequency task. This means that the Scheduling Delay, SD, in SS at most one Service Time. The worst case is that some low frequency task acquires the server and then a high frequency task requests the service but has to wait for the low frequency task to finish (up to ST) and then the task must wait for service for itself (another ST).

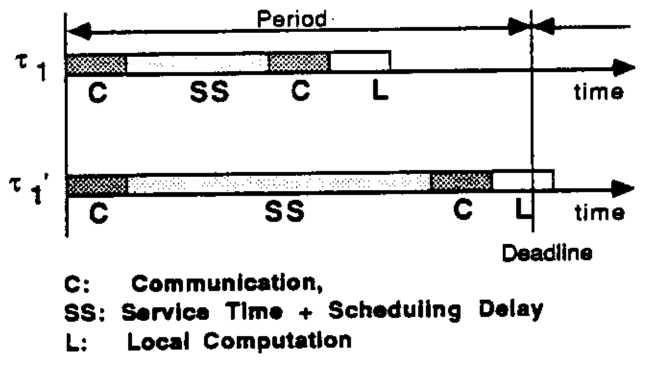

Figure 3-2: Client Task Time Requirements

3.2. DSHpq Series: Priority Queueing

The DSHpq Series task set is a (D)istributed, (S)ynchronized, and (H)armonic task set designed to test for priority queueing of communication packets. As in the previous series, the base task set is patterned after the periodic, harmonic task set in the original Hartstone benchmark. The task set contains an additional non-preemptable remote server, and each of the tasks $\tau_1, ..., \tau_5$ sends a request to the server before consuming its own computation time. We vary the computation time of the server to ascertain the value where the size of the critical region (non-preemptable service) causes deadlines to be missed.

The DSHpq Series is quite similar to the DSHcl Series except for the difference in granularity[4]. This reflects the fact that both tests are testing for the same sort of effect on different levels of operating system services. The DSHcl Series tests for fine grain differences in response time (from the client tasks' point of view) due to the difference in communication latency. The DSHpq Series tests for coarse grain differences in the response time due to ineffective queueing disciplines and poor scheduling on the remote host.

In Figure 3-1, we illustrate the task set and the communication paths.

DSHpq Series Task Set		
Task	Workload	Period
τ_1	1 KWS	160 ms
τ_2	1 KWS	320 ms
τ_3	2 KWS	640 ms
τ_4	2 KWS	1280 ms
τ_5	8 KWS	2560 ms
τ_{SERVER} (remote)	variable	N/A

[4]The fine-grained DSHcl uses shorter periods for the tasks and milliseconds to measure the server workload. The coarse-grained DSHpq uses longer periods for the tasks and kilo-Whetstones to measure server workload.

3.3. DSNpp Series: Preemptability of the Protocol Engine

The DSNpp Series task set is a (D)istributed, (S)ynchronized, and (N)on-harmonic task set designed to test the degree of preemptability of the protocol engines. The base task set is patterned after the periodic, non-harmonic task set in the Hartstone benchmark. The task set contains two remote servers; the high priority server can preempt the low priority server. We construct a task set composed of one high priority (high frequency) task and a variable number of low priority (low frequency) tasks. Task τ_1 sends a request to the high priority server at the beginning of its period, and each of the tasks τ_2, ..., τ_n sends a request to the low priority server at the beginning of its period before consuming its own computation time. The intention here load up the protocol engines with low priority work and then try to send a high priority message through. We vary the number of low priority tasks to determine how the preemptability of the low priority protocol processing affects the service of the high priority task. If the system is not careful about minimizing priority inversion at the protocol level (e. g., by increasing preemptability), then the high priority task will miss its deadline. If the system is careful about this, then even if we increase the number of low priority tasks, the high priority message will still be delivered, and the high priority task will meet its deadline. But eventually, if the number of low priority tasks is very large, even a system which is careful about this may begin to miss deadlines; at least the interrupt handler will execute for each low priority message, so small amount of priority inversion is unavoidable and may cause the high priority processing to be delayed.

We note that even if the protocol engine which is designed for increased preemptability incurs additional overhead, the overall schedulability of the communication system may be better than a more streamlined, less sophisticated system [3].

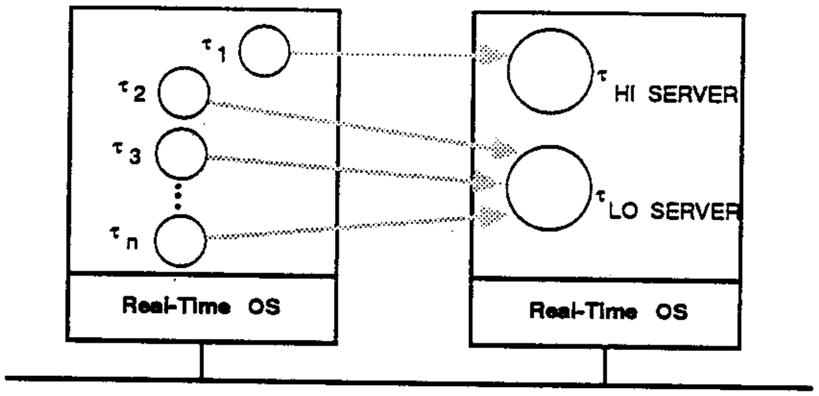

Figure 3-3: N Clients with Multiple Servers

Figure 3-3 shows the structure of the DSNpp Series task set. Communication invocations are illustrated by the arrows. The table gives the timing constraints of the task set.

DSNpp Series Task Set			
Task	Workload	Period	Priority
τ_1	1 KWS	50 ms	0
τ_2	1 KWS	5120 ms	1
...			
τ_n	1 KWS	5120 ms	1
$\tau_{\text{HI SERVER}}$ (remote)	1 KWS	N/A	0
$\tau_{\text{LO SERVER}}$ (remote)	0 KWS	N/A	1
Rate monotonic priorities are assigned to make the intention of the task set clear.			

The addition of more low frequency tasks into this task set serves to increase the on the load on the protocol engine at the server's host. Then the challenge for the system is to be able to obtain service from the server for a high frequency task so that the high frequency task as well as the lower frequency tasks can meet their deadlines. The degree of preemptability of the communication software determines the ability of the system to avoid priority inversion at the level of the protocol engine. Figure 3-4 illustrates the timing of a task set running on a system which does the protocol processing at the "software interrupt" priority level. In this case, the hardware interrupt from the network device initiates a software interrupt to continue processing the packet according to the appropriate protocol. This processing is done before making the processor available to user level tasks, and thus the protocol processing is not preemptable, since the high priority user level tasks cannot preempt the protocol engine. The effect is that low frequency message consume processor cycles which may cause high frequency tasks to miss deadlines; this is shown in the figure.

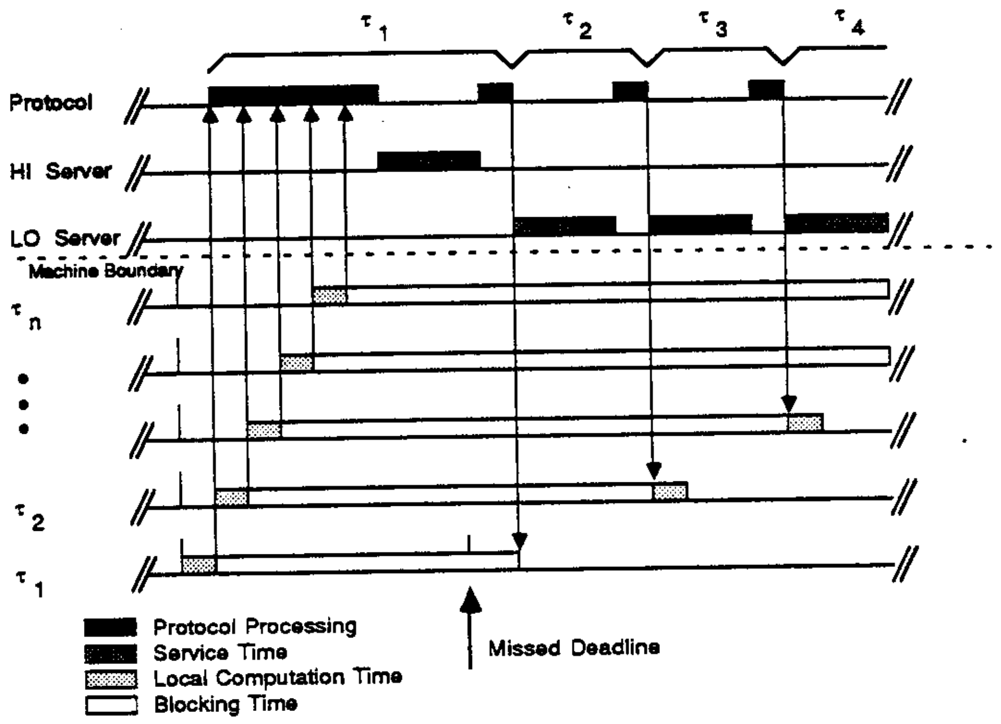

Figure 3-4: Non-preemptable Protocol Processing

In Figure 3-5, we have the same task set, but the underlying system will allow more preemptability in the protocol engine. For instance, the protocol processing of a low frequency task can be scheduled with the user level tasks at the appropriate priority. Then if the server inherits the priority of the message, the server servicing a high frequency message will compete with the protocol engine servicing a low frequency message, and the server will get the processor cycles. If a high frequency message is received, then the protocol processing for the low frequency message can be preempted in favor of the new message. By increasing the level of preemptability of the protocol engines and using priority inheritance in the server, we can maintain better schedulability. The figure shows that the protocol processing is deferred until the high priority server activity is finished.

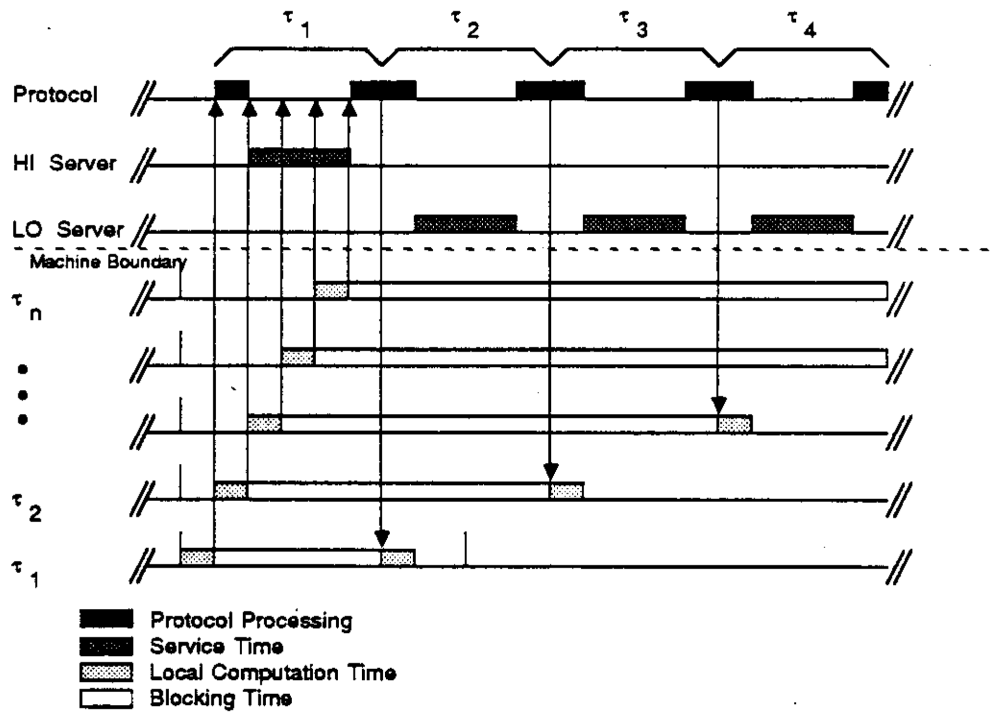

Figure 3-5: Preemptable Protocol Processing

3.4. DSHcb Series: Communication Bandwidth

The DSHcb Series task set is a (D)istributed, (S)ynchronized, and (H)armonic task set which tests the communication bandwidth. The base task set is patterned after the periodic, harmonic task set in the Hartstone benchmark. This task set contains a remote server that consumes no computation time; the server echoes the request back immediately as the result. Tasks $\tau_1, ..., \tau_n$ send requests to the server at the beginning of their periods and they consume no computation time. We increase the number of high priority tasks, which increases the load on the communication subsystem, until the first deadline is missed.

Figure 3-6 shows the structure of the DSHcb Series task set. Communication invocations are illustrated by the arrows. The timing constraints of the task set are shown in the table.

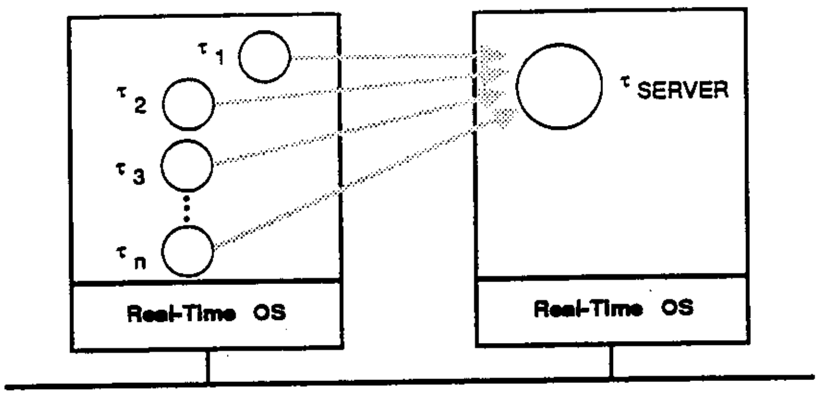

Figure 3-6: N Clients with Single Server

DSHcb Series Task Set			
Task	Workload	Period	Priority
τ_1	0 KWS	80 ms	0
τ_2	0 KWS	80 ms	0
...			
τ_{n-4}	0 KWS	80 ms	0
τ_{n-3}	0 KWS	160 ms	1
τ_{n-2}	0 KWS	320 ms	2
τ_{n-1}	0 KWS	640 ms	3
τ_n	0 KWS	1280 ms	4
τ_{SERVER} (remote)	0 KWS	N/A	N/A
Rate monotonic priorities are assigned to make the intention of the task set clear.			

3.5. DSHmc Series: Media Contention

The DSHmc Series task set is a (D)istributed, (S)ynchronized, and (H)armonic task set intended to stress the media contention algorithms. The task set consists of many low priority tasks which request service from a remote server and a single high priority task which also requires the service. We make the service time zero so that the preemptability of the server does not become an issue in this series. It is important that the low priority tasks be on different machines so that the network can be loaded more effectively[5]. The stress point of this task set is the high priority message. If the media contention algorithm is not effective in preventing priority inversion at high levels of network utilization, then the high priority message will not be handled in time and the high priority task will miss its deadline. We find the level of network utilization at which this occurs by loading the network more and more with low priority messages from the low priority tasks.

Figure 3-6 shows the structure of the DSHmc Series task set. Communication invocations are illustrated by the arrows.

[5]Alternatively, a host could be used to generate a synthetic network load of low priority messages.

DSHmc Series Task Set			
Task	Workload	Period	Priority
τ_1	0 KWS	80 ms	0
τ_2	0 KWS	1280 ms	1
τ_3	0 KWS	1280 ms	1
...			
τ_n	0 KWS	1280 ms	1
τ_{SERVER}	0 KWS	N/A	N/A
Rate monotonic priorities are assigned to make the intention of the task set clear.			

4. ARTS DHS Performance

We now give the benchmark results from a distributed, real-time operating system. ARTS is a distributed, real-time operating system testbed developed as part of the ART Project at Carnegie Mellon University. The testbed is for verification and evaluation of research in distributed, real-time technologies, and we currently have the system running on the SUN3, the SONY NEWS workstation, and the FORCE CPU-30 single card computer. We have several results from the ARTS implementation of the Distributed Hartstone benchmark (on the SUN3) which at once demonstrate the usefulness of the benchmark and verify the real-time capability of the ARTS Kernel.

The ART testbed we used for these results consists of a collection of SUN3 workstations connected by Ethernet and token ring networks. . We use the Ethernet for booting the kernel and the token ring to investigate real-time communication issues. Section 4.1 gives some measurements for message passing using different protocols on each of these media.

We first give the performance results for some primitive operations on ARTS to provide some background, and then we give the results of the Distributed Hartstone benchmark. To denote the result of a Hartstone test, we give an abbreviation of the name of the particular task set (such as DSHcl) followed by a number which indicates the highest value of the variable parameter for that task set achieved without missing any deadlines. For example, DSHcl-35 indicates that the DSHcl task line did not miss deadlines for values of the execution time of the server up to 35 units but when the server execution time was 36 units, the task set began missing deadlines. The type of unit used depends on the particular task set. The value could be synthetic workload units, units of time or number of clients.

Note that the DSHcl Series results from ARTS are measured in milliseconds. This is because a single kilo-Whetstone takes 4.2 ms to execute on a SUN3, but we require more resolution to differentiate between the two communication media we test in DSHcl. Therefore, we switch to a 1 ms workload in order to get the additional resolution on the result. Also, the DSNmc Series was not yet verified due to limitations in the physical testbed, but we are currently working to upgrade the testbed to allow us to run the DSNmc Series.

4.1. Basic Performance

The basic performance of the ARTS Kernel was measured by using SUN3/140s. Since the SUN3's timer chip (i.e. Intersil ICM7170) can provide up to 10 msec granularity, all measurements were done by repeating the same target actions over 10,000 times for each test.

We are tuning the system towards high predictability and schedulability, but the current measurements of the base time of the basic system functions are shown in Figure 4-1.

Null System Call:	0.04 ms
Real-Time Thread Context Switching:*	

single address space	0.22 ms
among mapped objects	0.22 ms
Instantiation and Destruction of an Object:**	
Thread Creation/Kill	0.64 ms
Lightweight Object Creation/Kill	4.24+.40*DS ms
Object Creation/Kill	7.98+.40*DS ms
Object Invocation:***	
Invocation, same address space	1.13+0.00062*MS ms
Invocation, another address space	1.41+0.00062*MS ms
Remote invocation, RTP/IE	8.97+0.0031*MS ms
Remote invocation, RTP/TR	10.24+0.010*MS ms
Remote invocation, VMTP/IE	9.12+0.0047*MS ms
Remote invocation, VMTP/TR	11.31+0.012*MS ms

* The SUN3 has eight hardware address spaces and we give the context switch time for threads which reside in the same address space and for threads which are in different objects in different address spaces.

** Object creation entails the allocation of resources such as memory regions for the object image. Thread creation results in a new thread of control in a previously instantiated object. DS indicates the data segment size in Kbytes.

*** Invocation times are the round trip message passing times between objects. We differentiate between invocation between objects in the same address space, objects in different hardware address spaces, and objects on different network hosts. RTP is the ARTS Real-time Transport Protocol and VMTP is the Versatile Message Transaction Protocol [1]. IE indicates Intel Ethernet and TR denotes Token Ring. In the remote invocation case, MS indicates the message size in bytes.

Figure 4-1: ARTS Basic Performance Statistics

4.2. DSHcl Series

The DSHcl Series measures the communication latency of the target system. The service time for each task is increased until deadlines are missed, so that means that the higher the result of the test in terms of service time, the lower the communication latency.

In the ARTS Kernel, we have two communication media with different latency as illustrated in Section 4.1. The Ethernet latency is lower than the token ring latency, so we would expect the result from the DSHcl Series to be higher for the Ethernet (in a dedicated environment) than the token ring (also in a dedicated environment). In fact this is true; we find that with the Ethernet, we can go as high as DSHcl-35 whereas the token ring can only go to DSHcl-34. The numbers 34 and 35 indicate the largest amount of server time (in ms) that the task could withstand without missing any deadlines for the token ring and Ethernet, respectively. We measure the server time in milliseconds in this case to provide a finer granularity than measuring computation time in kilo-Whetstones (KWS) which require 4.2 ms on a SUN3. This result verifies the ability of the DSHcl task set to differentiate between the performance of systems using network media with different performance characteristics.

4.3. DSHpq Series

The DSHpq Series is similar to the DSHcl Series in that it is intended to detect differences in the response time of the communication subsystem. However, the DSHpq Series is measures the response time with a larger grain than the DSHcl. The DSHpq Series is better tuned to differences in message priority handling and communication scheduling. Again, differences in performance are measured in terms of different remote service times (measured in KWS).

In the ARTS Kernel, we have the capability to send service requests at any particular priority from any task. So we can assign message priorities according to the task priorities assigned by the rate monotonic scheduling policy, or we can assign the same priority to all messages to get a FIFO queueing discipline for the message handling. With prioritized messages, the ARTS Kernel achieves DSHpq-18, but if the messages are not prioritized, the result is

DSHpq-13. This means that with prioritized messages, the server time could be set to 18 KWS, but without priorities on the messages, the limit on the server time was 13 KWS. This verifies that the benchmark task set can detect differences in performance of two different systems, one which uses priority information from network packets vs. a system which just does FIFO processing of network packets.

4.4. DSNpp Series

The DSNpp Series tests for preemptability in the protocol processing of the communication subsystem. We have a high priority task which periodically requests service time from a remote high priority server. We also have a number of low priority tasks which request time from a remote low priority server. We vary the number of low priority tasks, and a difference in the number of tasks for two communication scheduling algorithms indicates a difference in the preemptability of the protocol processing. If a particular algorithm allows the low priority protocol processing to preempt the high priority server, then the low priority tasks in this task set will be able to affect the timing characteristics of the high priority task. However, if the protocol processing of the low priority message is preempted in favor of the high priority service, then the number of low priority tasks will have little or no effect on the high priority task.

In the ARTS Kernel, we have two implementations of our protocol processing engine. One implementation uses a software interrupt mechanism which runs at a higher priority than any user tasks. The other implementation uses several prioritized worker threads to increase preemptability for messages with different priorities. The worker thread implementation is 10% slower than the software interrupt version, but it provides increased preemptability which is more important for real-time computing. The software interrupt version performed at the DSNpp-13 level. The result of this Series using the worker thread implementation is at least DSNpp-20, but due to a physical memory limitation, 20 is the largest number of threads we were able to test with. This test verifies that the DSNpp Series can detect distinguish between different degrees of preemptability in the network software.

4.5. DSHcb Series

The DSHcb Series tests the communication bandwidth by trying to maximize the number of message generated by the task set without spending any CPU time on additional computations such as workload for the periodic tasks or service time. We have several periodic tasks which send messages to a remote server and we increase the number of high priority tasks in the system. In this way, we push the utilization of the communication subsystem up without bringing preemptability or priority inversion handling into play; the competing tasks are mostly high priority tasks.

In ARTS, we find that by using the dedicated Ethernet at 10 Mbit/sec we can achieve DSHcb-17. We using a token ring at 4 Mbit/sec, we only get DSHcb-4. This reflects the difference in speed of the two networks and the difference in the performance of the hardware interface to the host computer and the device driver software.

4.6. DSHmc Series

Limitations in the current ARTS testbed prohibit us from loading the network enough to differentiate between communication media in terms of the priority inversion at the media access level. We are currently working on the solution to this problem. However, simulations have shown that the proper use of priorities in the media access level can promote predictable communication and substantially improve system response time [8].

5. Summary

We have described the Distributed Hartstone benchmark, an extension of the uniprocessor Hartstone benchmark for the distributed real-time environment. We attempt to improve on previous real-time benchmarks which concentrate only on issues such as context switch time, interrupt latency, and scheduling delay from synchronization. These issues are important in real-time systems, but a benchmark for the distributed real-time environment must also treat communication related issues such as the effective coordination between the processor scheduling domain and the communication scheduling domain. Also, we would like to evaluate operating system structures which should support the predictability and schedulability analysis for the system. Our Distributed

Hartstone measures system performance in the critical areas of communication latency and bandwidth, protocol preemptability, and priority queueing at the protocol and media access levels.

Future work includes the identification and evaluation of additional operating system structures which are critical for predictable real-time systems and the definition of task sets to stress these areas. Also, the performance of other types of systems such as pipe-lined systems should be examined to determine the types of tests that would be appropriate.

Acknowledgements

We would like to thank Dr. Nelson Weiderman and our reviewers for their useful comments on this paper and the members of the ART Project at CMU for their implementation effort.

References

[1] Cheriton, D. R.
VMTP: Versatile Message Transaction Protocol.
Technical Report, Computer Science Department, Stanford University, January, 1987.

[2] Curnow, H. J. and Wichmann, B. A.
A Synthetic Benchmark.
Computer Journal 19(1):48-49, January, 1976.

[3] Ishikawa, Y., Tokuda, H. and Mercer, C. W.
Priority Inversion in Network Protocol Module.
Proceedings of 1989 National Conference of the Japan Society for Software Science and Technology , October, 1989.

[4] Kar, R. P. and Porter, K.
Rhealstone -- A Real-Time Benchmarking Proposal.
Dr. Dobbs Journal 14(2):14-24, February, 1989.

[5] Le Lann, G.
Issues in Real-Time Local Area Networks.
Proceedings of Pacific Computer Communications Symposium , October, 1985.

[6] Liu, C. L. and Layland J. W.
Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment.
JACM 20 (1):46 - 61, 1973.

[7] Sha, L., Lehoczky, J. P. and Rajkumar, R.
Solutions for Some Practical Problems in Prioritized Scheduling.
In *IEEE Real-Time Systems Symposium.* December, 1986.

[8] Strosnider, J.K. and Marchok, T. and Lehoczky, J.
Advanced Real-Time Scheduling Using the IEEE 802.5 Token Ring.
Proceedings of the 9th Real-Time Systems Symposium , Dec., 1988.

[9] Tokuda, H. and Mercer, C. W.
ARTS: A Distributed Real-Time Kernel.
ACM Operating Systems Review 23(3), July, 1989.

[10] Tokuda, H., Mercer, C. W., Ishikawa, Y. and Marchok, T. E.
Priority Inversions in Real-Time Communication.
In *Proceedings of 10th IEEE Real-Time Systems Symposium.* December, 1989.

[11] Tokuda, H. and Mercer, C. W. and Marchok, T. E.
Towards Predictable Real-Time Communication.
In *Proceedings of Sixth IEEE Workshop on Real-Time Operating Systems and Software.* May, 1989.

[12] Weicker, R. P.
Dhrystone: A Synthetic Systems Programming Benchmark.
Communications of the ACM 27(10):1013-1030, October, 1984.

[13] Weicker, R. P.
Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules.
SIGPLAN Notices 23(8):49-62, August, 1988.

[14] Weiderman, N.
Hartstone: Synthetic Benchmark Requirements for Hard Real-Time Applications.
Technical Report CMU/SEI-89-TR-23 (ESD-89-TR-31), Software Engineering Institute, Carnegie Mellon University, June, 1989.

I. Kilo-Whetstone C Code

```
/*
 *      Whetstone benchmark in C.  This program is a translation of the
 *      original Algol version in "A Synthetic Benchmark" by H.J. Curnow
 *      and B.A. Wichman in Computer Journal, Vol  19 #1, February 1976.
 *
 *      Used to test compiler optimization and floating point performance.
 *
 *      Compile by:             cc -O -s -o whet whet.c
 *      or:                     cc -O -DPOUT -s -o whet whet.c
 *      if output is desired.
 *
 *      The counter values (module weights) have been adjusted from the one
 *      million instruction range to the one thousand range.  Unfortunately,
 *      some of the counts go to zero.
 *
 *      In this stage of development, ARTS kernel does not support trig
 *      functions, so this part of the test was also omitted.  We are
 *      currently working to add these functions in the ARTS library.
 */

#define ITERATIONS      10 /* 1 Million Whetstone instructions */

#include <math.h>

double          x1, x2, x3, x4, x, y, z, t, t1, t2;
double          e1[4];
int             i, j, k, l, n1, n2, n3, n4, n6, n7, n8, n9, n10, n11;

kwhets(nkwhets)
int     nkwhets;
{
    int kwhetcount;             /* loop counter */

    /* initialize constants */
    t   =   0.499975;
    t1  =   0.50025;
    t2  =   2.0;

    /*
     * set values of module weights
     * (measured in kilo-whets: some go to zero)
     */
    n1  =    0 * ITERATIONS / 1000;
    n2  =   12 * ITERATIONS / 1000;
    n3  =   14 * ITERATIONS / 1000;
    n4  =  345 * ITERATIONS / 1000;
    n6  =  210 * ITERATIONS / 1000;
    n7  =   32 * ITERATIONS / 1000;
    n8  =  899 * ITERATIONS / 1000;
    n9  =  616 * ITERATIONS / 1000;
    n10 =    0 * ITERATIONS / 1000;
    n11 =   93 * ITERATIONS / 1000;

    for (kwhetcount = 0; kwhetcount < nkwhets; kwhetcount++) {

        /* MODULE 1:  simple identifiers */
        x1 =  1.0;
        x2 = x3 = x4 = -1.0;
        for(i = 1; i <= n1; i += 1) {
            x1 = ( x1 + x2 + x3 - x4 ) * t;
            x2 = ( x1 + x2 - x3 - x4 ) * t;
            x3 = ( x1 - x2 + x3 + x4 ) * t;
            x4 = (-x1 + x2 + x3 + x4 ) * t;
        }
#ifdef POUT
        pout(n1, n1, n1, x1, x2, x3, x4);
#endif

        /* MODULE 2:  array elements */
        e1[0] =  1.0;
        e1[1] = e1[2] = e1[3] = -1.0;

        for (i = 1; i <= n2; i +=1) {
            e1[0] = ( e1[0] + e1[1] + e1[2] - e1[3] ) * t;
            e1[1] = ( e1[0] + e1[1] - e1[2] + e1[3] ) * t;
            e1[2] = ( e1[0] - e1[1] + e1[2] + e1[3] ) * t;
            e1[3] = (-e1[0] + e1[1] + e1[2] + e1[3] ) * t;
        }
#ifdef POUT
        pout(n2, n3, n2, e1[0], e1[1], e1[2], e1[3]);
#endif

        /* MODULE 3:  array as parameter */
        for (i = 1; i <= n3; i += 1)
            pa(e1);
#ifdef POUT
        pout(n3, n2, n2, e1[0], e1[1], e1[2], e1[3]);
#endif

        /* MODULE 4:  conditional jumps */
        j = 1;
        for (i = 1; i <= n4; i += 1) {
            if (j == 1)
                j = 2;
            else
                j = 3;

            if (j > 2)
                j = 0;
            else
                j = 1;

            if (j < 1 )
                j = 1;
            else
                j = 0;
        }
#ifdef POUT
        pout(n4, j, j, x1, x2, x3, x4);
#endif

        /* MODULE 5:  omitted */

        /* MODULE 6:  integer arithmetic */
        j = 1;
        k = 2;
        l = 3;
        for (i = 1; i <= n6; i += 1) {
            j = j * (k - j) * (l -k);
            k = l * k - (l - j) * k;
            l = (l - k) * (k + j);

            e1[l - 2] = j + k + l;              /* C arrays are zero based */
            e1[k - 2] = j * k * l;
        }
#ifdef POUT
        pout(n6, j, k, e1[0], e1[1], e1[2], e1[3]);
#endif

#ifdef SIN_OKAY
        /* MODULE 7:  trig. functions */
        x = y = 0.5;
        for(i = 1; i <= n7; i +=1) {
            x = t * atan(t2*sin(x)*cos(x)/(cos(x+y)+cos(x-y)-1.0));
            y = t * atan(t2*sin(y)*cos(y)/(cos(x+y)+cos(x-y)-1.0));
        }
#ifdef POUT
        pout(n7, j, k, x, x, y, y);
#endif
#endif SIN_OKAY

        /* MODULE 8:  procedure calls */
        x = y = z = 1.0;
        for (i = 1; i <= n8; i +=1)
            p3(x, y, &z);
#ifdef POUT
        pout(n8, j, k, x, y, z, z);
#endif

        /* MODULE9:  array references */
        j = 1;
        k = 2;
        l = 3;
        e1[0] = 1.0;
        e1[1] = 2.0;
        e1[2] = 3.0;
        for(i = 1; i <= n9; i += 1)
             p0();
#ifdef POUT
        pout(n9, j, k, e1[0], e1[1], e1[2], e1[3]);
#endif

        /* MODULE10:  integer arithmetic */
        j = 2;
        k = 3;
        for(i = 1; i <= n10; i +=1) {
             j = j + k;
             k = j + k;
             j = k - j;
             k = k - j - j;
        }
#ifdef POUT
        pout(n10, j, k, x1, x2, x3, x4);
#endif

        /* MODULE11:  standard functions */
        x = 0.75;
        for(i = 1; i <= n11; i +=1)
             x = sqrt( exp( log(x) / t1));
#ifdef POUT
        pout(n11, j, k, x, x, x, x);
#endif
    }
}

pa(e)
double e[4];
{
    register int j;

    j = 0;
 lab:
    e[0] = (  e[0] + e[1] + e[2] - e[3] ) * t;
    e[1] = (  e[0] + e[1] - e[2] + e[3] ) * t;
    e[2] = (  e[0] - e[1] + e[2] + e[3] ) * t;
    e[3] = ( -e[0] + e[1] + e[2] + e[3] ) / t2;
    j += 1;
    if (j < 6)
         goto lab;
}

p3(x, y, z)
double x, y, *z;
{
    x  = t * (x + y);
    y  = t * (x + y);
    *z = (x + y) /t2;
}

p0()
{
    e1[j] = e1[k];
    e1[k] = e1[l];
    e1[l] = e1[j];
}

#ifdef POUT
pout(n, j, k, x1, x2, x3, x4)
int n, j, k;
double x1, x2, x3, x4;
{
    printf("%6d%6d%6d  %5 e  %5 e  %5 e  %5 e\n",
            n, j, k, x1, x2, x3, x4);
}
#endif
```