The Journal of Real-Time Systems, 4, 353-382 (1992)

Hartstone Uniprocessor Benchmark: Definitions and Experiments for Real-Time Systems

NELSON H. WEIDERMAN
Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213-3890

NICK I. KAMENOFF
Department of Computer and Information Sciences, Fordham University, Bronx, New York 10458-5198

Abstract. The purpose of this paper is to define a series of requirements and associated experiments called the Hartstone Uniprocessor Benchmark (HUB), to be used in testing the ability of a uniprocessor system to handle certain types of hard real-time applications. The benchmark model considers the real-time system as a set of periodic, aperiodic (sporadic), and synchronization (server) tasks. The tasks are characterized by their execution times (workloads), and deadlines. There are five series of experiments defined. They are, in order of increasing complexity, PH (Periodic Tasks, Harmonic Frequencies), PN (Periodic Tasks, Nonharmonic Frequencies), AH (Periodic Tasks with Aperiodic Processing), SH (Periodic Tasks with Synchronization), and SA (Periodic Tasks with Aperiodic Processing and Synchronization). The general stopping criteria of the experiments is defined as follows: Change one of the following four task set parameters: number of tasks, execution time(s), blocking time(s), or deadline(s) until a given task set is no longer schedulable, i.e., a deadline is missed. The derivation of the Hartstone experiments from one static scheduling algorithm (Rate Monotonic) and one dynamic scheduling algorithm (Earliest Deadline First) is presented. Because of its high-level application view of the underlying hardware and real-time system software the Hartstone experiments can be used for fast prototyping of real-time applications. Implementation of such benchmarks is useful in evaluating scheduling algorithms, scheduling protocols, and design paradigms, as well as evaluating real-time languages, the tasking system of compilers, real-time operating systems, and hardware configurations.

1. Introduction

Time-critical applications in embedded systems have become extremely commonplace. These applications occur in the monitoring and control of systems in areas as diverse as financial information services, industrial processes, transportation, defense, and medicine. These applications typically have time constraints (deadlines) as part of their specified requirements. There has generally been a distinction made between *hard* real-time tasks, in which processes must meet specified deadlines in order to satisfy the requirements of the system, and *soft* real-time tasks, where a statistical distribution of response times is acceptable (Liu and Layland 1973).

A real-time system comprises hardware, real-time system software, and application software. The real-time system software can be a real-time operating system, a real-time executive, real-time kernel, or the runtime system of a programming language that supports tasking. For example, Ada (United States Dept. of Defense 1983) is a language designed to support real-time programming whose runtime system provides such real-time services as timing, task scheduling, and task synchronization and communication.

The goal of real-time system design and implementation is to construct systems that are guaranteed to meet all hard deadlines and that minimize the response time for all soft deadlines. This is a challenge that has frequently not been met. Systems that fail to meet their deadlines under certain circumstances because their designers lack a complete understanding of all the scheduling issues are all too common (Stankovic and Ramamritham 1988). The methodology for real-time design and implementation is often to build a system and test its performance with little or no scheduling theory to corroborate the results. Seldom are there are guarantees for how the system will perform when transient loadings occur or when the system is corrected, upgraded, or adapted. More recently the methodology has been shifting toward a more scientific basis in which the goal is to provide *a priori* that deadlines will be met and that response times will be minimized (Mok 1983; Sha and Goodenough 1989; Stankovic and Ramamritham 1989; Levi, Tripathi, Carson and Agrawala 1989). Research in this area has concentrated on various scheduling theories, scheduling algorithms, and communications protocols (Stankovic and Ramamritham 1988).

As real-time system technology advances the operating system designer is asked, "How good is this real-time configuration (hardware plus system software) in comparison to others (the competition)?" To this question the application developer always puts the additional question: "Can the specific configuration satisfy my real-time requirements?" The Hartstone Uniprocessor Benchmark (HUB) proposed here introduces complex system requirements for the evaluation of real-time configurations. The HUB implementation is completely independent of the system software and the underlying hardware.

2. Evaluation Technologies for Real-Time Systems

In order for real-time system design to be based on sound practical and theoretical principles, it is necessary to provide a strong coupling between scheduling theory and evaluation technology for performance measurement. This coupling provides several advantages. First, it allows the theory to be constructed based upon reasonable assumptions about measurable parameters. Second, it facilitates the verification of the theory using empirical measurements. Finally, it can uncover unexpected effects.

The evaluation technology for measuring of real-time performance has taken several forms. We distinguish three approaches for the performance evaluation of real-time systems: *fine-grained benchmarks, simulations, and application-oriented benchmarks*. The last one is the most complete and conclusive. The proposed benchmark here is based on this approach.

2.1. Fine-Grained Benchmarks

The fine-grained benchmarks address the low-level operations common to real-time systems: context switching, interrupt handling, and various system calls. Typically these benchmarks *try to isolate a specific real-time feature and then measure it*. The efforts connected with the measurement method can be considerable in regard to the extra code that has to be written, and/or the hardware to be used (for example logic analyzer).

The fine-grained benchmarks have drawbacks. There are difficulties in getting accurate and repeatable results from software as well as difficulties in isolating separate features

to be measured. Typically, many of the features that have to be measured take only a small number of microseconds on modern hardware. Because software timers often do not have the required precision, the operation must be repeated a number of times to get an aggregate measurement consistent with the precision of the timer. Unfortunately, when this is done, there are a number of problems concerned with the subtraction of the overhead of the loop, differing time requirements depending on the placement of code in memory, the presence and status of a cache, and the presence and status of pipelining in an architecture. These and other problems have been documented in (Altman and Weiderman 1988). One partial solution to these problems is to obtain measurements through hardware tools, such as logic analyzers, that have very high precision (in the nanosecond range).

One attempt to provide technology in this area is the Rhealstone Real-Time Benchmark which was developed by (Kar and Porter 1989) and subsequently implemented in C (Kar 1990).

2.2. *Simulation*

At the other end of the spectrum is simulation. Here, *a model is developed and programmed in an appropriate language.* The simulation can be used to predict the performance of a real-time system based on certain assumptions about its design, including the processing requirements and the scheduling algorithms. An example of extensive testing of scheduling algorithms is given by (Locke 1986).

In the case of simulation there is the *problem of validating the model.* A simulation may be misleading if the model does not reflect reality in the appropriate detail and adequate precision. For example, if the simulation models the context switch overhead as a constant, and in the real system it is a variable depending on the number of tasks or the nature of the context switch, then the conclusions drawn from the study may be erroneous. Another drawback of simulation is that it rarely takes into account all the hardware and software components of a real-time system. Detailed simulations are often prohibitively expensive and require a lot of coding efforts. The main question that remains is the *model precision.* If the hardware is not carefully modelled, or not modelled to the detail of the actual system, the results of the simulation may not be borne out when the actual system is built. Another problem is obtaining accurate values (from fine-grained benchmarks for example) to use in the model.

In summary, the results of the simulation are only of limited use for the application developer. They can lead to some important conclusions for the system to be implemented, but nevertheless the simulation is not the real world.

2.3. *Application-Oriented Benchmarks*

The Hartstone model described here represents the application-oriented approach for evaluation of real-time systems. It is a *synthetic* benchmark consisting of a set of concurrent activities (tasks). While Hartstone is not meant to replace fine-grained benchmarking or simulation, its primary advantage is that it produces results in the context of a real-time system by *presenting itself as an application to the system.* It does this at a cost much less

than a full simulation (because the system used is the actual one and need not be simulated) while producing results that are more useful and representative than fine-grained benchmarks. It provides an evaluation standard by which entire systems can be compared.

Synthetic benchmarks are programs which, rather than performing any useful function, are constructed using instruction frequencies thought to be typical of some class of representative programs. Typical synthetic benchmarks are Whetstone (Wichmann 1988) and Dhrystone (Weicker 1984, 1988). Whetstone is meant to be representative of scientific numerical calculations and is heavily weighted toward floating point computation. Dhrystone was developed later and was intended to reflect the use of features in modern programming languages (e.g., record and pointer data types). Both the Whetstone and Dhrystone report their results in terms of work performed per time unit. In the case of Whetstone it is a defined unit called Whetstones per second and in the case of Dhrystone it is called Dhrystones per second. Whetstone was originally written in Algol and Dhrystone was originally written in Ada, but each has been translated into several other computer languages.

Rather than measuring throughput, as is the case with both Whetstone and Dhrystone, the purpose of a Hartstone implementation is to *measure the breakdown point of a real-time system*. The breakdown point is defined as the point at which the computational and scheduling load causes a hard deadline to be missed. This breakdown point can be expressed as a *utilization rate*. The utilization rate is defined as the throughput rate achieved in the presence of overhead and timing constraints (i.e., waiting for the next period of activation to begin), divided by the throughput rate achieved in the absence of overhead and timing constraints. Scheduling theory shows (Liu and Layland 1973) that the breakdown point can reach 100% utilization with convenient problem timing constraints (harmonic task frequencies) and the assumption of zero overhead (i.e., when overhead is assumed to be included in the computational workload). The extent to which breakdown is less than 100% must be attributed to factors such as overhead, delays, or timing anomalies of the real-time system.

Another significant difference between Hartstone and other synthetic benchmarks is that Hartstone, as expressed in this paper, is a system requirement rather than an implemented program. It bares open, not only the implementation language, but also the scheduling algorithms and the system synchronization primitives to be used. This flexibility allows good practical comparisons to be made among languages, scheduling algorithms, system software and hardware platforms. Only the most important aspects of real-time system (e.g., meeting deadlines) is specified by the Hartstone requirement.

In summary, the application-oriented benchmarks (in this case Hartstone) have the advantages of providing the application developer with figures of merit for the complete behavior (hardware and system software) of the real-time system. Because the benchmark resembles a small real-time application, the efforts for its implementation are manageable. On the other hand, the applicability of the results is not a problem because the actual system software and target hardware are used.

3. The Hartstone Real-time Model

Just like a simulation, a set of requirements for real-time system evaluation must start with a model for real-time computation. The goal in this effort is to choose a model that is

characteristic of real-time applications and simple in concept, but complex enough to represent a wide variety of application scenarios. This is done by choosing three basic types of tasks with parameters that allow a number of variations. Obviously, the closer an application conforms to the Hartstone real-time model, the more useful will be the Hartstone benchmark in evaluating real-time systems on which the application will eventually be run.

In the Hartstone real-time model the application in the general case can be conceived as a *set of periodic, aperiodic, and synchronization (server) tasks* (Sha and Goodenough 1990; Borger, Klein and Veltre 1989).

3.1. Periodic Tasks

A periodic task is in general characterized by a period, execution time (workload), phase, and deadline. In the Hartstone model, a periodic task is initiated at fixed intervals in time and has as its deadline the end of the period (which is also the beginning of the next period). The *workload* of a periodic task is its execution time per period and will be considered to be constant for the duration of the experiment. Constant execution time can be justified by a large number of applications with this characteristic. For those applications that do not have constant execution time, the worst case execution time or the average execution time can be used depending on the requirements of the user.

The phase of a periodic task is its starting time relative to other periodic tasks. Since the worst case for phasing occurs when all tasks are started at the same time (Liu and Layland 1973), we will assume that all phases are zero so that a task's phase does not have to be considered as a parameter of the model.

In summary, a periodic task in the Hartstone model is characterized by its period and its execution time. The deadline is defined at the end of the period. All periodic tasks are defined to have hard deadlines.

3.2. Aperiodic Tasks

An aperiodic task is characterized by an execution time, a sequence of interarrival times, and a deadline. As with the periodics, the computation time for each invocation is constant. In the Hartstone model the aperiodic request arrivals are simulated using a generator for interarrival times as described in Section 8.4. The mean of the interarrival times can be chosen so as to model the worst case interarrival scenario.

The deadlines for aperiodics can be soft or hard depending on the particular experiment. A deadline for a hard aperiodic (also called a *sporadic* task) is defined as the next aperiodic arrival. Hard deadline aperiodics must have a minimum interarrival time specified so that the system can guarantee completion and avoid queueing of requests. A soft deadline is operationally not a deadline at all since failure to meet the deadline does not cause a system failure. For soft deadline aperiodics, the optimization criterion is the minimization of the response time from the arrival of the request to the completion of the computation. Two of many possible optimization criteria are to minimize the maximum response time and to minimize the average response time. Alternatively, one could maximize the percentage of aperiodic requests that meet the soft deadline. Queueing of the aperiodic requests is required because of the arbitrarily small interarrival times.

3.3. Synchronization Tasks

A synchronization or server task executes the behalf of *client* tasks which can be either periodic or aperiodic tasks. Its activity must be conducted sequentially and completely for one client before it can undertake the same activity for another client. Since it is acting on behalf of other tasks, it has no deadline. It is one model for synchronizing concurrent activities. A typical example is access to a serial shared resource (critical section). The server task must *employ in its implementation at least one of the synchronization primitives or mechanisms* provided by the real-time system software.

In the Hartstone model, the server task is characterized by its execution time which consists of two parts: one that must be completed before the client is released (execution of the critical section), and one that can be executed after the client is released (independent execution of the server task).

3.4. Summary

In summary, the tasks in the Hartstone real-time model are characterized as having hard or soft deadlines. A hard deadline task has to perform in a timely fashion. A soft deadline task is required to respond as fast as possible, but is not constrained to finish by specific times. The Hartstone experiments include periodic tasks with hard deadlines, aperiodic tasks which may have hard or soft deadlines, and server tasks with no deadlines.

4. Hartstone Uniprocessor Benchmark Series

This paper defines the operational concept for requirements for five series of experiments of increasing complexity. The *Hart* of *Hartstone* was derived from the fact that the experiments were to address the *Ha*rd *R*eal-*T*ime application domain. The *stone* follows the legacy of the Whetstone, Dhrystone, and Rhealstone benchmarks.

The five series of experiments are named as follows:

PH Series: Periodic Tasks, Harmonic Frequencies
PN Series: Period Tasks, Nonharmonic Frequencies
AH Series: Period Tasks with Aperiodic Processing
SH Series: Period Tasks with Synchronization
SA Series: Periodic Tasks with Aperiodic Processing and Synchronization

In an earlier paper (Weiderman 1989) only the PH Series of experiments was defined. Subsequently, the first series of experiments was implemented in the Ada programming language (Donohoe, Shapiro, and Weiderman 1990a) and the implementation was used to test the characteristics of seven different Ada compilers and runtime systems running on four different embedded systems configurations (Donohoe, Shapiro, and Weiderman 1990b). This experimentation was conducted by the Real-time Embedded Systems Testbed (REST) project and the Software Engineering Institute. The initial implementation of the Hartstone

PH Series benchmark in Ada has been distributed, at the request of users, internationally to over 100 sites. Cooperative efforts are under way to rewrite the program in other languages, such as C.

In this paper the all five series of experiments are defined in detail. The experiment definitions given are general and not restricted to any real-time system software (operating system, kernel, real-time language, or runtime system) or hardware. This means that the hardware configuration (the type, speed, and architecture of the processor and memory) will influence the results of the experiments, but not the implementation issues for the benchmark. Nor are the requirements meant to be biased toward any particular scheduling mechanism or policy. The requirements can be used equally well to test cyclic round-robon, rate monotonic, earliest deadline first, or any other policies for scheduling tasks. Hartstone is a benchmark that dynamically tests a real-time system, including the hardware and the full scheduling abilities of the system software.

The HUB experiments are similar to application software that will provide results (figures of merit) for the scheduling ability of the hardware and real-time system software configuration under test. For example, if some characteristic of the configuration such as CPU speed or main memory speed is changed, or if another version of an executive is used, then the experiments will deliver different results.

Because of the application character of the experiments, the benchmarks can be viewed additionally as a tool for rapid prototyping of hard real-time applications. By modifying the workload, the number of tasks, and the frequencies of the tasks to reflect the application, the user could create a real-time system profile similar to the eventual application system and answer various hypothetical questions about the performance of that application. The use of Hartstone for rapid prototyping is not addressed in this paper, but is a topic for further research.

For each requirements series there is defined a *baseline task set* that includes the number and characterization of tasks by their workloads, deadlines, and periods or interarrival times. Each experiment starts with the baseline task set for the specific series in which all deadlines must be met for a given length of time. In moving from one experiment step to the next, one parameter of that task set is varied while the other parameters remain constant.

An experiment continues a step at a time, increasing the computing requirements, until a predefined stopping condition is reached. The most frequently applied stopping condition is a missed deadline, but it could also be the exceeding of some limit on the average response time for aperiodic requests or the exceeding of a given utilization rate. The purpose of all experiments is to put the underlying system (hardware and system software) under stress until it can no longer meet its assigned work requirement. In each case, the result of the experiment is the value of the parameters when the stopping condition is reached and the utilization achieved just before system breakdown.

We wish to emphasize that the results of the Hartstone experiments are figures of merit which characterize the underlying hardware and the real-time system software. When carefully interpreted, they are useful from a user's point of view for evaluating a complete system in context. The experiments are not appropriate for pure comparisons of only one component of a real-time system unless that component can be changed while the other components remain constant.

Additional work has been done on the Hartstone Distributed Benchmark (HDB) series (Kamenoff and Weiderman 1991). These experiments are designed for the hard real-time distributed systems. The experiments integrate the processor scheduling domain and the communication scheduling domain, and they take into consideration the end-to-end scheduling of messages.

5. Derivation of the Hartstone Experiments

In this section the Hartstone experiments will be justified and their generality will be shown by deriving them from one static scheduling algorithm, Rate Monotonic (RM), and one dynamic scheduling algorithm, Earliest Deadline First (EDF) (Liu and Layland 1973).

As mentioned already, many real-time application can be conceived as a set of periodic, aperiodic, and synchronization tasks. A periodic task for the hard real-time requirements is characterized by its period T_i, execution time C_i, and deadline D_i. The deadline of a periodic task job is typically considered to be at the end of its period ($D_i = T_i$), which is the case with the RM and EDF scheduling algorithms.

For aperiodic tasks with hard deadlines (sporadic tasks), Sprunt, Sha and Lehoczky (1989) have shown for the RM scheduling that a *sporadic task can be associated with a sporadic server and in terms of schedulability can be treated as a periodic task* with a period equal to the minimum interarrival time IAT_i, and an executive time C_i. Furthermore Ghazalie (1990) has shown that the sporadic server model can be adapted to improve the response times with EDF scheduling, as it is known to do with RM scheduling. In both cases (RM and EDF) the basic idea is to associate a period deadline with each replenishment of server execution time (Baker 1991).

Associating an aperiodic task with a sporadic server enables this task to be scheduled within the RM or EDF scheduling paradigms. Thus for a set of n independent periodic and aperiodic (sporadic server) tasks can be written in first approximation:

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq W(n) \leq 1 \quad (RM) \tag{1}$$

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq 1 \quad (EDF) \tag{2}$$

where

$$D_i = \begin{cases} T_i\text{: Period for periodic tasks} \\ IAT_i\text{: Mean or Minimum Interarrival Time for periodic tasks} \end{cases}$$

$W(n)$ decreasing function of the number n of tasks.

Liu and Layland (1973) showed that the RM algorithm is optimal among all fixed priority schemas, and the EDF algorithm is optimal among all dynamic priority schemas. These algorithms are optimal in the sense that they can produce a feasible schedule if any other priority assignment of the same scheme is able to do so.

In the case where the tasks have to synchronize and/or communicate by using nonpreemptable critical sections it can happen that a higher priority task can be blocked by a lower priority task which is executing the same critical section required by the higher priority task. This phenomenon known as *priority inversion* can significantly reduce the schedulability of a given task set. To keep the priority inversion bounded for both (RM and EDF) scheduling algorithms, Priority Ceiling Protocols (PCP) were designed. In essence, these protocols are assigned ceiling priorities to the synchronization tasks. Sha, Rajkumar, and Lehoczky (1987) generalize the result in (1) for the RM scheduling, showing that the jobs are schedulable if:

$$\forall k,\ k = 1,\ \ldots\ n,\ \sum_{i=1}^{k} \frac{C_i}{D_i} + \frac{B_k}{T_k} \leq W(k) \leq 1 \qquad (RM) \tag{3}$$

Here B_k is an upper bound on the duration of blocking that the kth job may experience due to resources held by lower priority jobs.

For the EDF algorithm Chen and Lin (1990) extended (2), using the dynamic PDP for semaphore locking, to:

$$\sum_{i=1}^{n} \frac{C_i + B_i}{D_i} \leq 1 \qquad (EDF) \tag{4}$$

Here B_i is worst-case blocking time of the ith job.

Although expressions (3) and (4) can be used for quantitative evaluation of scheduling results, we want to use them only as qualitative basis for the derivation of the Hartstone experiments. There are *four parameters* in these expressions: tasks' deadlines D_i tasks' execution times C_i tasks' blocking times B_k or B_i, and number of tasks n. The main idea behind all experiments is to *change one of these parameters by keeping the other three constant until a deadline is missed* (i.e., the task set is no longer schedulable). In the following we will discuss the derivation of the experiments based on the above cited parameters. The full set of the Hartstone experiments is shown in Figure 1.

In the table in Figure 1 the up arrow means increase the value, and the down arrow means decrease the value, of the given parameter in the experiment until the stopping criteria is reached. For the better understanding of the table note that the PH series consists of five periodic harmonic tasks P1 to P5. P5 is the highest frequency task (shortest period). The PN series consist of five periodic, but nonharmonic tasks with frequencies which are very close to those of the PH series. The AH series consist of the five periodic tasks of the PH series plus one sporadic task (aperiodic with hard deadline), and one aperiodic task with soft deadline. The SH series consist of the five periodic tasks of the PH series plus one sporadic task (aperiodic with hard deadline), and one aperiodic task with soft

Experiments / Series	Expr1	Expr2	Expr3	Expr4	Expr5	Expr6
PH	$T_5 \downarrow$	$T_1 - T_5 \downarrow$	$C_1 - C_5 \uparrow$	$n_{Task_3} \uparrow$		
PN	$T_5 \downarrow$	$T_1 - T_5 \downarrow$	$C_1 - C_5 \uparrow$	$n_{Task_3} \uparrow$		
AH	$IAT_{sp} \downarrow$	$C_{sp} \uparrow$	$C_1 - C_5 \uparrow$	$n_{Task_3} \uparrow$	$IAT_{ap} \downarrow$	$C_{ap} \uparrow$
SH	$C_{s-within} \uparrow$	$C_{s-outside} \uparrow$	$C_1 - C_5 \uparrow$	$n_{Task_3} \uparrow$	$n_s \uparrow$	
SA	$IAT_{sp} \downarrow$	$C_{s-within} \uparrow$	$C_1 - C_5 \uparrow$	$n_{Task_3} \uparrow$		

Figure 1. Full set of HUB experiments.

deadline. The SH series consist of the five periodic tasks of the PH series plus one synchronization (server) task. The SA series consists of the five periodic tasks of the PH series plus the aperiodic tasks of the AH series, plus the synchronization task of the SH series.

Full details of the parameters of each experiment of each series are given in Sections 6 through 10.

5.1. *Varying Tasks' Deadlines D_i*

For periodic tasks D_i represents the end of a task period. By decreasing the task period of one periodic task (task P5, T_5), or the entire set (Tasks P1–P5, $T_1 - T_5$) and keeping all other parameters constant, experiments 1 and 2 are derived for the PH and PN series.

For hard aperiodic (sporadic) tasks D_i represents the minimum Interarrival Time IAT_{sp}, and for soft aperiodic tasks the mean Interarrival Time IAT_{ap}. By decreasing the minimum or mean interarrival time and keeping all other parameters of the task set constant, experiment 1 and 5 for the AH series and experiment 1 for the SA series are derived.

5.2. *Varying Tasks' Execution Time C_i*

By increasing the execution time C_i of all periodic tasks ($C_1 - C_5$) and keeping all other parameters of the task set constant, experiment 3 for all test series (PH, PN, AH, SA, and SA) is derived.

By increasing the execution time of the hard aperiodic task C_{sp}, or the soft aperiodic task C_{ap} and keeping all other parameters of the task set constant, experiments 2 and 6 for the AH series are derived.

If the synchronization task has an executable part which is outside of the critical section $C_{s\text{-}outside}$, then for this part the task can be considered as an independent task and depend-

ing on its current priority and length of code the task can decisively influence the schedulability of the task set. This case is represented with experiment 2 in the SH series. In this experiment the execution time of the synchronization task outside the critical section is increased until a deadline is missed.

5.3. Varying the Blocking Time B_k or B_i

By increasing the length of the critical section in the synchronization task $C_{s\text{-}within}$ and keeping all other parameters of the task set constant, experiment 1 of the SH series, and experiment 2 of the SA series are derived. As one can see, under the dynamic priority ceiling protocol (4) the worst-case blocking delay B_i, which is directly proportional to the length of the critical section $C_{s\text{-}within}$, can be treated as an extra execution time of a periodic task in addition to its normal execution time C_i.

5.4. Varying the Number of Tasks n

By increasing the number of periodic tasks n_{Task_3} and keeping all other parameters in the task set constant, experiment 4 for all test series (PH, PN, AH, SH, and SA) is derived.

In experiment 5 of the SH series the number of synchronization tasks is increased. Referring to expression (3) and (4) this can be interpreted as an increase of the blocking delays B_k and B_i for the client tasks P1–P5.

5.5 Some Terminology

In the following experiment descriptions we distinguish between *workload* and *scheduling load*.

An increase in the workload is achieved by an increase in the execution times of the tasks. An increase in the scheduling load is achieved by increasing the number of tasks or the frequency of the tasks. An increase in the frequency of tasks is equivalent to the decrease of the periods for periodic tasks, or the mean interarrival times for aperiodic tasks.

The increase in the scheduling load leads to an increase in the workload rate as well.

6. Definition of the PH Series

The objective of the PH series is to provide test requirements with a task set that is purely periodic and harmonic. This series is the simplest of the five test series. The attraction of the harmonic case is that with most scheduling policies (Rate Monotonic Scheduling, Cyclic Executives or Earliest Deadline First), 100% utilization can be achieved in the absence of overhead.

6.1. *Baseline Task Set*

The baseline task set consists of five periodic harmonic tasks. Harmonic means that each task's frequency is an integer multiple of the frequency of every lower frequency task. The tasks are independent in the sense that their execution is not synchronized; they do not communicate with each other. The tasks are logically concurrent and compete for processor time to complete their assigned computation requirement. Each periodic task is characterized by a frequency or period, computation time (workload), and a deadline. Frequencies, F, are expressed in Hertz (cycles per second), the reciprocal of the frequency is the task's period, T, in seconds per cycle $T = 1/F$.

A task workload is a fixed amount of work, which must be completed within a task's period. The workload of a Hartstone periodic task is provided by a variant of the well known composite synthetic Whetstone benchmark (Curnow and Wichmann 1976) called Small Whetstone. Small Whetstone has a main loop which executes one thousand Whetstone instructions (a Whetstone instruction is roughly equivalent to one floating point operation), or one *Kilo-Whetstone*. A Hartstone task is required to execute a specific number of Kilo-Whetstones within its period. The *workload rate* of a task is equal to its workload multiplied by the task's frequency. The workload rate is measured in Kilo-Whetstone instructions per second, or *KWIPS*. It is used as a measure of utilization. The *deadline* for completion of the workload is the beginning of the task's next period.

Table 1 gives the set of 5 periodic harmonic tasks with their frequencies (and corresponding periods), workloads, and workload rates:

Table 1. PH baseline task set.

Task	Frequency (Hertz)	Period (Millisec)	Workload (Kilo-Whets)	Rate (KWIPS)
P1	2	500.00	32	64
P2	4	250.00	16	64
P3	8	125.00	8	64
P4	16	62.50	4	64
P5	32	31.25	2	64
			Total	320

6.2. *Experiments*

Four experiments are defined for the PH Series. Each of the experiments is constructed so as to preserve the harmonic nature of the task set throughout the steps of the experiment.

Experiment PH-1. Starting with the baseline task set, the frequency of the highest frequency task (Task P5) is increased by the amount equal to the frequency of Task P4 (16 Hz) until a deadline is missed. The frequencies of other tasks and the workload of all tasks do not change. This experiment tests a system's ability to handle fine granularity of time and to rapidly switch between tasks. The experiment may fail prematurely (before fully utilizing

the processor) if the period of the highest frequency task approaches the timing resolution of the system's timing services.

Experiment PH-2. Starting with the baseline set, the frequencies of all tasks (P1–P5) are increased by multiplying the original frequencies by 1.1, then 1.2, then 1.3, and so on for each new test until a deadline is missed. The workloads of all tasks remain unchanged. The scaling preserves the harmonic frequencies. This experiment tests the system's ability to handle an increasing but balanced workload and scheduling load.

Experiment PH-3. Starting with baseline task set, the workload of each task is increased by 1, then 2, then 3, and so forth Kilo-Whetstone per period for each new test continuing, until a deadline is missed. This experiment tests the system's ability to handle an increasing workload without changing the scheduling load. Because only the workloads are changed while the frequencies remain unchanged, the amount of context switching should remain constant under most scheduling policies.

Experiment PH-4. Starting with the baseline task set, new tasks with the same frequency and workload as the *middle* task, Task P3, of the baseline set are added until a deadline is missed. The frequencies and the workloads of the baseline set do not change. This experiment tests the system's ability to handle an increasing scheduling load due to more tasks being added to the system. If tasking overhead is a function of the number of tasks because of the search algorithm for queues, it should become apparent in this experiment.

6.3. Application Domain

In general, periodic tasks (harmonic and nonharmonic) represent real-time applications that monitor several banks of sensors and store or display the results with no user intervention or interrupt-processing requirements. Other applications include control devices that are required to deliver periodic outputs. According to Information Theory, sampling a process at a rate of at least twice its maximal frequency yields complete knowledge of the process. This establishes periodic tasks as the basis of real-time systems (Gafni 1989).

It should be noted that the Periodic Harmonic case is ideally suited to the cyclic executive model that is so prevalent as an early model for real-time systems in defense and other application areas. This is because time can be neatly broken down into slices based on the least common divisor of the task periods (the *major cycle*). Each of the major cycles is divided into *minor schedules* or *frames* into which all the processing time must be allocated. A task's computation time may fit completely into a single frame or may have to be allocated across several frames. Each major cycle is then repeated indefinitely. Cycle executives can handle the nonharmonic case only by modifying the frequencies of the tasks or by having extremely large major cycles. For handling aperiodics, the cyclic executive must use a polling model. All synchronization must be handled in an *ad hoc* manner. Thus while the PH Series is amenable to solution with a cyclic executive implementation, the other Hartstone Series are more appropriate for implementations that use the more modern priority-based preemptive schedulers. Baker and Shaw (1988) give a good summary of the cyclic executive model together with an analysis of some of its strengths and weaknesses.

7. Definition of the PN Series

The objective of the PN series is to provide test requirements with a set of tasks that are purely periodic, but nonharmonic. The PN series retains the simplicity and structure of the PH series. When nonharmonic frequencies are chosen, scheduling theory can be used to show that extremely high utilization rates may no longer be possible. In fact, with optimally assigned static priorities, it has been shown that worst case utilization approaches 69.3% (ln 2) for a larger number of tasks (Liu and Layland 1973). In other words, the upper bound for processor utilization can be as small as 69.3%. For five tasks the least upper bound is 74.3% in the worst case. Since the Hartstone task set has not been constructed to be the worst case, its utilization bound will be somewhat greater than 74.3%.

7.1. Baseline Task Set

The baseline task set consists of five periodic nonharmonic tasks. Nonharmonic in this case means that each task's frequency is an integer multiple of none of the frequencies of the lower frequency tasks. The tasks are independent in the sense that their execution need not be synchronized; they do not communicate with each other. To provide a better comparison with the results of the PH series, the task frequencies of the PN series were chosen to be the closest prime number to the task frequencies of the PH Series. The workloads were adjusted so that the total workload rate of the entire task set approximated the baseline workload rate for the PH Series.

The five task nonharmonic baseline is defined in Table 2.

Table 2. PN baseline task set.

Task	Frequency (Hertz)	Period (Millisec)	Workload (Kilo-Whets)	Rate (KWIPS)
P1	3	333.33	21	63
P2	5	200.00	13	65
P3	7	142.86	9	63
P4	17	58.82	4	68
P5	31	32.26	2	62
			Total	321

7.2. Experiments

For the PN series four experiments are defined. They are directly analogous to those of the PH series. The experiments are constructed so as to maintain the basic nonharmonic nature of the task set, but some harmonics appear in some of the steps as the frequencies are increased. Hence the task set does not always remain purely nonharmonic as defined above. Because of the nonharmonic character of the task set, it is expected that the observed breakdown utilization rates of all PN experiments will be lower than those for the analogous PH series experiments. This is due to the lower theoretical upper bounds for utilization as described above.

Experiment PN-1. Starting with the baseline task set, the frequency of the highest frequency task (Task P5) is increased by the amount equal to the frequency of Task P4 (17 Hz) until a deadline is missed. The frequencies of other tasks and the workloads of all tasks do not change. This experiment tests the system's ability to handle fine granularity of time and to rapidly switch between processes in the context of nonharmonic frequencies.

Experiment PN-2. Starting with the baseline task set, the frequencies of all tasks (P1–P5) are increased by multiplying the original frequencies by 1.1, then 1.2, then 1.3, and so on for each new test until a deadline is missed. The workloads of all tasks do not change. This experiment tests the system's ability to handle an increasing but balanced workload and scheduling load in the context of nonharmonic frequencies.

Experiment PN-3. Starting with the baseline task set, the workload of each task is increased by 1, then 2, then 3 and so forth Kilo-Whetstones per period until a deadline is missed. This experiment tests the system's ability to handle an increasing workload without changing the scheduling load.

Experiment PN-4. Starting with baseline task set, new tasks with the same frequency and workload as the middle task, Task P3, of the baseline set are added until a deadline is missed. The frequencies and workloads of the baseline task set do not change. The frequencies of the duplicated task are obviously harmonic with one another, but this strategy is used to maintain comparability to PH-4. This experiment tests the system's ability to handle an increasing scheduling load.

7.3. Application Domain

This series represents the same application domain as the PH series. However, it is often the case, particularly in cyclic executives, that the frequency of the monitoring tasks is chosen based on a convenient frequency for the executive rather than based on a frequency dictated by the real-world problem. For example, if a natural frequency of the problem is 31 Hz, the implementation may execute that function at a higher rate such as 60 Hz because the major cycle time of the executive executes at that rate. When priority-based preemptive schedulers are used, the frequencies can be chosen to match the natural periods dictated by the application requirements. Thus the PN series is important for determining what, if any, penalty there might be in choosing frequencies in this way.

8. Definition of the AH Series

The objective of the AH series is to extend the PH series by introducing aperiodic processing requirements. Aperiodic processing requirements are defined as having processing request intervals that are irregular. Whereas periodic requests come at fixed intervals, aperiodic requests come at intervals which are unknown *a priori*. Sometimes, however, they can be characterized stochastically. To simulate these aperiodic requests, we define distributions

and pseudorandom sequences from which numbers are generated for the interarrival times. This will ensure a repeatable set of experiments independent of the configuration on which the experiments are run.

Aperiodic processing demands have been characterized as being of two types. The first type represents processing demands that have no explicit deadlines. Many times they represent less urgent processing that can go on in the background as compared to periodic processing. Rather than having fixed, firm deadlines, these tasks are measured by their response times which real-time systems designers attempt to minimize. We will henceforth refer to these tasks simply as *background tasks*. An implication of this type of processing is that aperiodic requests for this type of service must be queued so that if there is an overload of such requests, the requests are handled in some appropriate order and not lost.

The second type of aperiodic processing has explicit deadlines. Aperiodic processing with deadlines usually represents urgent processing that may take precedence over periodic processing. It can frequently be associated with sporadic incoming data in which any loss of data may be considered a catastrophic failure. A deadline for an aperiodic task in the Hartstone model is based on the time of the arrival of the next processing request. Thus the only way to guarantee that deadlines can be met is to place a minimum value on these interarrival times. These aperiodic tasks will henceforth be referred to as *sporadic tasks* as defined originally by Mok (1983).

8.1. Baseline Task Set

The baseline task set consists of a periodic processing load and an aperiodic processing load. The periodic processing load is the same as for the PH series. In particular, there are five periodic tasks that are each characterized by a frequency (period) and workload. The aperiodic processing load consists of two tasks, a background task and a sporadic task.

The background and sporadic tasks will be characterized by:

1. The workload expressed in KWI.
2. The sequence of interarrival times taken from a given generating function.

The sequence of interarrival times in a real world application may or may not follow some distribution. The interarrival times for the Hartstone model could be placed in a table, but since it is necessary to have a very large number of these times, it is more convenient to have a generating function for them. This generating function will generate random variates from a known distribution with a known mean which is defined fully in the appendix. As stated above, the aperiodic task has no deadline and will be measured by its response time while the effective deadline for the sporadic task is the arrival time of the next request.

In the case of the sporadic task, the interarrival time will consist of two parts: a fixed part representing the minimum interarrival time and a variable part representing the rest of the interarrival time. The fixed part will be defined as one half of the mean interarrival time.

The baseline aperiodic processing load is defined such that the sporadic task has mean interarrival time equal to the period of the highest frequency periodic task. The background task has mean interarrival time equal to the period of the lowest frequency periodic task.

Table 3. AH baseline task set.

Task	Frequency (Hertz)	Period (Millisec)	Workload (Kilo-Whets)	Rate (KWIPS)
P1	2	500.00	32	64
P2	4	250.00	16	64
P3	8	125.00	8	64
P4	16	62.50	4	64
P5	32	31.25	2	64
S1	32	31.25	1	32
B1	2	500.00	16	32
			Total	384

Note: For the sporadic and aperiodic (S1 and B1) the frequency column represents mean frequency, the period column represents the mean interarrival time, and the rate column represents the mean workload rate.

The workloads are set so that the workload rate of each of the aperiodics is one half of the workload rate of each of the periodics. Series AH uses the PH baseline task set (Table 1) and adds two additional aperiodic tasks, one sporadic and one background. The modified baseline is shown in Table 3.

In Table 3, P1 through P5 are the periodic tasks, B1 is the background task and S1 is the sporadic task. The mean interarrival time for S1 is equal to the period of task P5 (31.25 milliseconds) and the workload is equal to half the workload of task P5. The mean interarrival time for B1 is equal to the period of task P1 (500 milliseconds) and the workload is equal to one half of the workload of task P1.

In order to make this baseline (and subsequent experiments) explicit, and portable to all machines, it will be necessary to define not only the distribution, but the algorithm for generating random variables from this distribution. This algorithm must be specified in such a way that it will give the same series of interarrival times on all implementations on which it is run. This can be done by making some restrictive assumptions about the arithmetic capabilities of the underlying hardware. This will be discussed in more detail in Section 8.4.

8.2. *Experiments*

In the AH series, we are interested not only in whether the task set meets all its deadlines, but also in the *response time* of the background task. We define the response time to be the time interval between the arrival of the request and the completion of the request. Response times, particularly those for aperiodic tasks handling user interfaces, can be as important as the correct functionality of the real-time system.

It is desirable in this set of experiments to use the interrupt handling capabilities of the configuration and not to simulate interrupts. One of the prime purposes of this experiment is to incorporate the interrupt processing overhead of the underlying system. In most cases, this will require that interrupts be generated by an external source, but in some cases it

may be necessary to generate them internally by software. In particular, this strategy would be desirable for a portable software implementation. The overhead of generating the interarrival times must be measured relative to the minimum 1 KWI workload and taken into consideration when evaluating the results of the experiment.

There are six experiments defined for the Aperiodic Series. The first two experiments increase the demands of the sporadic task. The first decreases the mean interarrival time and the second increases the workload. The third and fourth experiments increase the demands of the periodic task set. The last two experiments increase the demands of the background task, first by decreasing the mean interarrival time and then by increasing the workload.

Experiment AH-1. Starting with the baseline task set, the average interarrival time for the sporadic task is decreased until some deadline (periodic or sporadic) is missed. The workload of the sporadic task and the parameters of the periodic and background tasks do not change. The method of decreasing the interarrival time will be to increase the corresponding average *frequency* by the same frequency as the middle frequency periodic task P3 (8 Hz) in each step of the experiment. Thus, if the original average frequency is 32 Hz, producing a starting average interarrival time of 31.25 milliseconds, then the next average frequency will be 40 Hz (32 Hz + 8 Hz) with average interarrival time of 25 milliseconds. The minimum interarrival time will always be one half of the average period and the other half of the interarrival time will be taken from the pseudorandom exponential distribution generated as described below. This experiment will be terminated when some deadline is missed.

Experiment AH-2. Starting with the baseline task set, the workload of the sporadic task is increased until a deadline (periodic or sporadic) is missed. The average interarrival time of the sporadic task and the parameters of the periodic and background tasks do not change. The method of increasing the workload is by adding 1 KWI to the workload in each step of the experiment. Thus the workload will progress from 1 to 2 to 3 and so forth from the original baseline.

Both Experiments AH-1 and AH-2 test the system's ability to absorb an increasing transient load. In Experiment AH-1 the emphasis is on the transient scheduling load (task switching). In Experiment AH-2 the emphasis is on the transient computational load.

Experiment AH-3. Experiment three is analogous to experiment three of the PH series. Starting with the baseline task set, the workloads of the periodic tasks are increased uniformly until a deadline (periodic or sporadic) is missed. The workloads and interarrival times of the background and sporadic tasks do not change. The method of increase is by adding 1 KWI to each periodic workload at each step of the experiment. In the baseline task set given in Table 3, the workload rate will increase by 62 KWI per step (32 + 16 + 8 + 4 + 2 = 62—see column Workload of Table 3).

Experiment AH-4. Experiment four is analogous to experiment four of the PH series. Starting with the baseline task set, the periodic scheduling load is increased by adding new tasks until a deadline (periodic or sporadic) is missed. The workloads and interarrival times of the background and sporadic tasks do not change. The method of increasing the

scheduling load is to add one new task (the middle periodic task P3) at each step. In the case of the baseline task set of Table 3, the workload rate will increase as well by 64 KWI per step.

Experiment AH-5. Experiment five is analogous to experiment one of this series. Starting with the baseline task set, the average interarrival time of the background task is decreased until a deadline (periodic or sporadic) is missed. The workload of the background task and the parameters of the sporadic and periodic tasks do not change. The method of decreasing the interarrival time will be to increase the corresponding average frequency by 8 Hz in each step of the experiment. In this experiment and the next we have a situation with two possible failure modes. The first is that the periodics or the sporadic may miss deadlines. The second is that the queue of requests and the response times increase without limit as the experiment proceeds. If no deadlines are missed the experiment is stopped when the requested utilization first exceeds 100%.

Experiment AH-6. Experiment six is analogous to experiment two of this series. Starting with the baseline task set, the workload of the background task is increased until a deadline is missed or until the requested utilization exceeds the maximum utilization. The interarrival time of the background task and the parameters of the sporadic and periodic tasks do not change. The method of increasing the workload is by adding 16 KWI to the workload in each step of the experiment. This means that the workload will progress from 16 to 32 to 48 and so forth. This increases the workload rate (32 KWIPS) at the same rate as experiment two because of the low frequency of the background task for this experiment. The failure mode and stopping criteria are the same as experiment 5. In properly designed implementations, one might expect that neither AH-5 nor AH-6 should fail due to deadlines being missed. Since the background task has no deadlines, the response time should continue to increase so that the hard deadline tasks can meet the timing constraints. On the other hand, the possibility of missed deadlines by the periodics should not be ruled out because servicing aperiodic interrupts at ever increasing rates may well add enough overhead to interfere with periodic processing.

8.3. Application Domain

An interrupt-driven model is often used instead of polling when the occurrence of events is not deterministic. Sometimes sporadic events are used in controlled processes where the control parameters are changing in a wide spectrum of frequencies (Gafni 1989). Polling can be unnecessarily expensive when the events occur rarely or at changing frequencies so that the polling consumes unnecessary cycles from a scarce resource. In these cases an interrupt-driven model is chosen. Examples include communication systems, where messages may come at unpredictable times, and weapon systems where targets are acquired at unpredictable times. In operating systems, input and output systems are mainly interrupt-driven. Both overhead and response time can be evaluated with this series of experiments.

8.4. Generating Random Numbers

The random numbers generated for the experiments must be the same random numbers on each machine on which the experiments are conducted. This requires a repeatable sequence of random numbers (pseudorandom numbers) and an algorithm for converting the sequence of random numbers into a sequence of numbers that is exponentially distributed according to the parameters of the distribution. The exponential distribution is chosen because it is the most common distribution for simulating random events (Knuth 1969). Such a distribution may not capture the *burstiness* of some aperiodic activities in the real world, but it is typically used as an approximation.

It is not sufficient to rely on any random number generator that happens to be available. To ensure that we have achieved the objective of repeatability, we have provided the random number generator, the means of generating the exponential random variates based on the random sequence, and the initial set of the first 50 random variates (see Table A-2) generated by this sequence so that the user may have confidence that the method is the one that is prescribed.

The random number generator should work efficiently on 16-bit computer architectures. This will restrict the period that can be obtained, but will permit these sets of experiments to be run on more restricted computer architectures that are often still used in real-time applications. This will necessarily restrict the period of the random number generator to $32768(2^{15})$. The numbers will actually vary from 0 to 32767 which is the range of positive numbers on a 16-bit machine and which permits multiplication of two such numbers on a 32-bit machine without overflow. We will assume also 32-bit floating point arithmetic (hardware or software) to permit the generation of the real variables. The details of this are presented in Appendix A.

9. Definition of the SH Series

The objective of the SH series is to provide test requirements for a set of tasks that require synchronization among tasks. Synchronization between two tasks is required when they have to use a shared resource or when the action of one depends on the completion of an action by the other. A shared object can be any hardware or software resource. An action dependency can be a producer-consumer relationship. To ensure strict sequential access to a shared object or to coordinate the activities of two or more tasks, there are synchronization primitives such as monitors, semaphores, event flags, locks, signals, etc. In Ada the synchronization is achieved through a synchronization task, which in its implementation must embody at least one of the synchronization primitives. The synchronization task, known as a server task, guarantees the serialized and protected access to the shared resource and executes on behalf of a client task requiring access to this resource.

The SH series gives an opportunity in the Hartstone set of requirements to investigate the properties of the real-time systems with respect to priority inversion and blocking.

9.1. *Baseline Task Set*

The baseline test set consists of five periodic harmonic tasks with workloads and frequencies identical to those for the PH series. In addition there is a 6th *server* task with which each periodic *client* task must synchronize once per period. During the synchronization time the server task performs an amount of work equal to 1 KWI while synchronized with the client task. The server task also has a workload (initially zero) which it can perform after the synchronization with any client task. The workloads of the periodic tasks have been each reduced by 1 KWI per period so that the total workload with the server activations is equal to the original workload of the PH series. To minimize the priority inversion and keep the blocking times for the client tasks bounded, the server task would normally be assigned the highest priority in the baseline set. The baseline task set for the SH series is shown in Table 4.

Table 4. SH baseline task set.

Task	Frequency (Hertz)	Period (Millisec)	Workload (Kilo-Whets)	Rate (KWIPS)
P1	2	500.00	31	62
P2	4	250.00	15	60
P3	8	125.00	7	56
P4	16	62.50	3	48
P5	32	31.25	1	32
Y1	62	Server	(1,0)	62
			Total	320

Note: For the Server Task (Y1) the frequency column is the sum of the frequencies of the client tasks. The two components of the server workload are the time for the critical section (synchronization) and the time for independent (outside the critical section) execution.

9.2. *Experiments*

For the SH series there are five experiments defined as follows:

Experiment SH-1. Starting with the baseline task set, the amount of work to be performed by the server task during synchronization is increased by 1 KWI per period for each step until a deadline is missed. The parameters of the periodic tasks (frequencies and individual workloads) do not change. This experiment tests the system's ability to schedule tasks when their blocking times are increased.

Experiment SH-2. Starting with the baseline task set, the amount of work to be performed by the server task following the synchronization with the periodic task is increased by 1 KWI until a deadline is missed. This experiment tests the effect of decreasing availability of the server task on the task set. The decreasing availability can be interpreted as an equal additional preemption time to the periodic (client) tasks.

Experiment SH-3. Starting with the baseline task set, the amount of work to be performed by the periodic tasks is increased by 1 KWI until a deadline is missed. This experiment resembles experiment PH-3 of the PH series. The difference is the added synchronization of the periodic tasks with the server task. The experiment tests the system's ability to handle an increased workload when the amount of blocking time does not change. It is possible to draw a comparison between the result of this experiment and experiment PH-3 of the PH series. The differences between the results should represent primarily the effect of synchronization in addition to the extra overhead needed for the server task.

Experiment SH-4. Starting with the baseline task set, new periodic tasks, each with the same frequency and workload as the middle periodic task P3, are added until a deadline is missed. The parameters of the periodic and synchronization tasks do not change. This experiment resembles PH-4 of the PH series. The difference is the added synchronization of the periodic tasks with the server task. The experiment tests the system's ability to schedule additional tasks which at the same time increases the blocking time in the system due to synchronization. A comparison between the results of this experiment and experiment PH-4 of the PH series should show the reduced schedulability of periodic tasks due to synchronization.

Experiment SH-5. Starting with the baseline task set, new synchronization tasks, each with the same workload as Task Y1, are added until a deadline is missed. Each of the periodic tasks has to be synchronized with each of the synchronization tasks. The experiment tests the system's ability to schedule client tasks when they have to go through multiple servers.

9.3. Application Domain

The designed experiments can be used to test the effectiveness of the synchronization primitives and synchronization algorithms whose main goal is the avoidance of deadlocks and keeping the blocking times to a minimum. All shared hardware (main memory, I/O devices, CPUs, etc.) and software (code, data) resources require, for their proper use, the implementation of synchronization primitives and algorithms. In general, whenever a real-time system manages shared resources of any kind, there will be synchronization requirements. By introducing synchronization to the set of experiments, the Hartstone model becomes more realistic for the purpose of evaluating and prototyping real systems.

10. Definition of the SA Series

The objective of the SA series is to provide test requirements for a set of tasks that combine various characteristics of the previous series of tests. This series is the most complex and demanding for a system to handle. It contains periodic and aperiodic tasks as well as synchronization among tasks.

10.1. Baseline Task Set

The baseline task set consists of the union of the task sets for the AH and SH series. This includes the five periodic harmonic tasks with workloads and frequencies equivalent to those for the PH series, the two aperiodic tasks introduced in the AH series, and the server task introduced in the SH series. As before, one of the aperiodics has soft deadlines and the other hard deadlines. The server task is called by each periodic task once per period. During the synchronization time, the server task performs an amount of work equal to 1 KWI. The baseline task set is shown in Table 5.

Table 5. SA baseline task set.

Task	Frequency (Hertz)	Period (Millisec)	Workload (Kilo-Whets)	Rate (KWIPS)
P1	2	500.00	31	62
P2	4	250.00	15	60
P3	8	125.00	7	56
P4	16	62.50	3	48
P5	32	31.25	1	32
S1	32	31.25	1	32
B1	2	500.00	16	32
Y1	62	Server	(1,0)	62
			Total	384

Note 1: For the sporadic and background (S1 and B1) tasks, the frequency column represents mean frequency, the period column represents the mean interarrival time, and the rate column represents the mean workload rate.
Note 2: For the Server Task (Y1) the frequency column is the sum of the frequencies of the client tasks.

10.2. Experiments

The SA series has four experiments defined. The first experiment deals with the sporadic (hard aperiodic) task, the second with the synchronization (server) task, and the third and fourth with the periodic tasks.

Experiment SA-1. Starting with the baseline task set, the mean interarrival time of the sporadic task is decreased until some deadline is missed. The workload of the sporadic task and the parameters of the periodic tasks, synchronization task, and the (soft) aperiodic task do not change. This experiment resembles AH-1 of the AH series. The difference is the added synchronization of the periodic tasks with the server task. The experiment tests the system's ability to handle aperiodic transient scheduling overload when synchronization and background aperiodic processing are present.

Experiment SA-2. Starting with the baseline task set, the amount of work to be performed by the synchronization task is increased by 1 KWI until a deadline is missed. The parameters of the periodic and aperiodic tasks do not change. This experiment resembles experiment SH-1 of the SH series. The difference is the added aperiodic processing in foreground and background. The experiment tests the system's scheduling ability, when there is an increase in the blocking time for all server tasks and foreground and background aperiodic processing are present.

Experiment SA-3. Starting with the baseline task set, the workload of all periodic tasks outside the critical section is increased uniformly by 1 KWI per period until some deadline is missed. The frequencies of the periodic tasks and the parameters of the synchronization task and the aperiodic tasks do not change. This experiment resembles experiments PH-3, SH-3, and AH-3. The experiment combines periodic scheduling with task synchronization, and aperiodic processing in foreground and background. The experiment tests the system's ability to handle an increase in the total periodic workload without significantly increasing the scheduling load when task synchronization and foreground and background aperiodic processing are present.

Experiment SA-4. Starting with the baseline task set, the periodic load is increased by adding new tasks, each with the frequency and workload of the middle periodic task P3, until a deadline is missed. This experiment resembles experiment PH-4, SH-4, and AH-4. It combines periodic scheduling with task synchronization, and with foreground and background aperiodic processing. The experiment tests the system's ability to handle an increase in the periodic scheduling load when task synchronization, and foreground and background aperiodic processing are present.

10.3. Application Domain

This series contains all of the basic complexities of many real-time systems. It is rich enough to modify in a number of ways to prototype many real systems. All the basic parameters can be modified by a user who wants to learn about system characteristics for a particular application. If the user knows of a workload more characteristic of the application, that workload can be substituted for the Whetstone workload. More periodic tasks can be added, frequencies and workloads of the periodic tasks can be changed, more aperiodics can be added, their interarrival times and workloads can be modified, and patterns of synchronization can be changed by adding additional server tasks or by changing the calling patterns of the client tasks.

11. Experience to Date

A number of lessons have been learned from the early implementation of the PH series of the benchmark in Ada. Perhaps the most important lesson was that the baseline task set has to be tailored to the individual system configuration. The benchmark will not give

the desired information if the baseline is too demanding or too easy. If the baseline in the PH series produces too small a starting utilization, the limiting factor will be the precision of the timing services. In experiment PH-1, this will enable a system with high throughput to fail at very low utilization rates while a system with low throughput will fail at a high utilization. For example, if the timing precision is 10 milliseconds, the highest frequency task cannot be driven above 100 Hertz without missing deadlines, independent of its utilization. This means that if two systems start with the same baseline task set (same workload), the one with the greater throughput will start (and finish) with a lower utilization rate. Here it is obvious that with the same baseline task set, breakdown utilization does not give an appropriate figure of merit (Donohoe, Shapiro and Weiderman 1990b).

Two conclusions were apparent from this observation. First, it is important to look at some combination of throughput measures and utilization measures. Second, it is necessary in some cases to scale the baseline workload up or down so that the same limiting factors are being tested. Guidelines for adjusting the baseline are given in (Donohoe, Shapiro and Weiderman 1990a). The basic rule of thumb is that the baseline workload be set to 10 to 30 percent of the workload achieved by a single task executing the workload without breaks or interruption. This is known as the raw Hartstone utilization and is expressed in KWIPS.

Another factor that must be regulated is the step size of the experiments. Since the step size determines the resolution with which the utilization can be measured, it must be small enough to produce reasonable accuracy, but large enough so that the experiments do not go on forever. A step size of 2 or 3 percent of CPU utilization seems to be desirable. This step size can be expressed in KWIPS as a percent of the raw Hartstone utilization in KWIPS.

In addition to the normal results given in terms of utilization rates, total number of deadlines processed per second, and response times, it is important to observe the way in which the system fails to meet its deadlines. If high priority tasks are missing their deadlines before low priority tasks in the PH series, it is an indicator that there is some kind of breakdown in the priority system. If deadlines are missed at one step in the experiment and then no deadlines are missed in subsequent steps of the experiment, it is an indicator of a possible transient event that may have preempted normal activities. If one deadline is missed by an extraordinary amount it can indicate either a deadlock situation or a transient event that affects only one task.

In conclusion, there is much that can be learned about a real-time system by having a set of benchmark requirements and associated experiments such as those described in this paper. Therefore the authors would like to encourage implementation in a number of programming languages for a number of systems so that some standard of real-time system behavior can be formulated. The Hartstone benchmark will be a useful tool for improving real-time system development.

A. Appendix: Generating Exponential Random Deviates

Exponential random deviates can be generated from a sequence of uniform random deviates. To generate the uniform random deviates, a linear congruential generator will be used of the form:

$$X_{n+1} := (aX_n + c) mod\ m$$

where:

X_0, the starting value $X_0 \geq 0$
a, the multiplier $a \geq 0$
c, the increment $c \geq 0$
m, the modulus $m > X_0$, $m > a$, $m > c$

The congruential generator generates a sequence of integers in the range from 0 to $m - 1$. These integers can then be divided by m to give m separate real numbers distributed uniformly between 0 and 1.0.

For the purpose of this study we have chosen the following parameters which follow the criteria suggested by Knuth:

$$X_0 = 0$$

$$a = 2621$$

$$c = 6927$$

$$m = 32768$$

Knuth observes that if the modulus is one greater than, or one less than, a power of 2(2^{15} in this case) there will be greater randomness in the lower order bits of X_n. But this does not influence the randomness of the high order bits. His conclusion is that "in most applications, the low order bits are insignificant and the choice of [modulus equal to a power of two] is quite satisfactory—provided that the programmer using the random numbers does so wisely." In our case the randomness of the low order bits is much less important than the randomness of the high order bits and the speed of computation of each random deviate is of greater concern.

The first 5 numbers in the sequence, using the parameters above, are: 6927, 9122, 27817, 6484, 27667. The first 50 numbers in the series are shown in Table A-1.

Dividing these by the modulus, we arrive at the first 5 uniformly distributed random deviates of: 0.21139526, 0.27838135, 0.84890747, 0.19787598, 0.84432983. The first 50 uniformly distributed random deviates are also given in Table A-1.

The exponential random deviates can be generated from the uniform random deviates using Algorithm E (attributed to George Marsaglia) in Section 3.4.1 of Knuth (1969), Edition 1. (In Edition 2 the version of the algorithm we used was replaced by a different version, but we continue to use the first one because of its simplicity). Algorithm E is reasonably fast and requires few uniform deviates to be calculated for each exponential deviate. It is repeated here in somewhat different form using pseudocode:

Prepare a table of constants as follows:

$$P[j] = 1 - 1/e^j$$

$$Q[j] = (1/(e - 1)) * (1/1! + 1/2! + 1/3! + \ldots + 1/j!)$$

Table A-1. Uniform and exponential random deviates.

i	X_i	Uniform Deviates	Exponential Deviates
1	6927	0.21139526	1.27838135
2	9122	0.27838135	0.84432983
3	27817	0.84890747	0.75708008
4	6484	0.19787598	0.01712036
5	27667	0.84432983	0.56118774
6	6550	0.19989014	0.10113525
7	4045	0.12344360	0.96795654
8	24808	0.75708008	3.74923706
9	16983	0.51828003	0.34701538
10	20426	0.62335205	0.17330933
11	561	0.01712036	0.91369629
12	2748	0.08386230	0.23379517
13	475	0.01449585	0.08093262
14	6718	0.20501709	0.07778931
15	18389	0.56118774	0.34039307
16	2768	0.08447266	1.02612305
17	20127	0.61422729	0.90921021
18	3314	0.10113525	1.02539063
19	9401	0.28689575	1.75451660
20	5412	0.16516113	0.96914673
21	3235	0.09872437	0.16436768
22	31718	0.96795654	0.28207397
23	7389	0.22549438	1.23486328
24	7608	0.23217773	1.10134888
25	24551	0.74923706	2.69619751
26	31514	0.96173096	0.62774658
27	29761	0.90823364	1.18597412
28	22668	0.69177246	0.18365479
29	11371	0.34701538	0.82641602
30	24206	0.73870850	1.45007324
31	12005	0.36636353	0.31246948
32	14752	0.45019531	0.37512207
33	5679	0.17330933	4.86904907
34	14914	0.45513916	1.14825439
35	4297	0.13113403	0.12338257
36	29940	0.91369629	1.80905151
37	307	0.00936890	0.25448608
38	25142	0.76727295	1.43286133
39	7661	0.23379517	1.52102661
40	32392	0.98852539	0.34954834
41	4471	0.13644409	0.41580200
42	27242	0.83135986	0.23748779
43	6737	0.20559692	1.79376221
44	2652	0.08093262	0.13323975
45	11003	0.33578491	0.27096558
46	9950	0.30364990	2.02697754
47	2549	0.07778931	0.92520142
48	3184	0.09716797	0.20950317
49	29119	0.88864136	0.39419556
50	11154	0.34039307	0.54742432

Table A-2. P and Q array values.

i	$P[i]$	$Q[i]$
1	0.63212056	0.58197671
2	0.86466472	0.87297506
3	0.05021293	0.96996118
4	0.98168436	0.99421021
5	0.99326205	0.99906001
6	0.99752125	0.99986831
7	0.99908812	0.99998379
8	0.99966454	0.99999822
9	0.99987659	0.99999982
10	0.99995460	0.99999998
11	0.99998330	1.00000000
12	0.99999386	1.00000000
13	0.99999774	1.00000000
14	0.99999917	1.00000000
15	0.99999969	1.00000000

The size of the table is determined by the precision of the machine. Assuming six digits of accuracy, the tables should include values that are within 10^{-6} of 1.0. In our case we will assume 32 bit floating point arithmetic with a 24-bit mantissa for 6.7 digits of accuracy. Thus the tables include 14 values for P and 8 values for Q. These values are shown in Table A-2 to 8 decimal digits. Note that the value 1.0 can be inserted in arrays in the 14th and 9th positions for P and Q respectively for algorithmic convenience.

The algorithm has two parts, the first part generates the fractional part and the second part generates the integer part. The function call to Random generates the next number in the uniform distribution between zero and one:

Fractional part:

```
U0 := Random()
U1 := Random()
X  := U1
j  := 1
while Q[j] <= U0 do
  j := j + 1
  U := Random()
  if U < X then X := U
  endwhile
```

Integer part:

```
U := Random()
j := 1
while P[j] <= U do
  j := j + 1
  X := X + 1
  endwhile
```

Knuth shows the validity of this method for generating exponential random deviates with mean 1, and shows also that, on the average, there are only 3.582 uniform deviates calculated per exponential deviate.

The first 16 uniform deviates are required to generate the first five exponential random deviates. The first five exponential random deviates are: 1.27838135, 0.84432983, 0.75708008, 0.01712036, and 0.56118774. The first 50 exponential random deviates generated by this method are shown in Table A-1.

Generating exponential random deviates with mean **m** from the sequence of exponential random deviates with mean 1 is simply a case of multiplying by **m**.

References

1. Altman, N., and Weiderman, N.H., 1988. Timing variation in dual loop benchmarks, *Ada Letters*, 8(3): 98–102.
2. Baker, T.P., 1990. Fixing some time-related problems in Ada, *Ada Letters—Special Edition on the Third International Workshop on Real-Time Ada Issues 1989*, 4: 136–143.
3. Baker, T.P., Stack-based scheduling of realtime processes, *Journal of Real-Time Systems*, 3: 67–99.
4. Baker, T.P., and Shaw, A., 1988. The cyclic executive model and Ada, *Proceedings of the IEEE Real-Time Systems Symposium*, Huntsville, AL, December, pp. 120–129.
5. Borger, M., Klein, M., and Veltre, R., 1989. Real-time software engineering in Ada: Observations and Guidelines, CMU/SEI-89-TR-22, DTIC: ADA219020, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, September.
6. Chen, Min-Ih, and Lin, Kwei-Jay, 1990. Dynamic priority ceilings: A concurrency control protocol for real-time systems, *Real-Time Systems*, 2: 325–346.
7. Cheng, S.C., Stankovic, J.A., and Ramamritham, K., 1988. Scheduling algorithms for hard real-time systems: A brief survey, *Tutorial on Hard Real-Time Systems*, J.A. Stankovic and K. Ramamritham, eds.,, Computer Society Press of the IEEE, Washington, D.C.
8. Clapp, R.M., Duchesneau, L., Volz, R.A., Mudge, T.N., and Schultze, T. 1986. Toward real-time performance benchmarks for Ada, *Communications of the ACM*, 29(8): 760–778.
9. Curnow, H.J., and Wichmann, B.A., 1976. A synthetic benchmark, *Computer Journal*, 19(1): 43–49.
10. Donohoe, P., Shapiro, R., and Weiderman, N., 1990a. *Hartstone Benchmark User's Guide*, CMU/SEI-90-UG-1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
11. Donohoe, P., Shapiro, R., and Weiderman, N., 1990b. *Hartstone Benchmark Results and Analysis*, Version 1.0, CMU/SEI-90-TR-7, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
12. Gafni, V., 1989. A tasking model for reactive systems, *Proceedings of the Real-Time Systems Symposium*, Santa Monica, CA, December, pp. 258–265.
13. Ghazalie, T., 1990. Improving aperiodic response with deadline scheduling, Master's Thesis, Florida State University.
14. Goodenough, J.B., and Sha, L., 1988. The priority ceiling protocol: A method of minimizing the blocking of high-priority Ada tasks. *Proceedings of the Second International Workshop of Real-Time Ada Issues*, Mortonhamstead, Devon, UK, June.
15. Harbaugh, S., and Forakis, J., 1984. Timing studies using a synthetic Whetstone benchmark, *Ada Letters*, 4(2): 23–34.
16. Kamenoff, N.I., and Weiderman, N.H., 1991. Hartstone distributed benchmark: Requirements and definitions, *Proceedings Twelfth Real-Time Systems Symposium*, San Antonio, TX, December 4–6, 1991, IEEE Computer Society Press (1991), 199–208, Los Alamitos, CA.
17. Kar, R.P., and Porter, K., 1989. Rhealstone—A real-time benchmarking proposal, *Dr. Dobbs Journal*, 14(2): 14–24.
18. Kar, R.P., 1990. Implementing the rhealstone the real-time benchmark, *Dr. Dobbs Journal*, 15(4): 46–104.
19. Knuth, D.E., 1969. The art of computer programming: Seminumerical algorithms, Reading, MA, Addison-Wesley, Vol. 2.

20. Levi, S.T., Tripathi, S.K., Carson, S.D., and Agrawala, A.K., 1989. The MARUTI hard real-time operating system, *Operating Systems Review*, 23(3): 90–105.
21. Liu, C.L., and Layland, J.W., 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment, *Journal of the Association of Computing Machinery*, 20(1): 46–61.
22. Locke, C.D., 1986. Best-effort decision making for real-time scheduling, Ph.D. Thesis, Carnegie Mellon University, May.
23. Mok, A., 1983. *Fundamental design problems of distributed systems for the hard-real-time environment*, Cambridge, Massachusetts Institute of Technology.
24. Pollack, R.H., and Campbell, D.J., 1990. Clock resolution and the PIWG benchmark suite, *Ada Letters—Special Edition on Ada Performance Issues*, X, 3: 91–97.
25. Roy, D., and Gupta, L., 1990. PIWG analysis methodology, *Ada Letters—Special Edition on Ada Performance Issues*, X, 3: 217–229.
26. Sha, L., Rajikumar, R., and Lehoczky, J.P., 1987. *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*, CMU-CS-88-181, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA.
27. Sha, L., Goodenough, J.B., 1990. Real-time scheduling theory and Ada, CMU/SEI-89-TR-14, DTIC: ADA211397. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
28. Sha, L., and Goodenough, J.B., 1990. Real-time scheduling theory and Ada. *IEEE Computer*, 23(4): 53–62.
29. Sprunt, B., Sha, L., and Lehoczky, J., 1989. *Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System*, CMUSEI-89-TR-11. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
30. Sprunt, B., Sha, L., and Lehoczky, J., 1989. Aperiodic task scheduling for hard-real-time systems, *Journal of Real-Time Systems*, 1(1): 27–60.
31. Sprunt, B., and Sha, L., 1990. *Implementing Sporadic Servers in Ada*, CMU/SEI-90-TR-6. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
32. Stankovic, J.A., and Ramamritham, K., 1988. Real-time computing systems: The next generation (J.A. Stankovic, ed., *Tutorial: Hard Real-Time Systems*, IEEE Computer Society Press, pp. 14–37.
33. Stankovic, J.A., and Ramamritham, K., 1989. The spring kernel: A new paradigm for real-time operating systems, *Operating Systems Review*, 23(3): 54–71.
34. United States Department of Defense, 1983. *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A-1983, American National Standards Institute, New York.
35. Weicker, R.P., 1984. Dhrystone: A synthetic systems programming benchmark, *Communications of the ACM*, 27(10): 1013–1030.
36. Weicker, R.P., 1988. Dhrystone benchmark: Rationale for version 2 and measurement rules, *SIGPLAN Notices*, 23(8): 49–62.
37. Weiderman, N., 1989. Hartstone: Synthetic benchmark requirements for hard real-time applications, CMU/SEI-89-TR-23, DTIC: ADA219326, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
38. Weiderman, N., 1990. Hartstone: Synthetic benchmark requirements for hard real-time applications, *Ada Letters—Special Edition on Ada Performance Issues*, X, 3: 126–136 (also available as CMU/SEI-89-TR-23, DTIC: ADA219326).
39. ISO-IEC/JTC1/SC22/WG9 (Ada) Numerics Rapporteur Group, 1989. Proposed Standard for a Generic Package of Elementary Functions for Ada, ISO/IEC/JTC1/SC22/WG9 N57,, WG9 Numerics Rapporteur Group, October.
40. Wichmann, B.A., 1988. Validation code for the Whetstone benchmark, DITC 107/88, National Physical Laboratory, Teddington, Middlesex, UK, March.