A Project To Investigate Mechanisms and Methodologies for the Design and Construction of Communicating Concurrent Processes in Real-Time Environments

Yearly Report, 1979

R. H. Campbell

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, Illinois 61801

217-333-0215

December 21, 1979

ABSTRACT

Software for embedded aerospace computer systems is becoming more sophisticated and increasingly involved with the programming of asynchronous processes, synchronization, coordination and communication between processes, and guarantee of performance within a set of real-time constraints. Such real-time software is difficult to develop, error prone, and expensive. This report summarizes the results of research undertaken in 1979 into effective and appropriate mechanisms to aid in the design and construction of such software for use in the flight research programs undertaken by NASA.

TABLE OF CONTENTS

1 Introduction.

This report describes the results of research undertaken at the University of Illinois by a project funded by a grant from NASA [Campbell, 78b]. The problems of real-time programming including high-level language mechanisms to support device programming, concurrent processing, encapsulation, fault-tolerance and reliability were investigated.

The advent of low cost, lightweight computers with sophisticated computational abilities enables analog control systems, monitoring systems, and pilot functions in aerospace vehicles to be replaced or enhanced by the inclusion of embedded computer systems. This technology may permit the application of radically new techniques to the design of aircraft, spacecraft and ground control. Software for such systems involves the programming of asynchronous processes, coordination and communication between processes, design of network protocols, programming of hardware devices, and guarantee of performance within a set of real-time constraints. Such software is difficult to develop, often error prone and a major expense.

Several new mechanisms have been devised to aid programming of real-time systems. High-level declarations of synchronization, concurrency, and encapsulation have been proposed and included in an experimental language called Path Pascal. The language has been used to program a wide variety of applications as supportive evidence of the effectiveness of these mechanisms. In addition, a deadline mechanism has been proposed that provides a specification of real-time constraints within the real-time program. The mechanism provides a means to program algorithms whose completion must occur before the deadlines specified and provides tolerance to classes of timing-faults in the

design of the algorithms or the hardware. Finally, research results indicate that the programming of hardware devices is possible from a high-level language permitting a more readable, modifiable, and concise description of such algorithms. High-level mechanisms to allow the programming of device handling routines were included in Path Pascal and have been used to implement several practical stand-alone systems.

1.1 Real-Time Systems Programming.

Real-time systems require programming of synchronization, coordination and communication between concurrent processes. This activity should be performed in a clear, structured manner which aids in the verification of the programs produced. An experimental real-time systems programming language called Path Pascal has been constructed [Kolstad & Campbell, 79], [Campbell, et al., 79]. This language is based upon Pascal [Jensen & Wirth, 75] and extended to include processes, a synchronization mechanism based on Open Path Expressions, input and output device programming features, and an encapsulation mechanism. These additions to Pascal are sufficiently general that they can be adopted as extensions to programming languages currently used by NASA such as HAL/S. The language is being used as a testbed to experiment with Path Expressions and real-time programming problems.

1.1.1 Path Pascal System.

The Path Pascal programming system provides a small set of simple, experimental software development tools for the design and production of real-time systems. Currently included is a compiler and several interchangeable run time environments for the Path Pascal programming language. The run time environments allow the language to be used to write real-time systems

that run efficiently on machines with or without native operating systems. Path Pascal programs may be executed interpretively (for emulating the performance of a machine programmed in Path Pascal) or used to simulate the performance of algorithms using the simulation package. The simulation and emulation environments have been used successfully as design tools for several simple stand-alone PDP-11 software systems including simple network communication software.

Path Pascal is a superset of the Pascal P language [Ammann, et al., 76]. It contains constructs for concurrency along with encapsulated data objects and synchronization provided by Open Path Expressions [Campbell, 76]. The Path Pascal compiler translates programs into an intermediate language called P-code. The P-code can be interpreted or transformed into a particular machine language. The interpreter, compiler, and machine code assemblers are all written in Pascal [Kolstad & Campbell, 79], [Balocca & Campbell, 79]. The interpreter for the intermediate code can simulate concurrency and can collect statistical information on the behavior of Path Pascal programs. Translators and run time environments exist for the Zilog Z-80, the PDP-11 and the PRIME 500 machines. The PDP-11 environments include a single process (pseudo-concurrent) environment under the Unix operating system, a stand-alone multiprocessor environment, and a stand-alone multiprogramming environment.

The compiler, interpreter, and the stand-alone system for the LSI/11 appear to be reliable and have been distributed to other organizations as experimental systems. NASA distributes the compiler and interpreter to interested organizations on request.

Several short presentations of Path Pascal have been given [Campbell, 78a], [Campbell, 79a], [Campbell, 79b] in addition to papers presented at

conferences: [Campbell & Kolstad, 79a], [Balocca & Campbell, 79], and [Campbell & Kolstad, 79b].

1.1.2 Experiments.

In order to evaluate the practical aspects of programming real-time systems using proposed language constructs, several systems with Path Pascal have been implemented and Path Pascal has been used in operating system courses at the University of Illinois. Among the various examples programmed are:

1) A real-time executive based on a description of ASPS, a space telescope system designed by NASA which schedules periodic tasks. The resulting program is smaller and more concise than the original executive written in assembler code.

2) Teletype device driver software and a simple message passing system. The software allowed full 4800 baud communication between two terminals controlled by an LSI/11.

3) Dijkstra's concurrent garbage collection algorithm.

4) A simulation of a Batch Spooling System.

5) Network software based on a design by Brinch Hansen [Campbell & Kolstad, 79a]. This network software has been reconfigured to execute on several stand-alone PDP-11s linked together by serial lines. The changes in the software to allow it to run on the PDP-11s were minor and localized to the hardware/software interfaces.

6) Simulation language much in the same manner as Simula. A simulation package to support simulation in Path Pascal has been written and will soon be available. Various simulations have been performed and a report describing this application of Path Pascal is being prepared. Unlike Simula, the algorithms simulated in Path Pascal may be transported directly to a stand-alone machine. Thus, the Path Pascal system permits direct development of algorithms in one language, from design through simulation to actual implementation. This provides a useful tool in the construction of systems [Randell, 68].

7) A simple file system for electronic mail on a LSI/11. Concurrent access to the mail system by users is possible. The file system uses a floppy disk and is written entirely in Path Pascal (including device drivers).

Path Pascal has been used as the major programming tool in three operating system courses at the University of Illinois. It proved to be a very useful teaching tool as it permitted students to explore the programming of concurrent systems without requiring them to familiarize themselves with a particular set of hardware. The Path Pascal compiler is being used experimentally by Dr. Foudriat at NASA Langley.

1.1.3 I/O Device Programming.

Path Pascal includes interrupt processes which may be used to program I/O device drivers in a high level language. An Experimental translator from the intermediate code of Path Pascal to PDP-11/20 machine code was developed and has been used to construct 'stand alone' software. Device drivers for teletypes and disks have also been written.

1.1.4 Real-Time Implementation.

A translator from P-code to PDP-11 machine code has been developed which uses an average of only 3.1 bytes of PDP-11 storage for each P-code instruction. This translator allows the implementation of fast, efficient PDP-11 real-time programs in Path Pascal [Balocca & Campbell, 79].

1.1.5 Portability.

Path Pascal is easily portable to various computer systems since the compiler is written in Pascal P4. Although the intermediate P-code produced by the Path Pascal compiler includes device processing, synchronization and concurrent processing instructions, this code retains the machine independence of the Path Pascal Language. Translators for this P-code to a particular

machine are easy to write as demonstrated by the current implementations of Path Pascal on several machines:

1) An 8 bit word microprocessor: Z-80 [Morgan, 79]

2) Small, 16 bit word machines: PDP-11/20, PDP-11/40, and LSI-11 [Balocca & Campbell, 79].

3) A large minicomputer with 32 bit words: the Prime 500

The run time system for Path Pascal programs is quite small and easy to program for a target machine. One LSI-11 implementation has 1,645 source lines which translate to 2,792 bytes of code (including several hundred bytes of run time messages and diagnostics).

1.1.6 Encapsulation.

One aspect of the encapsulation mechanism of Path Pascal allows the programming of abstract data types which can be manipulated only through special procedures. The purpose of this restriction is to encourage the programmer to construct modular programs in which the description and implementation of a data type is declared only within a small module of the program. Indeed, knowledge of the internal representation of an abstraction may be harmful [Parnas, 72]. Path Pascal can implement data abstraction in several ways:

> The standard Pascal data type mechanism permits the programming of user defined types. However, the abstraction of such data types is not enforced and the details of any data structure may always be accessed in any program statement.
>
> Path Pascal includes a construct called an 'object' which can be used to program abstract data types. This abstraction is compiler enforced -- variables of type object cannot be assigned, compared, or manipulated other than by invocation of the object's operations.
>
> Objects may include constant and type definitions to be used in the formation of the internal data structure of the object or to provide abstract data types which are 'managed' by the object. An abstract data type to be managed by an object though instantiated

outside the object is declared as an entry type within that object. Entry types are exported to the scope containing the object and may be used in type and variable declarations. The internal data structure of the entry type is completely inaccessible outside the object. Comparison and assignment of variables of the same entry type is permitted. An example of an entry type and its object manager is shown below:

```
var clx: object          (* complex numbers *)
    path  1:(add, cons)  end;

    type pair =     record
            ip, ir : real
            end;
         complex =  entry ^pair;

    entry cons(var z: complex; x,y : real);      (* instantiate *)
            begin
                new(z); z^.ip := x; z^.ir := y
            end;

    entry add(c1, c2: complex; var c3: complex);     (* add *)
            begin
                new(c3);
                c3^.ip := c2^.ip + c1^.ip;
                c3^.ir := c2^.ir + c1^.ir
            end;
end (*clx*);

var root1, root2, root3: complex;

(* create a pair of complex numbers and add them *)

clx.cons(root1, 0.57, 3.3);
clx.cons(root2, 23.3, 4.3);
clx.add(root1,root2,root3);
```

An object for storage of complex numbers is created. The 'cons' routine allows construction of a complex number containing the usual real and imaginary parts. The 'add' routine performs the usual operation which has been expressed here in three address code. Within the object, immediate access is obtained to the detail of the variables declared as entry types. The synchronization of the object is used to maintain the integrity of any complex numbers declared in a concurrent processing environment. Entry types permit a data abstraction mechanism similar to that of Modula [Wirth, 77]. Operations within an object may be defined with efficient access to the internal representation of several variables declared to be entry types of that object. Note that the object synchronizes the execution of its operations and that it thus synchronizes access to all instances of its entry types.

1.1.7 Path Expressions.

Several variations of the Path Expression notation have been examined. In one variation, names are allowed to be repeated in a single Open Path Expression. An implementation of a stack is shown below to illustrate the utility of this concept:

```
const stacksize = 235;

type stack = object

        path stacksize:(push; pop), 1:(push, pop) end;

        var stk: array [1..stacksize] of element;
            pointer: 1..stacksize;

        entry procedure push(var item: element);
            begin
                stk[pointer] := item;
                pointer := pointer + 1
            end (*push*);

        entry function pop: element;
            begin
                pointer := pointer - 1;
                pop := stk[pointer]
            end (*pop*);

        init; begin pointer := 1 end;

        end (*stack*);
```

The Path Expression defines all the legal sequences of operations that can be performed on the stack data structure without contravening its stack-like behavior. The Path Expression specifies that there are 'stacksize' resources to be shared between executions of the procedures 'push' and 'pop'. The 'push' procedure acquires a resource and 'pop' releases one. The second part of the Path Expression (1:(push, pop)) declares that a single resource (the stack pointer) is to be shared between 'push' and 'pop'. The synchronization constraints expressed on each operation apply in the order declared: from left to

right. Thus, several executions of 'push' may occur before a 'pop' need occur. The scheme allows many synchronization schemes that would require an implementation involving several nested objects to be expressed in one object. It does, however, permit deadlocks to be expressed by a single Path Expression. The current Path Pascal compiler does not include this extension.

1.2 Specification of Timing.

Fault-tolerance is difficult to achieve in real-time systems without the ability to respond to timing errors in real-time. One way proposed [Horton, et al., 78], [Campbell, et al., 79] [Horton, 79] to meet this time-dependent requirement for fault-tolerance is the deadline mechanism. The deadline mechanism provides a means to specify and program the timing constraints that apply to algorithms within the system. The mechanism also provides a means to define completely redundant algorithms that are to be performed in the event of a timing fault in a primary algorithm of the system. The redundant algorithms are typically short and complete their execution within a known length of time. The deadline mechanism ensures that either the primary algorithm or the alternate algorithm is performed before the deadline. The mechanism permits the construction of completely redundant systems which maintain their full system service despite timing faults or partially redundant systems which gracefully degrade as timing faults occur. In either case, the mechanism guarantees that each deadline of the system is met. Fault-tolerant real-time systems are required in many applications where immediate human intervention is impossible. Examples of such applications occur in aerospace, control systems, process control, computer networks and telecommunications. For example, network routing algorithms may provide a primary service which determines an optimized route for communication given the current

network load, or as an alternative may find any available route. Currently, an experimental version of the deadline mechanism is being implemented in Path Pascal using the recovery cache built into the Path Pascal interpreter [Wei, et al., 78]. A simulation of the multi-mission modular satellite software is being written using this deadline mechanism to provide fault-tolerance. The simulation, we believe, will be useful in evaluating the effectiveness of the mechanism in practice [see appendix].

1.3 Theoretical Results.

The deadline mechanism can be applied in a modified form to time-shared single processor systems (such as ASPS) which execute a set of jobs periodically and within a fixed set of deadlines. An algorithm has been defined which selects and schedules alternates and primaries to optimize the number of primaries that are executed [Liestman, 79].

For many periodic real-time systems like the ASPS system it is possible to devise schedules that optimize the degree of fault-tolerance of those systems. That is, if timing faults occur and future deadlines can no longer be met with primary algorithms, it is possible to determine a schedule which will permit the maximum number of primaries to be attempted in the available time. This schedule may be determined at compile time or at run-time by an algorithm with order n computations (where n is the number of real-time tasks.) A report on the theoretical aspects of the deadline mechanism applied to periodic systems follows [see appendix].

2 Summary.

The project has proposed several mechanisms to aid in the construction and the design of software for reliable, fault-tolerant real-time systems containing asynchronous processes. Some of these mechanisms have been incorporated in the Path Pascal compiler which can be used to program practical real-time systems. Theoretical, practical, and simulation studies have been performed on fault-tolerant mechanisms for real-time programming and the results of these studies are very encouraging.

The Path Pascal language is based on Pascal and includes Path Expressions, concurrent processes, device programming and real-time features. Path Pascal has been implemented efficiently and easily on several computers and may be used to program software that executes on a bare machine or execute under supervision from an operating system. Path Pascal programs can be run interpretively by emulating a particular computer or run in a simulation environment. The use of Path Pascal for simulation permits the design and measurement of particular algorithms (including algorithms involving the use of multiprocessors or networks). These designs may be refined by the addition of configuration and machine implementation dependent modules into a practical system on a given set of hardware. Path Pascal has been used to program example real-time control systems and network communication systems.

The deadline mechanism provides a means to implement program real-time systems to meet deadlines and allows such programming to include recovery from timing faults that may occur because of design or hardware faults. The feasibility of the deadline mechanism has been examined by simulations and theoretical studies. The theoretical study indicates that for the periodic real-time

systems of the type frequently used by NASA for control, simple algorithms exist which may be used to generate fault-tolerant schedules optimizing possible recovery. The simulations demonstrate the applicability of the mechanism in demand and interrupt-driven real-time systems. A practical implementation scheme is being devised and various applications of this mechanism to actual NASA real-time control systems are being made.

We believe that the results from this research project will be of benefit to the production of software for aerospace computer systems in general, and in particular, to the flight research undertaken at NASA.

3 References.

[Ammann, et al., 76] Ammann, U., K. Nori, and C. Jacobi, "The Portable Pascal Compiler," Institut Fuer Informatik, EIDG, Technische Hochschule CH-8096, Zurich, 1976.

[Balocca & Campbell, 79] Balocca, R. and R. H. Campbell, "PP-11, A Path Pascal Language System for the PDP-11," Proceedings of the Eighth Texas Conference on Operating Systems, Dallas, November, 1979.

[Campbell & Kolstad, 79a] Campbell, R. H. and R. B. Kolstad, "Path Expressions in Pascal," Fourth International Conference on Software Engineering, Munich, September 17-19, 1979.

[Campbell & Kolstad, 79b] Campbell, R. H. and R. B. Kolstad, "Practical Applications of Path Expressions to Systems Programming," ACM79, Detroit, 1979.

[Campbell & Miller, 78] Campbell, R. H. and T. J. Miller, "A Path Pascal Language," Technical Report UIUCDCS-R-78-919, Department of Computer Science, University of Illinois, Champaign-Urbana, 1978.

[Campbell, 76] Campbell, R. H., "Path Expressions: A technique for specifying process synchronization," Ph.D. Thesis, The University of Newcastle upon Tyne, August, 1976; Also, Department of Computer Science Technical Report, University of Illinois at Urbana-Champaign, UIUCDCS-R-77-863, May, 1977.

[Campbell, 78a] Campbell, R. H., "Path Expressions for Real-Time Programming," NASA/AIAA Workshop on Tools for Embedded Computing Systems Software, Hampton, Virginia, November, 1978.

[Campbell, 78b] Campbell, R. H., "Progress Report 1: A Project to Investigate Mechanisms and Methodologies for the Design and Construction of Communicating Concurrent Processes in Real-Time Environments," Progress Report, Department of Computer Science, University of Illinois, Champaign-Urbana, 1978.

[Campbell, 79a] Campbell, R. H., "Path Pascal," Software Development Tools Workshop, Pingree Park, May, 1979.

[Campbell, 79b] Campbell, R. H., "Path Pascal," NASA Workshop on Tools for Embedded Computing Systems Software, Hampton, Virginia, November, 1979.

[Campbell, et al., 79] Campbell, R. H., K. Horton, and G. G. Belford, "Simulations of a Fault-Tolerant Deadline Mechanism," The Ninth Annual International Symposium on Fault-Tolerant Computing, June, 1979.

[Horton, 79] Horton, K. H., "A Fault-Tolerant Deadline Mechanism," M. S. Thesis, University of Illinois, Urbana, October, 1979.

[Jensen & Wirth, 75] Jensen, K. and N. Wirth, Pascal User Manual and Report,

Springer-Verlag, New York, 1975.

[Kolstad & Campbell, 79] Kolstad, R. B, and R. H. Campbell, "Path Pascal User Manual," Department of Computer Science, University of Illinois, Systems Research Group, September, 1979.

[Liestman, 79] Liestman, A., "A Fault-Tolerant Scheduling Problem," In Preparation, December, 1979.

[McKendry, et al., 79] McKendry, Martin, Roy Campbell and Robert Kolstad, "Implementation of a Tree-Structured Operating System," to be published, 1979.

[Morgan, 79] Morgan, R., "Translating Path Pascal Pseudo Code into Zilog Z80 Microprocessor Assembly Code," M.C.S. Report, University of Illinois, Urbana, 1978.

[Parnas, 72] Parnas, D. L. "A technique for software module specification with examples," Comm. ACM 15, pp. 330-336, 1972.

[Randell, 68] Randell, Brian, "Towards a Methodology of Computing System Design," Software Engineering Conference, Garmisch, Germany, October 7-11, 1968.

[Randell, et al., 78] Randell, B., P. A. Lee, and P. C. Treleaven, "Reliability Issues in Computing System Design," Computing Surveys, Vol. 10, No. 2, pp. 123-165, 1978.

[Schindler & Steinacker, 79] Schindler, Sigram and Michael Steinacker, "A Formal Specification of an X.25 Protocol Machine," Fachbereich Informatik Technische Universitat Berlin, Available through authors, 1979.

[Wei, et al., 78] Wei, Anthony Y., Roy H. Campbell, and Geneva G. Belford, "Reliability Modeling of Recovery Blocks," Systems Research Group Report, University of Illinois/Urbana-Champaign, 1978.

[Wirth, 77] Wirth, N., "Modula: a Language for Modular Multiprogramming," Software-Practice and Experience, Vol. 7, pp. 3-84, 1977.

Progress Report on

A Fault-Tolerant Scheduling Problem

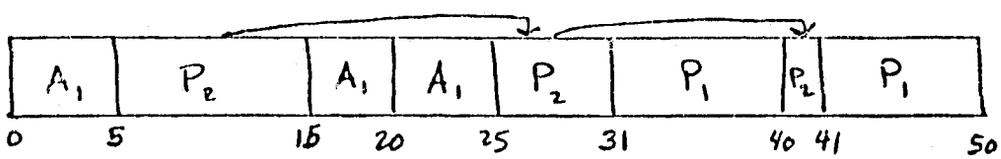
1 Introduction.

A real-time system is designed to provide a service which meets a set of specifications including real-time constraints. Should timing errors occur (either because of interrupt specifications or faults in design) and the system become heavily loaded with requests for service it may be impossible to satisfy all the requests by their respective deadlines. Based on the recovery block mechanism [Randell, 75] for fault-tolerant software, a deadline mechanism [Campbell, Horton & Belford, 79] has been proposed. In this mechanism two algorithms are provided for each service subject to timing constraints. The primary algorithm produces a better quality service than the alternate. The alternate is a simpler algorithm which requires less time to produce an acceptable result than the primary algorithm. This paper considers the problem of maximizing the number of primaries scheduled when error recovery is required.

We consider a scheduling problem in which a time-shared single-processor computing system is to execute a set of jobs each of which consists of a sequence of periodic requests. That is, each job periodically demands a response within a certain time interval. This response can consist of the completed execution of either a primary algorithm or an alternate algorithm. A further property of the proposed system is that each job's request period is a multiple of the next smallest request period. We will refer to such a system

as simply periodic. Let $J=\{J_1,J_2,\ldots,J_r\}$ denote a set of jobs with periodic requests. We shall use T_i to denote the request period, P_i to denote the computation time of the primary and A_i to denote the computation time of the alternate for job J_i $i=1,2,\ldots,r$. The jobs are ordered such that $m_iT_i = T_{i+1}$ for some positive integer m_i for $i=1,2,3\ldots,r-1$.

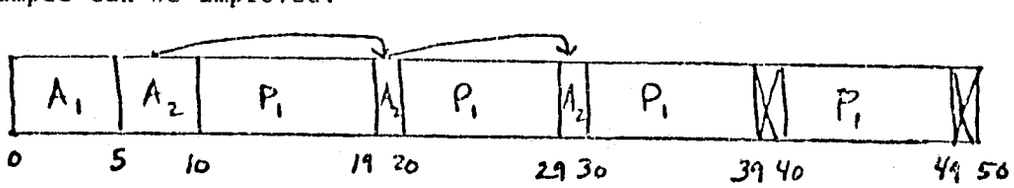
We define the deadline of a request to be the time at which the next request of the same job arrives. By scheduling a set of jobs with simply periodic requests we mean to specify which alternate or primary is to be executed at every time instant. A schedule is feasible if all requests will be satisfied before their deadlines. We assume that the execution of an alternate or primary can be interrupted if it is so desired. Consider the following example: Let J_1, J_2 denote jobs such that $A_1=5$, $P_1=9$, $T_1=10$ and $A_2=7$, $P_2=17$, $T_2=50$. A schedule for such a set of jobs can be described by a timing diagram:



The execution of P_2 is divided into three sections which are scheduled in the intervals 5-15, 25-31 and 40-41.

Due to the nature of the primary and alternate algorithms we would like to execute as many primaries as possible while still ensuring that all deadlines are met. We note that in the above example, two P_1's and one P_2 are executed during the period of J_2.

As the following schedule illustrates, the number of primaries executed in this example can be improved:



In this schedule four P_1's are executed and idle time is scheduled during the intervals 39-40 and 49-50. It is easy to see that this is the largest number of primaries which could be executed in one period of J_2.

2 A Scheduling Algorithm

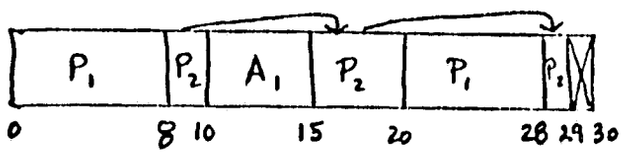
Given a set of jobs $J=\{J_1, J_2,\ldots,J_r\}$ we can create a schedule which will maximize the number of primaries executed. The algorithm below creates such a schedule for the period of J_r given A_i, P_i and T_i for i=1,2,...r. Each uninterrupted section of a job's execution is represented by a list element of the following type:

```
element = record
  start-time     : integer;
  scheduled-job  : jobtype;
  diff           : integer;
  next-job       : ↑element;
  next-diff      : ↑element;
  next-sec       : ↑element;
  next-idle      : ↑element;
  end;
```

The start-time field contains the time when this particular section begins execution. The scheduled-job field contains A_i, P_i for i=1,2,...r or IDLE to indicate what job is scheduled for this time period. The diff section contains the value P_i-A_i when the section is the first section of a scheduled P_i and a zero otherwise. The next-job field contains a pointer to the next list element. The next-section field points to the next section of the execution of this particular job. A list of the scheduled primaries is kept by the next-diff field. This list is sorted by diff values in nonincreasing order with equal values being ordered by decreasing start times. A list of scheduled

idle time is maintained using the next-idle field.

Given P_1=8, A_1=5, P_2=8, A_2=7, T_1=10 and T_1=30 we could represent the schedule:



as:

	start time	sched job	diff	next job	next diff	next sec	next idle
1	0	P_1	3	2	2	-	-
2	8	P_2	1	3	-	4	-
3	10	A_1	0	4	-	-	-
4	15	P_2	0	5	-	6	-
5	20	P_1	3	6	1	-	-
6	28	P_2	0	7	-	-	-
7	29	IDLE	0	-	-	-	-

The schedule for $J = \{J_1, J_2, \ldots, J_r\}$ is created by iteratively creating schedules for the sets $\{J_1\}$, $\{J_1, J_2\}, \ldots, \{J_1, J_2, \ldots, J_r\}$. In particular the schedule for $\{J_1, \ldots, J_i\}$ is constructed by concatenating m_{i-1} copies of the schedule for $\{J_1, \ldots, J_{i-1}\}$ and then modifying the resulting schedule. This modification is described in the following procedure:

```
(* procedure to create modified schedule by adding either an alternate
   or a primary to the existing schedule                                *)

MODIFY(ALT,PRIM)

while idle time < ALT
  begin
    change the first primary in the diff list to its alternate
    add the newly created idle time to the idle list
  end

if PRIM <= idle time
then begin
       idle time := idle time - PRIM
       schedule the primary at the current level to
         execute in the first PRIM units of idle time,
         inserting PRIM-ALT in the diff list.
     end
else begin
  if PRIM - ALT < largest diff value
  then begin
         idle time := idle time + largest diff value - PRIM
         change the first primary in diff list to its alternate
         schedule the primary at the current level to execute
           in the first PRIM units of idle time, inserting
           PRIM-ALT in the diff list.
       end
  else begin
         idle time := idle time - ALT
         schedule the alternate at the current level to
           execute in the first ALT units of idle time.
       end
  end
end (* of MODIFY *)
```

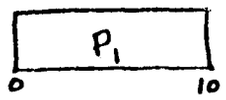
The schedule for J is then found by executing the following program:

```
for i := 1 to r do
begin
  concatenate m_{i-1} copies of the schedule for {J_1,...,J_{i-1}}

  MODIFY (A_i,P_i)
end
```

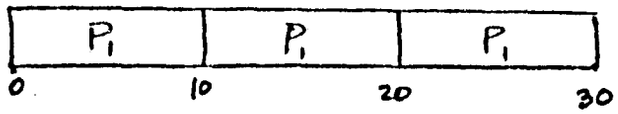
We note that the schedule for {} is one unit of idle time and $m_0=1$.

Let us consider an example. Let J_1, J_2, J_3 be jobs such that A_1=6, P_1=10, T_1=10, A_2=4, P_2=7, T_2=30, A_3=4, P_3=10 and T_3=60.

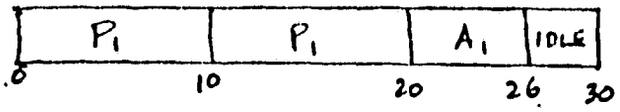
On the first call to MODIFY , P_1 is scheduled since P_1=10 <= idle time=10:



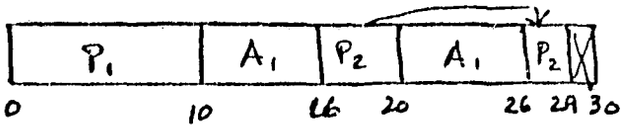
When i=2, 3 copies are concatenated to give:



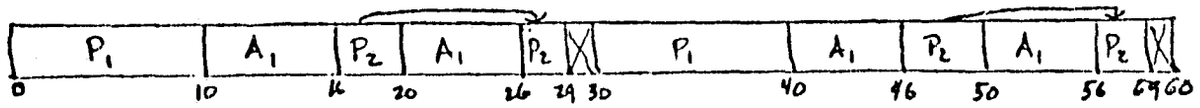
The first step of MODIFY changes the last P_1 to A_1:



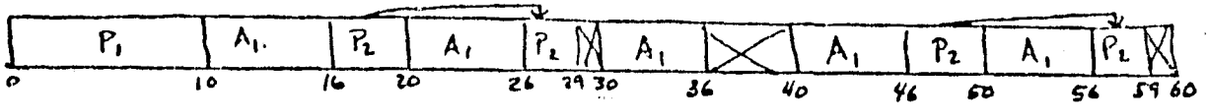
Since P_2=7 > idle time=4 and P_2-A_2=3 < largest diff value=4, one more P_1 is changed to A_1 and P_2 is scheduled:



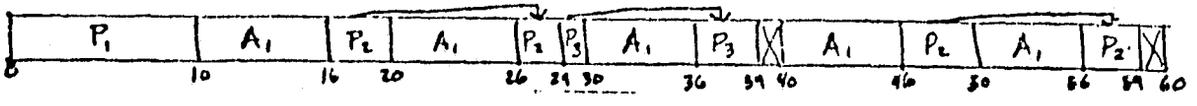
When i=3, 2 copies are concatenated to give:



The first step of MODIFY changes a P_1 to A_1:



Since P_3=10 > idle time=6 and P_3-A_3=6 > largest diff value=4, A_3 is scheduled:



We call a schedule optimal if it is feasible and has the maximum number of primaries scheduled among all feasible schedules.

Theorem: The schedule produced by the above algorithm is optimal and has as much idle time scheduled as any optimal schedule.

Proof: (by induction on r)

For r=1, the algorithm schedules P_1 if $P_1<T_1$ and schedules A_1 otherwise. This is clearly optimal and the idle time is maximized among optimal schedules since all optimal schedules have the same amount of idle time.

We assume that the algorithm produces an optimal schedule with maximum idle time for any set of p jobs.

Consider the set of jobs $J = \{J_1, J_2, \ldots, J_{p+1}\}$. The first p iterations of the algorithm produce an optimal schedule for $J' = \{J_1, J_2, \ldots, J_p\}$ with maximal idle time. Let us concatenate m_p copies of this schedule and call the resulting schedule S. Let t be the number of primaries in S. Clearly S is an optimal schedule for the jobs in J' over the period T_{p+1}. We must now add either A_{p+1} or P_{p+1} to the schedule.

We wish to maximize the number of primaries in the final schedule. The number of primaries contributed by the jobs in J' cannot exceed t. At least A_{p+1} units of idle time are needed to schedule a response for J_{p+1}. If the idle time in S is less than A_{p+1} then there is no feasible schedule for J with t primaries for the jobs in J'. Thus, some of the primaries must be changed to alternates so that either A_{p+1} or P_{p+1} can be scheduled. By changing those primaries with the largest diff values first, it is clear that the number of primaries changed is minimum and that among such changes, the idle time when A_{p+1} is scheduled is maximized. Thus if A_{p+1} is scheduled we have succeeded in finding an optimal solution for J.

There are two cases under which P_{p+1} might be scheduled instead of A_{p+1}. First, if P_{p+1} fits in the time allotted for A_{p+1} plus the remaining idle time then clearly this solution would be optimal since it includes one more primary than the solution with A_{p+1}. The second case under which P_{p+1} would be scheduled would be if a single P_j for $j<p+1$ could be converted to A_j so that P_{p+1} fits into the time allotted for A_{p+1} plus the idle time plus $P_j - A_j$ and the resulting idle time is greater than the idle time in the solution with A_{p+1}. In this case the idle time is increased and the number of primaries is not. Among such solutions, an optimal solution would be one such that P_j-A_j is maximum thus leaving the largest idle time in the solution for J.

Let $M_i = m_1 m_2 \cdots m_i$. Let $M_0 = 1$.

Theorem: The above algorithm creates a schedule for $O(M_{r-1})$ jobs in $O(rM_{r-1})$ time.

Proof: Let us first consider the number of jobs scheduled. Clearly there are $m_i m_{i+1} \cdots m_{r-1}$ requests for J_i for $i<r$ and 1 request for J_r. The total number of requests is

$\sum_{i=0}^{r-1} (M_{r-1}/M_i)$. Since $m_i>=2$ for all i then $M_i>=2^i$, thus

$$M_{r-1} <= \sum_{i=0}^{r-1} (M_{r-1}/M_i) <= 2M_{r-1}.$$

Thus $O(M_{r-1})$ jobs are scheduled.

Let us consider the time required to create the schedule. On the i^{th} iteration of MODIFY the first step changes k primaries to alternates. This requires O(k) steps where k is bounded by the number of primaries scheduled.

The second step of MODIFY will result in either an alternate or a primary being added to the schedule. This requires at most O(h) steps where h is the number of elements in the idle list. A call to MODIFY requires at most O(k+h) steps. Both k and h are bounded by the number of list elements.

The largest amount of time is consumed by the copy/concatenate step. The schedule created by the $i-1^{st}$ iteration must be copied m_{i-1} times to produce a schedule for the period T_i. The copies must then be concatenated which requires modification of the start times as well as forming the new diff list and idle list. This may be accomplished by visiting each element of the schedule for T_{i-1} m_{i-1} times. Thus, if s_{i-1} is the number of list elements after the $i-1^{st}$ iteration, the i^{th} step requires $O(s_{i-1}m_{i-1})$ steps and the entire algorithm requires $O(\sum_{i=1}^{r} s_{i-1}m_{i-1})$ steps.

The procedure MODIFY can easily be implemented so that at worst each T_i may contain one section from each level scheduled plus one idle section. Thus $s_{i-1} <= iM_{i-2}$ and the i^{th} step requires no more than $O(iM_{i-2}m_{i-1}) = O(iM_{i-1})$ steps. Summing over the r iterations we get:

$$\sum_{i=1}^{r} iM_{i-1} <= r\sum_{i=1}^{r} M_{i-1} <= rM_{r-1}\sum_{i=1}^{r} 2^{-i+1} <= 2rM_{r-1}.$$

Thus the algorithm requires $O(rM_{r-1})$ steps.

3 A Fault-tolerant Scheduling Algorithm.

The above algorithm maximizes the number of primaries executed in the system proposed. An interesting case arises when we make a slight change in the assumptions concerning the execution times of the primary algorithms. Let

us assume that the actual execution time of the primary is not known in advance. In this case the value P_i may be the expected execution time or the minimum execution time of the primary. The use of the above algorithm for this case can clearly lead to timing faults which result in failure to meet the real-time constraints.

In order to ensure a fault-tolerant schedule we must guarantee that every request i is fulfilled by either an A_i or a P_i.

We want to guarantee that if P_i fails then A_i can still be executed before the deadline. If we input P_i+A_i as the primary time for task i and A_i as the alternate time to the above algorithm, the result will be a schedule which maximizes the number of primaries scheduled with the additional constraint that whenever a primary is scheduled it's alternate must be scheduled to follow it. We must also make slight changes to MODIFY so that 'primary' means 'primary followed by alternate'. We may then use the following program to generate a fault-tolerant schedule for J:

```
for i := 1 to r do
begin
  concatenate m_{i-1} copies of the schedule for {J_1,J_2,...,J_{i-1}}
  MODIFY(A_i,A_i+P_i)
end
```

A schedule is <u>f-t feasible</u> if all requests will be satisfied before their deadlines even if no primary algorithms succeed. A schedule is <u>f-t optimal</u> if it is f-t feasible and has the maximum number of primaries scheduled among all feasible schedules.

<u>Theorem</u>: The schedule produced by the above algorithm is f-t optimal and has

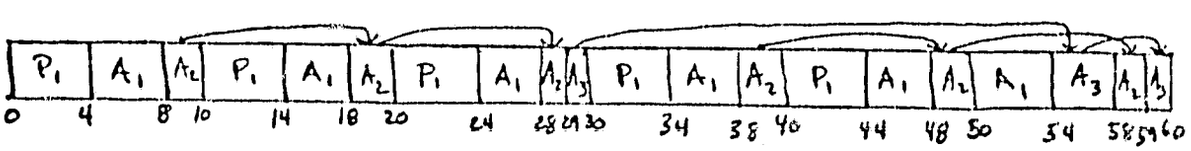
as much idle time as any f-t optimal schedule.

Proof: follows easily from the first Theorem.

Theorem: The above algorithm creates a f-t schedule for $O(M_{r-1})$ jobs in $O(rM_{r-1})$ time.

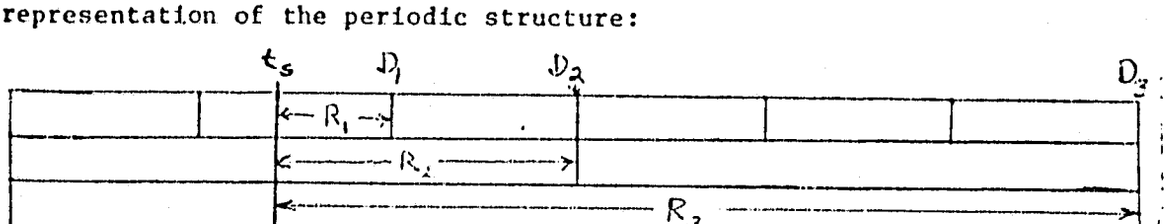
Proof: follows from the second Theorem.

Let us consider an example. Let J_1, J_2, J_3 be jobs such that A_1=4, P_1=4, T_1=10, A_2=5, P_2=7, T_2=30, A_3=6, P_3=8 and T_3=60. The f-t scheduling algorithm produces the following schedule:



As the scheduled jobs are executed let us assume that at time 4, P_1 fails to complete. A_1 is then executed and the deadline for J_1 at time 10 is met. The 2 units from 8 to 10 are used to begin execution of A_2. At time 10, P_1 interrupts and begins to execute. At time 14, P_1 succeeds. The request by J_1 has been satisfied and thus the time allocated to A_1 in the interval 14-18 can now be set to idle. We propose a new algorithm which can be used to reallocate this wasted time.

Let us assume that P_s succeeds at time t_s. We wish to create a new schedule for the interval t_s-T_r so that the number of primaries scheduled is maximized. We must bear in mind that some parts of alternates and primaries on other levels may have already been executed. Consider the following representation of the periodic structure:



We define EXA_i to be the number of time units of A_i already executed during the current J_i period when P_s succeeds at time t_s. Similarly, EXP_i is defined to be the number of time units of P_i already executed. We use $[x]$ to denote the largest integer not greater than x. D_i, the next J_i deadline after t_s, can be computed by: $D_i = [t_s/T_i] * T_i$. Let $R_i = D_i - t_s$ denote the remaining time before the next J_i deadline. When P_s succeeds, we compute D_i and R_i for each level $i \neq s$. Between R_i and D_i we must schedule a response to the request for J_i if it has not already been satisfied. We may schedule either a primary followed by an alternate or just an alternate. The times required for these responses are $P_i+A_i-EXA_i-EXP_i$ and A_i-EXA_i respectively. From D_i to D_r we must schedule responses as before.

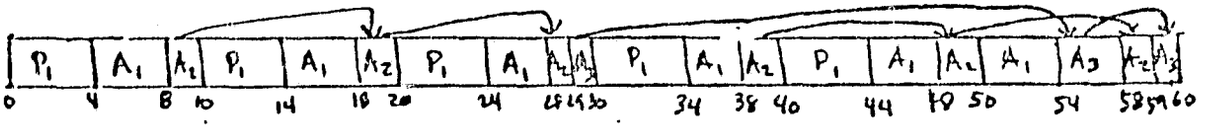
As before we create the schedule iteratively beginning at the lowest level. Except for levels s and r we create 2 schedules for level i. The first schedule is for the interval t_s-D_i and is built upon the schedule for t_s-D_{i-1} from the previous iteration concatenated with $(D_i-D_{i-1})/T_{i-1}$ copies of the second schedule at level i-1. We call this schedule $SHORT_i$. The second schedule at level i is built on m_{i-1} copies of the second solution at level i-1 as in the previous algorithm. We call this schedule $FULL_{i-1}$.

The following algorithm is executed whenever P_s succeeds:

$EXA_i := A_s$

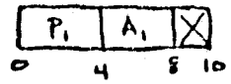
```
for i := 1 to r do
begin
  concatenate 1 SHORT_{i-1} schedule with (D_i-D_{i-1})/T_i
    copies of FULL_{i-1} schedule
  create SHORT_i = MODIFY(A_i-EXA_i, A_i+P_i-EXA_i-EXP_i)
  if D_i < D_r
  then begin
    concatenate m_{i-1} copies of FULL_{i-1} schedule
    FULL_i = MODIFY(A_i, A_i+P_i)
  end
end
```

Let us consider the use of this algorithm in the previous example. Recall that A_1=4, P_1=4, T_1=10, A_2=5, P_2=7, T_2=30, A_3=6, P_3=8 and T_3=60. The following schedule was produced for these jobs by the previous algorithm:

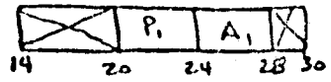


At time 14, P_1 succeeded. We have s=1, EXA_2=2 and EXP_1=4. All other EXA and EXP values are 0. D_1=20, D_2=30, D_3=60, R_1=6, R_2=16 and R_3=46.

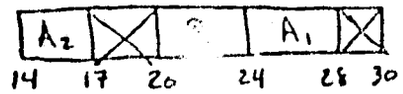
For i = s = 1, the algorithm produces 6 units of IDLE for the SHORT schedule and the following FULL schedule:



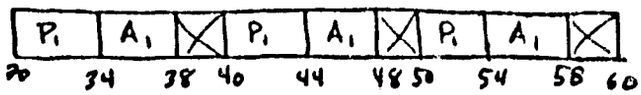
For i = 2, the first concatenation yields:



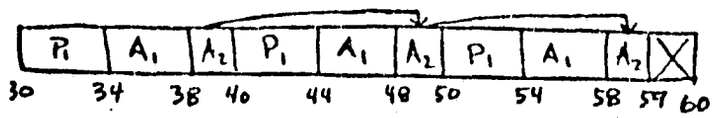
The $SHORT_2$ schedule is:



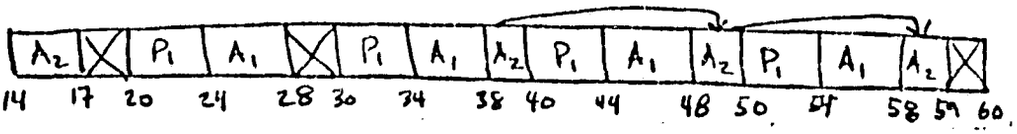
The second concatenation yields:



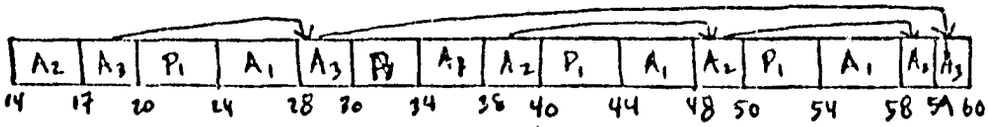
The $FULL_2$ schedule is:



For i = 3, the first concatenation yields:



The $SHORT_3$ schedule is:



The net effect of the new algorithm on this example is to add the execution of a P_1 in the interval 50-60.

Theorem: The schedule produced by the above algorithm is f-t optimal and has as much idle time as any f-t optimal schedule.

Proof: follows easily from the first Theorem.

Theorem: The above algorithm creates a f-t schedule in $O(rM_{r-1})$ time.

Proof: follows from the second theorem.

Using the above algorithms we can create an initial fault-tolerant schedule for a set of jobs J. The jobs can then be executed as scheduled. When a primary algorithm succeeds we can create a new schedule which may allow

more primaries to be executed. In all cases, the schedule produced includes as many primaries as any other schedule which guarantees that the deadlines will be met.

4 Work in Progress.

Currently we are investigating another algorithm for rescheduling after primary success. This algorithm creates a new schedule by making only local changes in the existing schedule.

We are also investigating the effect of different T_1 values on a given set of jobs. Given the A_i, P_i and m_i values, the number of primaries which are scheduled depends on T_1. If T_1 is chosen to be large enough, all of the primaries will be scheduled by the scheduling algorithm. If the primaries always succeed, a much smaller value of T_i will allow all primaries to be executed due to our rescheduling algorithm. We would like to be able to determine this smaller value of T_i and to investigate the behavior of the rescheduling algorithm in cases where a small number of primaries may be expected to fail.

5 References.

[Campbell, Horton, & Belford, 79] Campbell, R.H., K.H. Horton and G.G. Belford, "Simulations of a Fault-Tolerant Deadline Mechanism", Proceedings of the 1979 International Symposium on Fault-Tolerant Computing, June 1979.

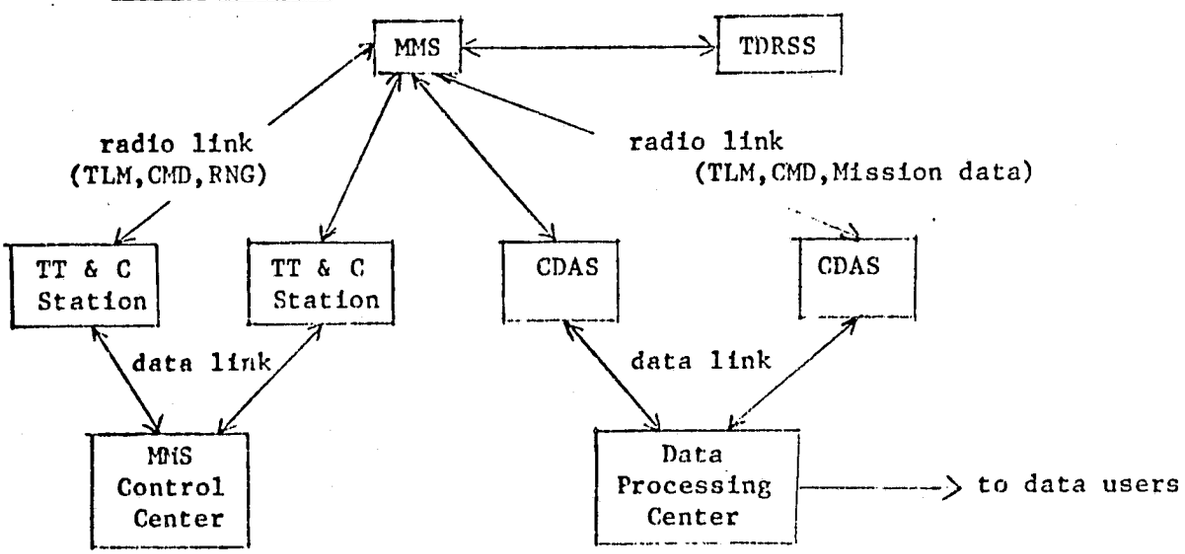
[Randell, 75] Randell, B., "System Structure for Software Fault Tolerance", IEEE Transactions on Software Engineering, Vol. SE-1. No. 2, pp. 220-232, 1975.

1 MMS Concept.

1.1 MMS (Multi-mission Modular Spacecraft).

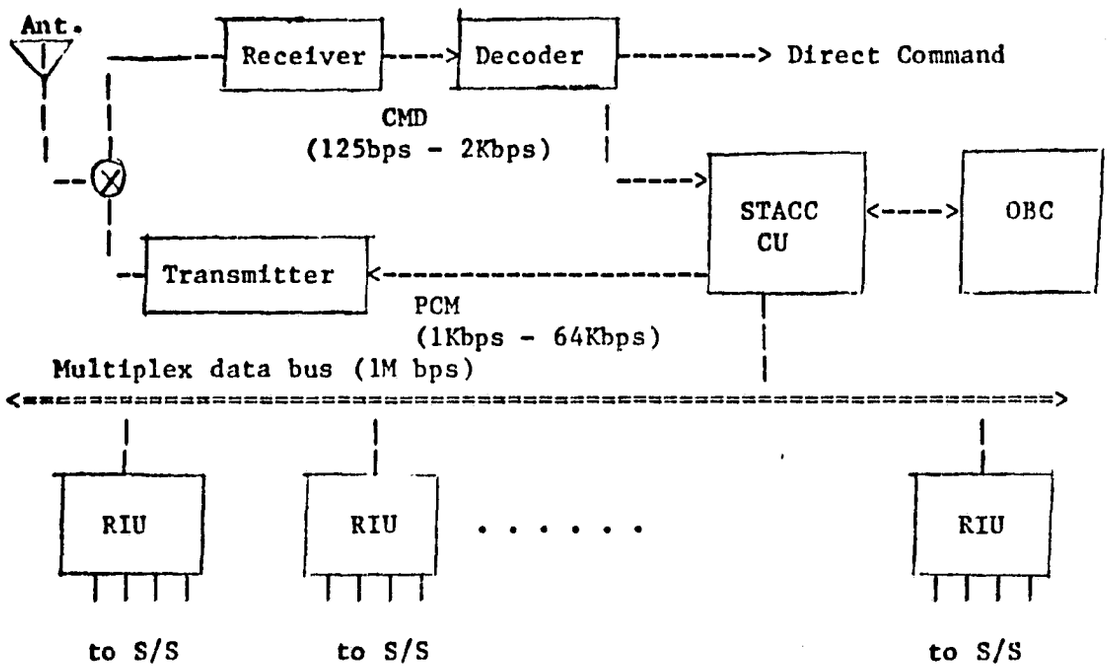
* 1980's program designed by NASA [NASA S-700-10, 77]
* unmanned satellite
* 1,500 kg at medium altitude orbit
* 1,000 kg at geostationary orbit
* multi-mission : meteorological, communication, remotesensing, scientific, broadcasting, etc.

1.2 System Concept.



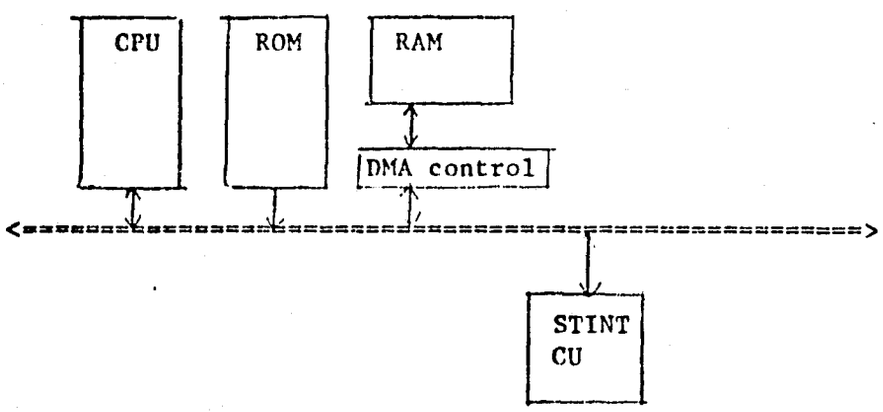
TT & C : Tracking, Telemetry and Command.
CDAS : Command and Data Acquisition Station.
RNG : Ranging.
TLM : Telemetry.
CMD : Command.
TDRSS : Tracking Data Relay Satellite System

1.3 MMS On-Board Data Handling.



STACC : Standard Telemetry And Command Component
CU : Central Unit (Multiplex bus control, OBC interface)
OBC : On-Board Computer
RIU : Remote Interface Unit
S/S : Subsystem
S/C : Spacecraft

1.4 OBC Configuration.



STINT : STACC Interface Unit

1.5 Functions of OBC.

1. Power Management
2. Attitude Control
3. Data Formatting
4. Stored Command Processing
5. Thermal Control
6. Delayed Command Storage
7. Program Loading
8. Command Output
9. Telemetry Data Input
10. Telemetry Format Control
11. Data Output to Real-time Telemetry
12. Data Dumps Direct to the Modulator
13. Direct Computer Access to any Satellite Data Point via the Multiplex Data Bus

2 Simulation of a Simplified MMS Data Handling System.

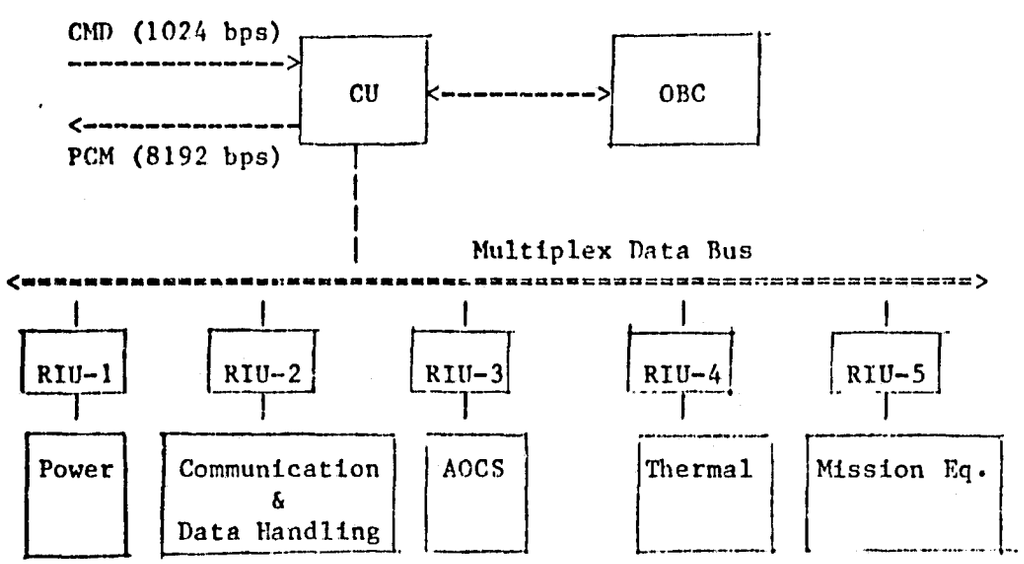
2.1 Objective of the Simulation.

MMS is chosen as a model for the simulation because it is a multipurpose satellite with a real-time executive which is written in the lower level language and developed by NASA.

The objective of this simulation is to show that Path Pascal with the extension of the deadline mechanism is capable to program the real-time software, especially for the flight project.

2.2 Scope of Simulation.

The following diagram shows the simplified MMS data handling system which is of interest for simulation.

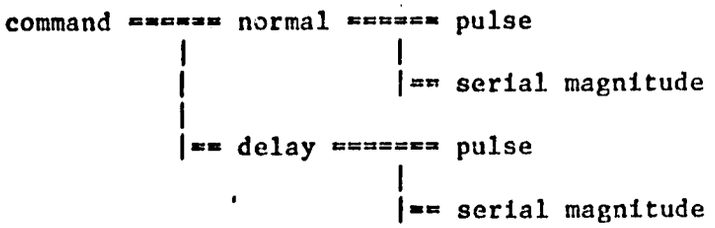


AOCS : Attitude and Orbit Control System
Eq. : Equipment

2.3 I/O signal.

2.3.1 Input command from the ground (1024 bps serial).

a. Command type



normal command : executed at a moment

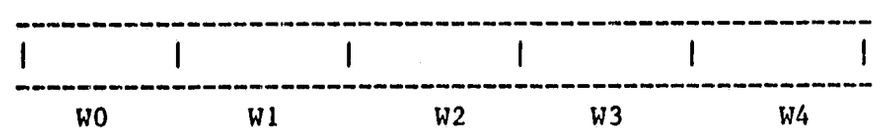
delay command : time taged, executed at the taged time

pulse command : switching command such as turn ON, OFF, etc.

serial magnitude command

: send magnitude

b. Command format



W0,W1 : introduction

W2 : satellite address

W3 : command type I.D.

W4 : command

2.3.2 Output PCM (8192 bps serial).

a. PCM data

* real time data from S/S

* processed data (min.,max.,avg.,etc)

* command answer back signal

* memory dump

b. PCM format

* 1 minor frame : 32 words (16 bits/word)

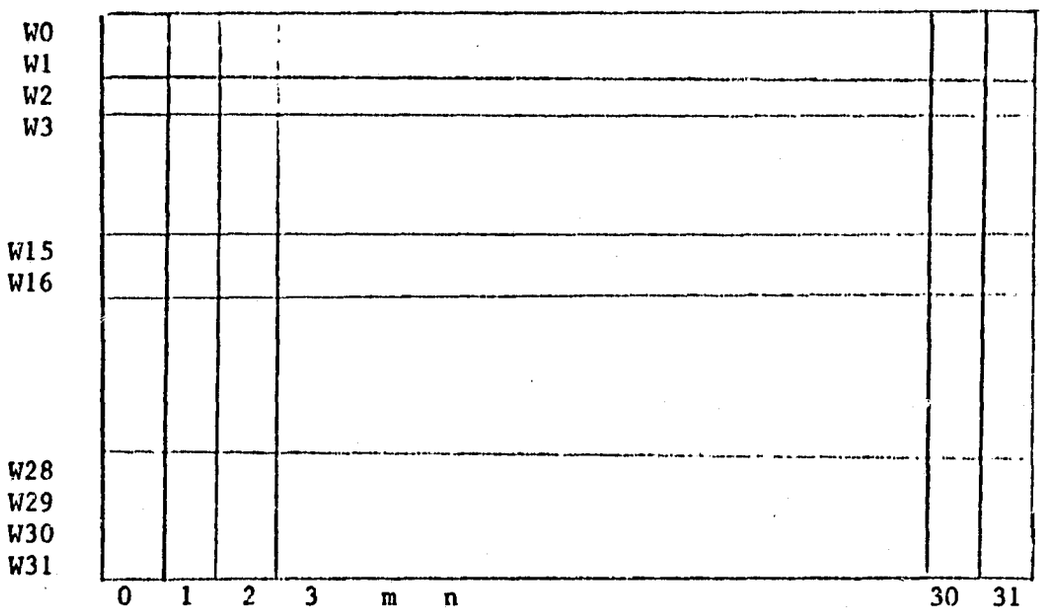
* 1 major frame : 32 minor frames

* word sampling rate : 512 Hz

* commutated word sampling rate : 16Hz

* subcommutated word sampling rate : 1/2Hz

c. major frame



* W0,1 : Frame Synchronization

* W2 : Minor Frame Count

* W15,16: Subcommutated Word

* W28,29

30,31: Subcommutated Word (Computer Processed Data)

* Minor frame (W0 - W31) : 62.5 ms

* Major frame (FRM0 - FRM31) : 2 sec

2.4 OBC Function.

2.4.1 Attitude Control.

a. Read Gyro Signal (every 125 ms)

b. Read Sun and Earth Sensor (every 1s)

c. Attitude Determination

d. Provide Inertial Wheel Control Signal (every 125 ms)

2.4.2 PCM.

a. Telemetry Data Input

b. Limit Checking

c. Statistical Calculation (min.,max.,av.,sdv)

d. Output to Real-time Telemetry

2.4.3 CMD.

a. Delayed Command Storage

b. Stored Command Processing

c. Command Output

2.4.4 Fault Tolerant Deadline Mechanism.

The fault-tolerant deadline mechanism [Campbell, et al., 79] has been implemented into Path Pascal [Campbell and Wei, 79]. It will be applied extensively to AOCS and other periodical processes.

2.5 OBC Data Rate.

2.5.1 Synchronous Data.

Sampling	Acquisition	Distribution	Item
8 Hz	3W		Gyro (AOCS)
		3W	Inertial Wheel (AOCS)
1 Hz	2W		Sun and Earth Sensor (AOCS)
1/2 Hz	1024W		PCM Acquisition
		128W	PCM Distribution

1W = 16 bits

2.5.2 Asynchronous Data.

Asynchronous Command Acquisition and Distribution

REFERENCE

[Campbell, et al., 79] Campbell, R. H., K. Horton, and G. G. Belford, "Simulations of a Fault-Tolerant Deadline Mechanism," The Ninth Annual International Symposium on Fault-Tolerant Computing, June, 1979.

[Campbell & Wei, 79] Campbell, R. H. and A. Y. Wei, "Fault-Tolerant Real-Time Programming in Path Pascal," to be published.

[NASA S-700-10, 77] NASA Goddard Space Flight Center, "Multimission Modular Spacecraft (MMS) System Specification," May 1977.

UIUCDCS-R-79-998

A FAULT-TOLERANT DEADLINE MECHANISM

by

K. H. Horton

December 1979

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

Supported in part by the National Aeronautics and Space Administration under grant US NASA NSG 1471 and submitted in partial fulfillment of the requirements of the Graduate College for the degree of Master of Science in Computer Science.

ACKNOWLEDGEMENTS

This research was supported by NASA grant NSG-1471. I would like to thank Professor Roy H. Campbell for his guidance while directing this research. I would also like to thank Professor Geneva G. Belford and Ira Greenberg for valuable assistance in preparation of earlier reports of this research.

TABLE OF CONTENTS

1 INTRODUCTION

This thesis reports research of a fault-tolerant deadline mechanism, [Campbell et al., 79b] and [Horton et al., 78] are earlier reports. The deadline mechanism can aid in the design of fault-tolerant real-time systems. Separate sections discuss terminology, deadline scheduling, implementation details, the simulation model and applications. Various scheduling algorithms are considered, each with its own advantages and disadvantages. Results from simulations of the deadline mechanism are presented indicating the mechanism's practicality.

A real-time system is designed to provide a service which meets a set of specifications including real-time constraints. Constructing software to meet these real-time constraints is a difficult problem for many applications, such as aerospace control systems and process control. Incorrect design or implementation of the system can cause timing faults which result in failure to meet the real-time constraints. Such timing failures are difficult to avoid, even though program proving and testing techniques are applied. Design of real-time systems which are tolerant to timing faults is currently ad hoc, expensive and difficult. The deadline mechanism can aid in the design of these systems.

The deadline mechanism is based upon the recovery block mechanism [Randell, 75] for fault-tolerant software. Two algorithms are provided for each task which is subject to timing constraints. The "primary" algorithm produces a better quality service than the "alternate". The alternate is a simpler

algorithm which produces an acceptable result in a known, fixed length of time. The reason that recovery blocks cannot be used to provide tolerance of timing faults is that they are insensitive to the passage of time. (That is, it is impossible to recover from a missed deadline by resetting the system clock and executing an alternate algorithm). The acceptance test of the recovery block is replaced in the deadline mechanism by a centralized scheduler and supervisor. The supervisor provides fault-tolerance by detecting timing errors in the primary and switching to its alternate. The deadline mechanism is orthogonal to recovery blocks and the mechanisms may be nested in a complementary manner.

The deadline mechanism has a variety of applications. Completely redundant algorithms may be specified to maintain full system services. The "quality" of the service performed can be gracefully degraded without producing timing failures. Load shedding during periods of high load can be programmed in a structured manner. In addition, the mechanism allows a flexible approach to system reliability. It permits algorithms which provide a very desirable service, but may contain timing faults, to be used in systems which must have a reliable real-time performance. Modifications may be made to maintain the system software without having the system become susceptible to a timing failure. Finally, time-dependent diagnostic routines may be scheduled by the mechanism during periods of low load.

2 FAULT-TOLERANT REAL-TIME SYSTEMS

The deadline mechanism is designed to permit the construction of fault-tolerant real-time systems. We now define this terminology more carefully, and introduce important characteristics of real-time systems for which the deadline mechanism is suited. The definitions of system, reliability, and fault-tolerance follow those of [Randell et al., 78] and [Hecht, 76].

A system is defined as a set of components (together with their interrelationships) which is designed to provide a specified service. (The components themselves may also be systems.) This service is regarded as being provided to one or more environments. An interaction between a system and one of its environments occurs in the following pattern. The environment requests a service from the system; the service is then performed by some component of the system, which is called the service component; and a response is made to the environment. Parts of this pattern may be implicit in a particular interaction. The internal state of a system is a summary of the states of its components.

Real-time programming concerns programs whose validity depends on the execution speed of the utilized processors [Wirth, 77]. A real-time system is a system in which the validity of some of its services depends on processor speed. One of the components of a real-time system is a system clock. System time is part of a real-time system's internal state. It is assumed that consistent views of the time in external environments can be obtained as functions of the state of the system clock.

Timing constraints are imposed on a system by its specification. These constraints are expressed in terms of response periods and arrival periods. A response period is the maximum allowable amount of system time that can elapse from a request time until a response time for a particular service. Thus a response period is a specification of a service that the system must provide. A request time is the system time when a request is detected by the system, and a response time is the system time when a response is completed by the system. An arrival period is the minimum interarrival time of requests for the service, measured with respect to the system clock. Thus the arrival period is an assumption made in the specification of the service.

A deadline is the system time by which a system must respond to a request. Given a request for a particular service, the deadline is calculated by adding the service's response period to the request time.

An execution period is the maximum amount of system time required to execute a particular block of instructions, assuming that it does not contain residual faults and is not interrupted. This is a measurable rather than a supplied quantity and is determined for a particular block of instructions executing on specified processors [Schaefges, 78]. The execution period of a service component is called the service period. The deadline mechanism is designed for services whose arrival periods are greater than their response periods, which are in turn assumed to be greater than their service periods.

These quantities are illustrated in Figure 1. Note that the i^{th} request for the service is processed in less system time than is indicated by the service period. Also, the service period could be repositioned within the response period, or even split into several pieces because it is smaller than

the response period.

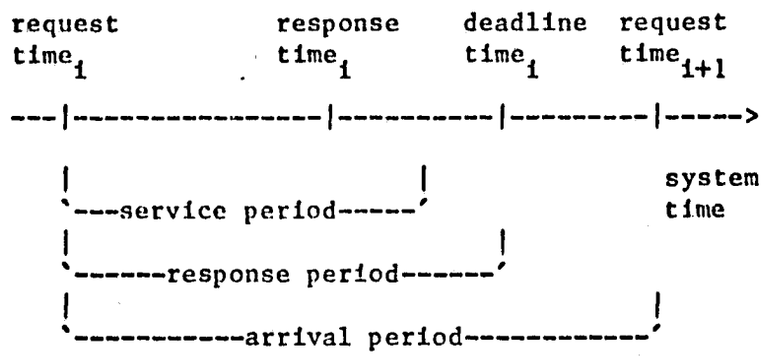


Figure 1. Relationship between timing quantities.

It is difficult to determine the correctness of large, complex systems. Instead, it is common to use the concept of reliability, which is a measure of the success with which a system conforms to some authoritative specification of its behavior. The problem of establishing correctness or reliability in real-time systems is compounded by the time-dependent nature of service components. Timing reliability is defined as the reliability with which a real-time system conforms to the timing constraints in its specifications. If a real-time system meets all of its timing constraints it is characterized as timely.

When a system deviates from its specified behavior, a failure is said to occur. A failure can ultimately be traced back to a fault, which is a mechanical or algorithmic construction that causes an error. An error is that part of an internal state of a system which is incorrect. An internal state of a system is called an erroneous state if there are circumstances (within the

specification of the use of the system) in which further processing, by the normal algorithms of the system, will lead to a failure which is not attributed to a subsequent fault.

The following definitions can then be made for a real-time system. A timing failure is a failure which occurs when a real-time system violates one of the timing constraints in its specification. A timing fault is a fault which causes a timing error, and a timing error is incorrect information about timing constraints in an internal state of the system. A timing error might be identified by information such as the system time, the instructions being executed, the point of execution, outstanding deadlines, etc.

A variety of timing faults can lead to a timing failure. One example is the execution of a service component whose service period has been miscalculated and is larger than the response period. In this case, a timing failure could occur even though all computations were correct. Another example is the execution of a service component in which a repetition bound is miscalculated. This could cause an unanticipated amount of system time to elapse. Precise detection of timing errors may be expensive, but watchdog timers provide a practical detection scheme. Such a timer, together with error confinement and repair techniques, based on recovery blocks, can be used to design fault-tolerant real-time systems.

A system can be designed to be fault-tolerant by incorporating additional computations and abnormal algorithms which attempt to ensure that occurrences of erroneous states do not result in later system failures. A fault-tolerant real-time system is a fault-tolerant system which contains additional computations and abnormal algorithms which attempt to ensure that the occurrences of

erroneous states do not result in timing failures. The deadline mechanism is an example of an abnormal algorithm which incorporates additional computations to provide fault-tolerance for timing faults.

For a real-time system to be timely, there must be some restriction on the arrival, response, and service periods. Given a single processor, arrival and response periods of the same length and constant interarrival times, a necessary and sufficient condition for the existence of a schedule which meets the system's deadlines is given by condition 1.

> Condition 1:
> The service period divided by the arrival period, summed over all components, is less than or equal to one [Liu & Layland, 73].

This constraint is, in general, too restrictive for the situations where the arrival period is a minimum interarrival time or the response period is less than the arrival period. It does provide, however, an easily computed upper bound on the ability of a timely system to process service components with deadlines.

3 DEADLINE SCHEDULING

Many recent papers discuss the question of optimal scheduling to meet a set of deadlines, e.g. [Liu & Layland, 73]. Generally the determination of an optimal schedule is a hard problem except in a few special cases, too slow to be of value in a real-time system. We can however consider heuristic approaches to feasible schedules. Two methods of deadline scheduling have been considered and are described below.

3.1 Earliest-Deadline-First Scheduling

The first approach is the earliest-deadline-first algorithm. Earliest-deadline-first selects the task having the nearest deadline for execution, running it to completion or until the arrival of a task with an earlier deadline. In the latter case the executing task is preempted in favor of the newly arrived task. It has been shown that this algorithm can schedule a set of tasks whenever a feasible schedule exists (if no feasible schedule exists the tasks cannot be scheduled to meet the deadlines by any technique) [Liu & Layland, 73].

3.2 Rate-Monotonic Scheduling

The second method considered is the rate-monotonic algorithm. This approach assigns each task a static priority dependent only on the response requirements of the task. Those tasks with short response times receive high priority, those with longer response times obtain a lower priority. The algorithm executes the highest priority task until a task of higher priority

arrives. It has been shown that the rate-monotonic algorithm can schedule a system if some feasible schedule exists, and the processor utilization is sufficiently low. The allowable processor utilization is a function of the number of tasks but reaches a limit of ln 2 (approx. .693), again see [Liu & Layland, 73]. Thus processor utilization should be kept below 69% if rate-monotonic scheduling is to be applied.

4 DEADLINE MECHANISM

4.1 Primary and Alternate Algorithms

The deadline mechanism requires each service component to have a primary and alternate algorithm. The primary algorithm provides a service which is in some sense more desirable. The alternate algorithm meets the specifications for that service component but may be less desirable. A scheduling algorithm ensures that each service request is satisfied by at least one of the two algorithms.

Reliable scheduling of primaries or alternates to meet real-time constraints requires the calculation of the execution period of each algorithm. This bound may be determined by a theoretical computation from the terminating conditions of the algorithm provided that 1) the use of repetition and recursion constructs that are unbounded is prohibited (for example, the "while loop" of Pascal (see [Anderson & Witty, 78]), or 2) assertions about repetition and recursion constructs are included in the algorithm and, if these assertions are found to be incorrect at run-time, the algorithm is terminated abnormally. In this last method, recovery blocks or a forward recovery mechanism may be used to provide fault-tolerance for incorrect assertions. (Note: the use of recovery blocks requires that the execution period include time for the acceptance test to be executed repeatedly and every alternate to be executed, as well as any overhead to restore variables from the recovery cache.) The accuracy of the determination of the execution periods is critical to system performance and reliability.

The deadline mechanism can schedule alternates using several basic algorithms. The two simplest are the rate-monotonic algorithm and earliest-deadline-first. The rate-monotonic algorithm uses a fixed priority scheme and requires a somewhat restrictive limit on processor utilization. The earliest-deadline-first algorithm can schedule a system if condition 1 (above) is satisfied. Execution of the primary and alternate within the service period can then be ordered in several different ways: primary before alternate, primary after alternate, primary and alternate in parallel or primary and alternate interleaved. The simulations, however, consider a single-processor system in which the primary is run either before or after the alternate.

If the primary completes within its execution period, its results are used in preference to those of the alternate. If the primary should fail to complete within its execution period, because of a timing fault in the primary or a miscalculation of the execution period, the results from the alternate are used. If the alternate is run before the primary, a cache may be used to hold results from the alternate until the primary either fails or completes successfully.

When the service period exceeds the sum of primary and alternate execution periods, both algorithms can be executed on a single processor, providing full redundancy. If the service period is greater than both execution periods (but perhaps less than their sum) a single processor can run either primary or alternate, but probably not both. A multiprocessor could, however, execute them in parallel and again provide full redundancy. Lacking an upper bound for the primary, both single and multiprocessor systems can at best provide only partial redundancy.

Requiring that the execution period of a primary be accurately determined restricts the algorithms that may be used as primaries. However, there are many applications where this requirement is overly restrictive and would prevent the use of very desirable algorithms. Removing this requirement allows primary algorithms whose average execution time is "small" but that have execution periods which 1) are very large, 2) cannot be determined accurately or 3) cannot be determined at all.

This admits primaries which have unknown execution times. The "service period" of a component can then be set, for scheduling purposes, at any figure that is less than the response period, but no less than the execution time of the alternate. Since in general the system may be unable to execute both the primary and the alternate, one must be allowed to complete. Since upper bounds are assumed to exist for alternates only, the deadline mechanism is designed to reserve a time for execution of the alternate. A primary scheduling algorithm is then applied to schedule primaries in any remaining time. This time is called slack time. The following section describes several approaches to scheduling primaries and alternates and indicates the conditions under which each may be applied. Specific combinations of primary/alternate scheduling algorithms and the resulting simulated performance (how effectively the slack time is used to run primaries) are then discussed.

4.2 Scheduling of Alternates

The first-chance scheduler selects alternates using the earliest-deadline-first algorithm. Alternates with deadlines closer to the current system time have higher priority and preempt lower priority alternates. Primaries are scheduled to execute in slack time after their alternates have been

completed. The first-chance scheduler allows a maximum number of service components to share a single processor. (The maximum number of service components is determined by condition 1 applied to the situation where all services are provided by alternates.) The results from the computation of the alternate must be retained until either the primary completes successfully or the deadline is reached. The results from the alternate could also be used as an acceptance test of the results from the primary.

The rate-monotonic scheduler selects alternates using the response period as a static priority, with small values having high priority. At any given instant the alternate with the highest priority (smallest response period) will be executing. One potential advantage of this method is a possible reduction in scheduling complexity and overhead, an aspect not considered in the simulations. Again, slack time is used for the execution of primaries.

The last-chance scheduler selects alternates using a modified earliest-deadline-first algorithm in which alternates may not preempt each other. The primaries are scheduled to execute in slack time, prior to their alternates. Whenever a request for service occurs, the alternate scheduler reserves processor time to execute the alternate. This time is scheduled at the last possible instant that the alternate can be executed to complete before the deadline. When alternate execution periods overlap, the alternate with the earliest deadline is scheduled first. The alternate scheduler may need to reschedule alternates as later service requests are received. In particular, an alternate may be rescheduled to run earlier as a result of a new request for a service. This occurs if the deadline for the new request causes potential overlap of the executions of the new alternate and the original

alternate and the original alternate has the earlier deadline. An alternate preempts a primary which is executing if the schedule of the alternate demands that it be run.

The last-chance scheduler organizes the primary and alternate executions so that if the primary succeeds, the alternate does not run. If a primary successfully completes before its alternate is scheduled to run, the slack time is incremented by the execution period of the alternate. Since alternates are scheduled to run at the last instant and may not be preempted, a more stringent requirement on system service periods (execution periods of the alternates) is needed to guarantee the timeliness of the system:

Condition 2:
The service periods, summed over all components, must be less than or equal to the minimum response period of the system.

Condition 2 is demonstrated by the following argument. Suppose a system has n components $\{c_1,..,c_n\}$ ordered by non-increasing response period (c_n will have the minimum response period). Further suppose that requests have arrived for components $c_1,...c_{n-1}$ so the last chance schedule completely occupies the processor for some interval starting at time t. Finally suppose a request for c_n arrives at t. Now the schedule for $c_1,...,c_{n-1}$ is last chance, so if any are rescheduled at a later time deadlines will be missed. Thus c_n cannot start before any of $c_1,...,c_{n-1}$. If it starts immediately after $c_1,...,c_{n-1}$ finish it will complete only after its service period. The earliest it can complete is t + sum of service periods of $c_1,...,c_n$. If c_n is to be timely the sum of the service periods must be less than or equal to the response period of c_n (which by hypothesis is the minimum). Since all components of a system must

be timely for that system to be timely a timely system must satisfy condition 2.

4.3 Scheduling of Primaries

Either of two primary scheduling algorithms may be used with the alternate scheduling strategy. The simple primary scheduler may select primaries to run in several ways: for example, in deadline order, round robin, least recently run, or according to the alternate scheduling strategy. The importance primary scheduler provides preferential treatment to "important" primaries. In many applications, the service provided by some service components of the system is more important with respect to the specification of the system than that provided by other components. For example, it may be more important for a spacecraft to fire retro-rockets at the correct time than to send sensor data back to earth every second. Importance may not be related to the arrival period or response period of the service, as shown by the spacecraft example. (The arrival rate of requests for firing the rockets may be once a trip.) The importance primary scheduling algorithm allows the services of a system to be given a ranking of importance. During slack time, the algorithm schedules the primaries in order of that importance. For primaries which have the same importance level, the scheduler may use strategies similar to the simple primary scheduler.

In comparing the schedulers, two important measures are the fractions of 'cpu' time wasted by 1) executing alternates whose results are never used and 2) abandoning primary executions. A simple simulation model of the deadline mechanism was developed in order to compare the scheduling algorithms.

5 SIMULATION MODEL

The simulations are based upon a simple model of a real-time system and provide preliminary information about the deadline mechanism. The first simulation provides a control for the later simulations: it measures the timeliness of the system without the deadline mechanism, using primaries scheduled in earliest-deadline-first order. The second simulation includes the deadline mechanism and alternates to make the system timely. The last-chance scheduling algorithm is used together with the simple primary scheduler. In the third simulation, the primaries are scheduled instead by the importance primary scheduler. Another simulation compares the first-chance and last-chance scheduling algorithms. A final set of simulations compares the first-chance and rate-monotonic schedulers for several different mixes of tasks. In each simulation run approximately 1500 requests occurred for each service component.

5.1 Simulation Program

The Simula programming language [Dahl et al., 68] was used to simulate the deadline mechanism using the various scheduling algorithms. The program was written and run on a Control Data Corporation Cyber 175. A skeletal version of the program appears as Appendix A. The basic structure and role of each part of the program is outlined below.

The mainline code of the program collects the run parameters from input and initializes the appropriate number of task initiators ('taskinit'). Run parameters consist of task descriptions including response period, request

period, primary and alternate execution periods, and distributions to be used. Additional system parameters include the request load and simulation period. The mainline code then creates the queues to be used during the simulation. The remaining duty is to wait for the simulation period to expire and report the simulation statistics.

The 'sked' class simulates the cpu scheduler. Together 'sked' and the queue maintenance routines ('intopriq', 'intoaltq', 'outaltq') embody the scheduling algorithm.

The 'taskinit' class simulates the request source and accumulates statistics for the report. 'taskinit' waits the request period (a random variable) and then initiates a new 'prialg' and 'altalg' (described below). The request period is scaled by the request load parameter to vary the request rate from 0 to 100% of maximum value. The random variables for 'prialg' and 'altalg' execution times are also computed and passed to the 'prialg' and 'altalg'. The newly created 'prialg' and 'altalg' are entered into the proper queues and the simulated 'cpu' is 'interrupted' if required.

When initiating tasks certain parameters (interarrival, primary and alternate execution time) are assigned random values according to a distribution. Four distributions are available:

1) Constant.

2) Negative exponential plus constant.

3) Uniform over a given interval.

4) Poisson distribution.

The 'prialg' and 'altalg' represent the primary and alternate algorithms for the service component. They are distinct processes in the simulation to allow the possibility of concurrent execution. These routines also contain the code which records statistics on primary and alternate completions. Internal routines ('intopriq', 'intoaltq', 'outaltq') maintain the primary and alternate queues ('priq' and 'altq') in the proper sequence for selection by 'sked'. To allow convenient keeping of statistics each 'prialg' and 'altalg' contains a pointer to its associated alternate (or primary) and its 'taskinit'. This allows the 'taskinit' class to record the common statistics. In addition the pointer allows an alternate algorithm to preempt (or cancel) its associated primary when using the last-chance algorithm.

The report routine summarizes the simulation by reporting these statistics for each task:

1) The fraction of idle 'cpu' time.
2) The fraction of 'cpu' time consumed by completing primary algorithms.
3) The fraction of requests satisfied by primary algorithms.
4) The fraction of 'wasted' time (that time consumed by alternates with corresponding primary completions and consumed by uncompleted primaries).

The statistics used in the comparisons below are averages of the above quantities over all tasks in the particular run.

5.2 Scheduling Algorithms in Simulation Program

The last-chance scheduler selects alternates from the 'altq' when that alternate must be started in order to meet its deadline (at the "last chance"). Because this is the last chance, preemption could cause the dead-

line to be missed and consequently is not allowed. The 'altq' is maintained in deadline order by 'intoaltq' and 'outaltq' keeping scheduled start and stop times for each alternate, and in case these times should overlap the earlier start and stop times are adjusted to even earlier values. In a real system the overhead of this approach may prove to be excessive, however, these simulations assume scheduling is 'instantaneous'. Primaries are selected from the 'priq' (ordered by deadline time) for execution during slack time. Should the primary complete before its deadline the alternate is removed from the 'altq', the 'altq' may be rescheduled to use the released time, and a primary completion is recorded. When an alternate is executed, it will complete before the deadline. At this time the uncompleted primary is removed from the 'priq' and an alternate completion is recorded. When new requests arrive the new primary and alternate are entered into appropriate queues, the 'altq' is rescheduled if required, and any executing primary is preempted.

The first-chance scheduler selects and executes alternates from the 'altq' until none remain. The 'altq' is maintained in deadline order, but start and stop times are not maintained in contrast to last-chance. When no alternates remain primaries are selected from the 'priq', again maintained in deadline order. When a primary completes before the deadline a primary completion is recorded. When the deadline arrives (only when the primary has not completed) the primary is terminated if executing, is removed from the 'priq' and an alternate completion is recorded. When a new request arrives the new primary and alternate are entered into their respective queues and, if required, the executing task is preempted.

The rate-monotonic scheduler follows the same basic algorithm as the first-chance scheduler, the difference being that both 'priq' and 'altq' are maintained in order of increasing response periods rather than deadline order.

Appendix B provides skeletal programs for the 'sked' class and queue maintenance routines for each scheduling algorithm simulated.

5.3 Parameters of the Simulations

The simulations measure the change in the behavior of a simple real-time system as the rate of requests for services varies. The request load is defined as the minimum interarrival time (the arrival period) expressed as a percentage of the average interarrival time. Thus, a request load of 100% represents requests which are arriving at the maximum rate, and a request load of 50% represents requests which are arriving at half the maximum rate. The actual interarrival times have exponential distributions with origin at the arrival period and mean given by the arrival period multiplied by 100 divided by the request load.

Parameter	Value
Service components	10
Arrival period (AP)	100 ("time units")
Response period	100 " "
Average primary duration	10, 12, 20 " "
Service period	10 " "
Request load (RL)	10% by 5% to 100%

Interarrival times have a distribution of:

AP + expdist(mean = AP * (100 / RL - 1))

Figure 2. Summary of simulation parameters

The parameters in the simulations were chosen so that at a request load of 100% the processor has no idle time. The service period used for each service component is the execution period of the alternate, and is constant. At a request load of 100%, the service periods account for all of the available "processor" time.

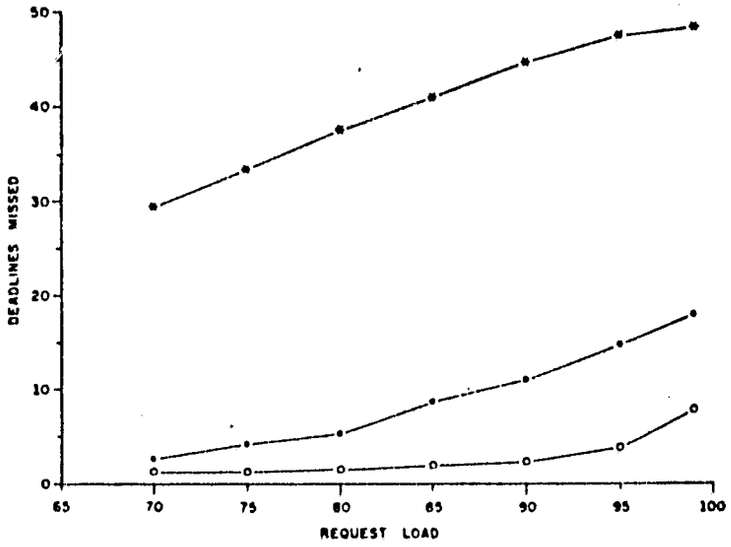
The response period for each service component is a constant chosen to allow the system to be timely (condition 1) at 100% request load (i.e., when requests arrive at the maximum rate).

The execution periods of primaries are assumed unbounded. (The primaries might contain indefinite repetition or recursion.) The actual duration of each primary execution is determined from an exponential distribution with the "average execution duration" as its mean. The execution periods of the alternates are constant.

In the simulation of importance levels, the model divides the primaries into three levels of importance. Three of the services have a low level of importance, four have an intermediate level of importance, and three have a high level of importance.

5.4 Simulation Model Results

First, the behavior of the system without the deadline mechanism is presented. Figure 3 shows the percentage of deadlines that are missed by the system for three cases: when the average duration of primary executions 1) equals the service period, 2) exceeds the service period by 20%, and 3) exceeds the service period by 100%. In case 1, only a small fraction (less than 5%) of the primaries fail to meet deadlines even at a request load of

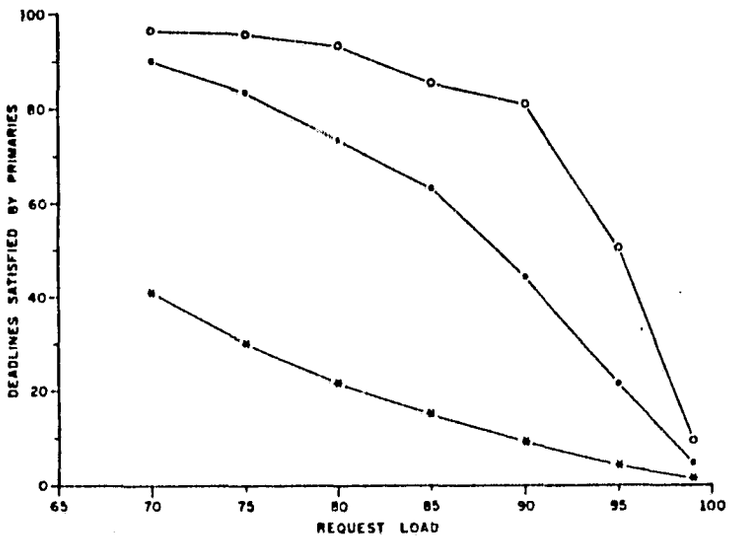


Legend: o case 1; average primary duration = 10
. case 2; average primary duration = 12
* case 3; average primary duration = 20

Figure 3. Percent deadlines missed without deadline mechanism.

95%. (When request load is reduced to 10%, approximately 2% of the deadlines are still missed because of the exponential distribution of execution time.) In cases 2 and 3, however, the failure rate increases significantly as request load increases.

Next, the behavior of the same system is measured with the deadline mechanism and alternates included. The last-chance algorithm is used to schedule the alternates and the simple primary scheduler to schedule the primaries. Even though fewer primaries are completed than in Figure 3, due to the deadline mechanism no deadlines have been missed. The question of interest is how many deadlines are met by primaries, as opposed to alternates. In Figure 4, the percentage of deadlines satisfied by primaries is plotted against request load for cases 1, 2 and 3 above. In each case, the fraction of deadlines met

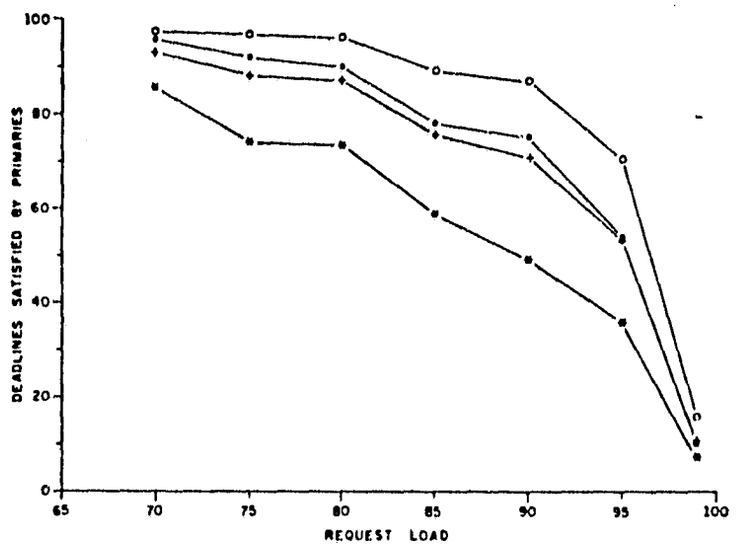


Legend: o case 1; average primary duration = 10
. case 2; average primary duration = 12
* case 3; average primary duration = 20

Figure 4. Percent deadlines met by primary algorithms using a single level of importance.

by primaries decreases as the request load increases. It is not surprising, moving from case 1 to case 2 to case 3, that the fraction of deadlines met by primaries decreases more rapidly as request load increases. Inspection of the number of deadlines met by primaries for each service component reveals that for this particular system they are "fairly" (i.e., uniformly) distributed among the components. The percentage of "processor" time wasted on executing primaries which are aborted is quite low. In case 1, the maximum percentage of wasted time is less than 6% for the various request loads simulated.

Third, the behavior of the system is measured when the service components are split among 3 levels of importance. In this simulation the average primary duration equals the service period (i.e., is set equal to 10 time units). Figure 5 shows the relationship between the percentage of completed primaries

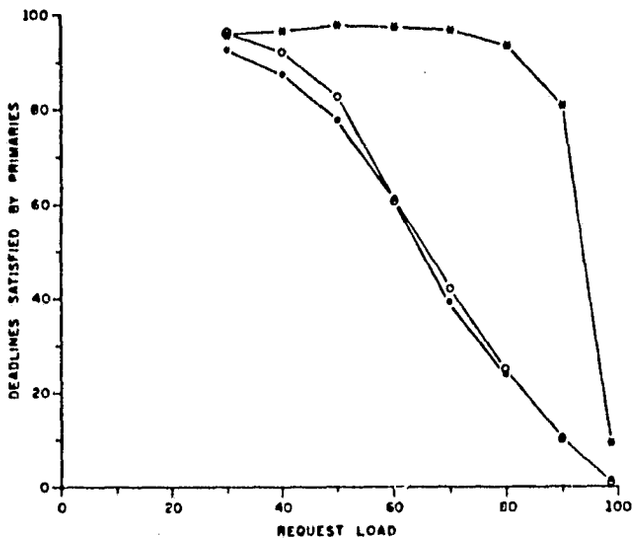


Legend: o high importance . medium importance
* low importance + overall (average)

Figure 5. Percent deadlines met by primary algorithms using 3 levels of importance.

at each level of importance and the request load. Those components of high importance consistently receive a large percentage of primary completions. Those components of medium importance receive a percentage of primary completions roughly equal to the percentage of primary completions which occur using the simple primary scheduler. Those components of lowest importance receive a much lower percentage of primary completions. The curve of overall percentages of deadlines met by primaries is somewhat lower than the comparable curve (case 1) of Figure 4, since the simple primary scheduler uses the more effective earliest-deadline-first strategy.

Finally, the behavior of a system employing the first-chance algorithm is compared with that of a system employing the last-chance algorithm. Both



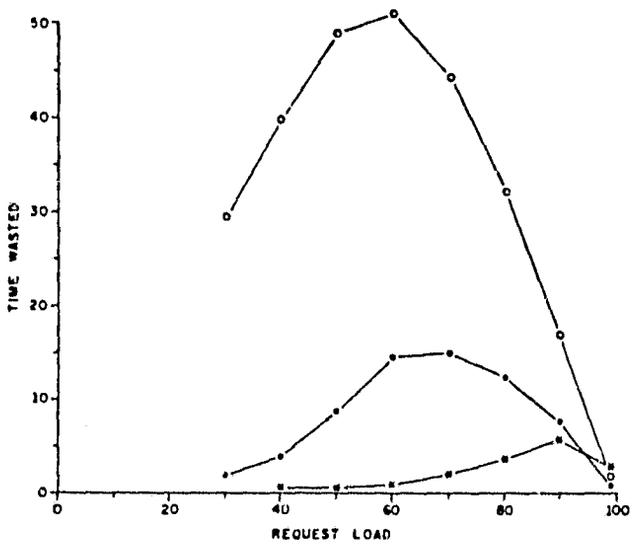
Legend: * last-chance; identical services
o first-chance; identical services
. first-chance; mixed services

Figure 6. Percent deadlines met by primary algorithms using first-chance and last-chance scheduling.

systems use the simple primary scheduler with average primary durations of 10. In Figure 6, the percentage of deadlines satisfied by primaries for each system is displayed against increasing request load. (See curves labelled "identical services".) The system using the first-chance algorithm satisfies fewer deadlines by primaries than the system using the last-chance algorithm. The last-chance algorithm, however, requires that condition 2 must hold for the service periods.

The behavior of a third system is also displayed in Figure 6. The third system uses the first-chance algorithm and has eight service components with arrival and response periods of 100, and one service component that may be requested at twice the rate of the other compon (i.e., arrival and

response periods of 50). This system does not satisfy condition 2, so that the last-chance algorithm could not be employed. (Note that a last-chance algorithm could be used if the number of components with response period 100 is reduced from eight to four.) Comparison of the curve for this third system with the first-chance curve for identical components suggests that the effect on performance of one component that is twice as active is comparable to two components of normal activity.



Legend: * last-chance; primary time wasted
. first-chance; primary time wasted
o first-chance; total time wasted

Figure 7. Percent time wasted by abandoned primary algorithms and redundant alternate algorithms.

Figure 7 displays the percentage of time wasted for the two systems of Figure 6 with identical components. The system employing the last-chance

algorithm wastes time on primaries which are not completed. The first-chance algorithm has wasted time for both primaries and alternates. The wasted time for alternates occurs when alternate results are discarded and explains the difference in behavior of these two systems as displayed in Figure 6. When the alternate execution periods are small compared to the primary execution periods (which is likely to be true for real systems) the wasted time for alternates will be less significant and the performance of the first-chance algorithm should be competitive with last-chance.

The first-chance and rate-monotonic schedulers were compared by a set of four simulations. Initially a set of identical tasks was simulated, but the results of both schedulers were exactly the same. This was explained because with identical response times the rate-monotonic scheduler became first-in first-out, identical with the deadline order used by the first-chance scheduler.

Consequently the next simulations consisted of several classes of tasks. The first simulation consisted of several tasks with differing response requirements, but with service periods in constant proportion to response period. This simulates a system in which small tasks have s short response period, while longer tasks have a larger response period. The second simulation held service periods constant but varied response requirements. This might represent a system performing some fixed service to differing response requirements (e.g., servicing interrupts for several terminals operating at different baud rates). In the third simulation the ratio of service period to response period decreased as response period increased. This parallels a situation in which a complex computation requires a fast response and slower

response is required by less complex computation. For example a real-time control application may have a control function requiring fast response and complex computation, while the logging function has simple processing (formatting output for a hardcopy device) with a slower response requirement. Finally the last simulation had service period response period ratio increasing as response period increased. This could represent a time-sharing system in which the response requirement (and processing complexity) for terminal i/o interrupts is smaller than the response from a user program processing a line of input.

The result of simulating each task mix using first-chance and rate-monotonic schedulers compared quite favorably so long as the processor utilization was below .69. As utilization rose above that limit the rate-monotonic schedule began to miss deadlines. So long as the utilization limit is observed the first-chance and rate-monotonic schedulers appear to be equally effective. The fixed priority of the rate-monotonic scheduler is an advantage in a system with limited processor utilization.

The simulation results have encouraged us to implement the deadline mechanism in an experimental version of Path Pascal [Campbell et al., 79a]. Programming actual applications using the deadline mechanism will further test the concept.

6 APPLICATIONS

The deadline mechanism can be used to provide completely redundant algorithms in order to maintain full system service or graceful degradation of service without producing timing failures. Fault-tolerant systems with these properties are required in many applications where immediate human intervention is impossible. Examples of such applications occur in aerospace, control systems, process control, computer networks and telecommunications. It is interesting to note that many hardware subsystems implement a mechanism analogous to the deadline mechanism. The hardware may provide a timeout for the receipt of the next command. Should it fail to arrive within the specified time, the devices performs an "alternate algorithm", usually some function of the last command received (for example, some motorized devices turn off the motor if a command has not arrived before the timeout).

While only demand requests (those derived from asynchronous interrupts) were simulated, the deadline mechanism can be used with periodic requests (derived from the system clock) as well. Many real-time systems are designed using the periodic request technique and a fast-loop/slow-loop/background organization [Hecht, 76]. Such systems form the basis for a series of experiments being conducted with the deadline mechanism.

The deadline mechanism also provides a structured approach to load shedding during intervals of high load by attaching the concept of "importance" to primary algorithms. (One possible use of the alternates of low-importance primaries could be to provide a mechanism to record and notify users of ser-

vices which have to be curtailed because of load.) Within a time-sharing system the alternates of primaries which are affected by load shedding could spool requests for services which cannot be serviced immediately.

It may be extremely difficult to program a service so that a fixed upper bound to the execution time of the service is known. Proof that an algorithm used in the program produces the desired result within some fixed period may require considerable effort and may not be available for a long time after the algorithm is conceived. In fact, for some algorithms there may be no bound on execution time that is valid for all input parameters. By allowing such algorithms to be used as primaries and providing a restricted but safe service from an alternate, such algorithms can be employed in reliable real-time systems.

Maintenance of software is an important cost in most applications where the useful lifetime of the software extends over many years. Modification of parts of the system which must run in real-time is expensive and difficult and frequently leads to timing failures. If alternates are only replaced by algorithms which have been tested in the application over long periods of time as primaries within the structure of the deadline mechanism, a system can gracefully evolve without incurring timing failures.

7 PATH PASCAL IMPLEMENTATION

The deadline mechanism is being implemented in an experimental version of Path Pascal reported in [Campbell & Wei, 79]. This implementation provides each deadline process with two algorithms whose execution is controlled by the service statement. The syntax of a typical application is shown in Figure 8.

```
deadline process [<response>] [<interarrival>];
        procedure priproc... begin ... end;
        procedure altproc... begin ... end;
begin
        repeat
                request <synch>;
                service by priproc;
                        else by altproc;
        until false;
end;
```

Figure 8. An example of deadline mechanism syntax in Path Pascal implementation.

The reserved word deadline flags this process to be scheduled to meet real time constraints. The <response> and <interarrival> parameters are compile time constants which specify the required response time and the minimum interarrival time respectively. These parameters and an analysis to determine execution times can provide automatic checks of the conditions required for a timely system. Procedures 'priproc' and 'altproc' indicate the primary and alternate algorithms. (Note it is the appearance of 'priproc' and 'altproc' in the service statement which indicates their role in the deadline process.) The deadline process body is (in this case) an indefinite repeat loop, await-

ing a request in the <synch> statement and servicing the request by the primary or alternate as appropriate. The <synch> statement could be a P operation on a semaphore, some path synchronized procedure, a 'delay' statement or a 'doio' statement.

In the experimental version the recovery cache is implemented in software and behaves as follows. First the alternate algorithm is executed. Upon completion the recovery cache and normal memory are exchanged. The primary algorithm is then attempted using the pre-alternate values for data. Should the primary fail the "recovery" is made by restoring the post-alternate values from the recovery cache.

The scheduling algorithm used in the Path Pascal implementation is the rate-monotonic scheduler. This fixed priority approach uses the <response> parameter as the priority (small response time corresponding to high priority). So long as processor loading is below roughly 69% this approach can meet all deadlines.

One facet of the Path Pascal implementation is as yet unresolved. This is the issue of inhibiting a deadline process from accepting further requests until the <interarrival> time has passed. Using the Path Pascal standard function 'time' (which returns the current system time) this might be accomplished:

```
repeat
        request <synch>;
        t:= time + <interarrival>;
        service by <primary>;
                else by <alternate>;
        delay( t -  time);
until false;
```

If the nature of the system is such that requests are guaranteed to meet the lower bound on interarrival time this delay would be superfluous. If the bound is not guaranteed, this feature would prevent the system from being swamped with requests arriving faster than the specified maximum rate.

8 CONCLUSION

Many applications of real-time systems require the system to perform services reliably within real-time constraints. Correctness proofs and testing are expensive and may not detect all the residual timing faults in the implementation or hardware. Systems which are tolerant of timing faults provide an approach to reliable real-time systems. The deadline mechanism permits software to be constructed which is tolerant of many varieties of timing faults. The mechanism may be applied in various ways to provide redundancy, graceful degradation and load shedding. In addition, it allows (without jeopardizing reliability) maintenance on time-critical software to be performed and desirable algorithms, which may contain timing faults, to be included in the system.

REFERENCES

[Anderson & Witty, 78] Anderson, T. and R. W. Witty, "Safe Programming," BIT 18, pp. 1-8, 1978.

[Campbell et al., 79a] Campbell, R. H., I. B. Greenberg, and T. J. Miller, "Path Pascal User Manual," Technical Report UIUCDCS-R-790-960, University of Illinois, Urbana, 1979.

[Campbell et al., 79b] Campbell, R. H., K. Horton, and G. G. Belford, "Simulations of a Fault-Tolerant Deadline Mechanism," The Ninth Annual International Symposium on Fault-Tolerant Computing, June, 1979.

[Campbell & Wei, 79] Campbell, R. H. and A. Y. Wei, "Fault-tolerant Real-time Programming in Path Pascal," in preparation.

[Dahl et al., 68] Dahl, O. J., B. Myhrhaug, and K. Nygaard, "The Simula 67 Common Base Language," Norwegian Computer Center, Oslo, 1968.

[Hecht, 76] Hecht, H., "Fault-Tolerant Software for Real-Time Applications," Computing Surveys, Vol. 8, No. 4, pp. 391-407, 1976.

[Horton et al., 79] Horton, K. H., R. H. Campbell, and G. G. Belford, "Meeting Real-time Deadlines," Proceedings of Computers, Electronics and Control, 1978, ACTA Press, Calgary, 1979.

[Liu & Layland, 73] Liu, C. L., and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," JACM, Vol. 20, No. 1, pp. 46-61, January, 1973.

[Randell, 75] Randell, B., "System Structure for Software Fault Tolerance," IEEE Trans. on Software Engineering, Vol. SE-1, No. 2, pp. 220-232, 1975.

[Randell et al., 78] Randell, B., P. A. Lee, and P. C. Treleaven, "Reliability Issues in Computing System Design," Computing Surveys, Vol. 10, No. 2, pp. 123-165, 1978.

[Schaefges, 78] Schaefges, T. M., "Estimating Execution Times of Path Pascal Programs," Masters Thesis, University of Illinois at Urbana-Champaign, 1978.

[Wirth, 77] Wirth, N., "Toward a Discipline of Real-Time Programming," CACM, Vol. 20, No. 8, pp. 577-583, August, 1977.

APPENDIX A: Skeletal Simula Program

```
simulation begin
   process class prialg(imp,exec,parent,inst);
      integer imp,inst;
      real exec;
      ref(taskinit) parent;
      begin


         procedure intopriq;
            begin comment see Appendix B;
            ...
            end *** intopriq ***;

         real entry,elapsed,timeleft;
         ref(altalg) assocalt;

         entry:=time; timeleft:=exec;
         intopriq;

         passivate;

         cancel(assocalt);
         assocalt.outaltq;
         out;
         if time>assocalt.duetime then error(' missed deadline ');
         elapsed:=time-entry;
         inspect parent do
            begin
               pridone:=pridone+1; priexec:=priexec+exec;
               priresp:=priresp+elapsed;
               altwaste:=altwaste+assocalt.exec-assocalt.timeleft;
               if altcnt>0 then
                  begin altrun[altcnt]:=altrun[altcnt]+1; altcnt:=0; end;
               pricnt:= min(10,pricnt+1);
            end;
      end  *** prialg ***;
```

```
process class altalg(exec,resp,parent,inst);
   integer inst;
   real exec,resp;
   ref(taskinit) parent;
   begin


      procedure intoaltq;
         begin comment see Appendix B;
         ...
         end *** intoaltq ***;


      procedure outaltq;
         begin comment see Appendix B;
         ...
         end *** outaltq ***;

      real entry,elapsed,duetime,timeleft,skedstart,skedstop;
      ref(prialg) assocpri;

      entry:=time;  timeleft:=exec;  duetime:=entry+resp;
      skedstop:=duetime;  skedstart:=skedstop-exec;
      intoaltq;

      passivate;

      assocpri.out;
      reactivate cpu;
      outaltq;
      if time>duetime then error(' missed deadline ');
      elapsed:=time-entry;
      inspect parent do
         begin
            altdone:=altdone+1; altexec:=altexec+exec;
            altresp:=altresp+elapsed;
            priwaste:=priwaste+assocpri.exec-assocpri.timeleft;
            if pricnt>0 then
                begin pririn[pricnt]:=pririn[pricnt]+1; pricnt:=0; end;
            altcnt:= min(10,altcnt+1);
         end;
   end  *** altalg ***;
```

```
process class taskinit(imp,pexec,aexec,cycle,resp,tasknumber);
   integer imp,tasknumber;
   real pexec,aexec,cycle,resp;
   begin


     real procedure dist(t,opt);
        real t;
        text opt;
        begin
           dist:= if opt='u' then uniform(ratio*t,t,u0)
                  else if opt='p' then negexp(+1.0/t,u0)
                  else if opt='m' then t+negexp(+1.0/(reqfactor*t),u0)
                  else t;
        end *** dist ***;

      integer pridone,altdone,init,pricnt,altcnt;
      integer array prirun[1:10], altrun[1:10];
      real priexec,priresp,priwaste,altexec,altresp,altwaste;
      ref(altalg) a;
      ref(prialg) p;

loop: init:=init+1;
      p:- new prialg (imp,dist(pexec,popt),this taskinit,init);
      a:- new altalg (dist(aexec,aopt),resp,this taskinit,init);
      p.assocalt:- a;  a.assocpri:- p;
      activate p;  activate a;

      if imp>cpu.priority then reactivate cpu;

      hold(dist(cycle,copt));

      if time<simperiod then go to loop;
   end  *** taskinit ***;



process class sked;
        begin comment see Appendix B;
        ...
        end *** sked ***;


procedure report;
   begin ...  end  ***  report  ***;
```

```
    ref(sked) cpu;
    ref(head) array priq[1:3];
    ref(head) altq,tasklist;
    ref(taskinit) t;
    integer tn,ntasks,imp,i, u0;
    real simperiod,cycle,resp,pexec,aexec,
         cpurate,ratio,fuzz,reqfreq,reqfactor;
    text aopt,copt,popt;

    fuzz:=.005;  u0:= 1;

    copt:-copy(intext(1));  popt:-copy(intext(1));  aopt:-copy(intext(1));
    ratio:= inint/100;  reqfreq:=inint/100;  reqfactor:=+1.0/reqfreq-1.0;
    cpurate:= inint/100;  simperiod:=inint;  ntasks:=inint;

    cpu:- new sked;

    altq:- new head;
    for i:=1 step 1 until 3 do priq[i]:- new head;

    tasklist:- new head;
    for tn:= 1 step 1 until ntasks do
       begin
          cycle:=inint;  resp:=inint;  pexec:=inint;
          aexec:=inint;  pexec:=pexec/cpurate;  aexec:=aexec/cpurate;
          imp:=inint;
          t:- new taskinit(imp,pexec,aexec,cycle,resp,tn);
          t.into(tasklist);
          activate t;
       end;

    hold(simperiod+fuzz);

    report;
end  *** simulation ***;
```

APPENDIX B: Scheduling Algorithms

The Last-Chance Scheduler

```
procedure intopriq;
   begin
      ref(altalg) q;          ref(prialg) p,p1;
      p:- priq[imp].first;
      while p=/= none do
         begin
            q:- p.assocalt;
            if assocalt.duetime<q.duetime then p:- p.suc
            else begin  p1:- p;  p:- none;  end
         end;
      if p1=/= none then precede(p1)
                    else into(priq[imp])
   end *** intopriq ***;

procedure intoaltq;
   begin
      ref(altalg) a,a1;           real t;
      a:- altq.first;
      while a=/= none do
        if a.duetime>duetime then begin  a1:- a;  a:- none;  end
         else a:- a.suc;
      if a1=/= none then
         begin
            precede(a1);
            if duetime>a1.skedstart then
               begin
                  skedstop:=a1.skedstart;
                  skedstart:=skedstop-exec;
               end;
         end
      else into(altq);
      t:= skedstart;
      a:- pred;
      while a =/= none do
         if t<a.skedstop then
            begin
               a.skedstop:=t;
               t:=a.skedstart:=a.skedstop-a.exec;
               a:- a.pred;
            end
         else a:- none;
      if t<time then error(' missed deadline ');
      reactivate cpu;
   end *** intoaltq ***;
```

```
procedure outaltq;
   begin
      real t;              ref(altalg) p,s,a;

      p:- pred;              s:- suc;
      out;
      if notaltq.empty then begin
         t:= if s=/= none then s.skedstart else p.duetime;
         a:- p;
         while a =/= none do
            if t>a.skedstop then
               begin
                  a.skedstop:= if a.duetime<t then a.duetime else t;
                  t:=a.skedstart:=a.skedstop-a.exec;
                  a:- a.pred;
               end
            else a:- none;
          end
   end *** outaltq ***;
```

```
comment this is the last-chance scheduler;
   process class sked;
      begin
         ref(altalg) a;          ref(prialg) p;
         real t,start;          integer priority;
         real array exectime[0:4];

   loop: if not  altq.empty then
            begin comment alternates (and primaries) are available;
               a:- altq.first;
               t:= a.skedstart-time;
               if t>0 then
                  begin comment use slack time (t) for primaries;
                     p:- if not priq[3].empty then priq[3].first
                         else if not priq[2].empty then priq[2].first
                         else priq[1].first;
                     t:= if t<p.timeleft then t else p.timeleft;
                     priority:=p.imp;  start:=time;

                     hold(t);

                     t:= time-start;  p.timeleft:=p.timeleft-t;
                     exectime[priority]:= exectime[priority]+t;
                     if p.timeleft<fuzz then activate p
                  end
               else
                  begin comment no slack time, must run alternate;
                     priority:=4;  start:=time;

                     hold(a.timeleft);

                     t:=time-start;  a.timeleft:=a.timeleft-t;
                     exectime[priority]:= exectime[priority]+t;
                     if a.timeleft<fuzz then activate a;
                  end
            end
         else
            begin comment no alternates (or primaries), cpu goes idle;
               priority:=0;  start:=time;

               passivate;

               t:=time-start;
               exectime[priority]:= exectime[priority]+t;
            end;
         go to loop;
      end  *** sked ***;
```

The First-Chance Scheduler

```
procedure intopriq;
   begin
      ref(altalg) q;          ref(prialg) p,pl;
      p:- priq[imp].first;
      while p=/= none do
         begin
            q:- p.assocalt;
            if assocalt.duetime<q.duetime then p:- p.suc
            else begin  pl:- p;  p:- none;  end
         end;
      if pl=/= none then precede(pl)
                    else into(priq[imp])
   end *** intopriq ***;

procedure intoaltq;
   begin
      ref(altalg) a,al;
      real t;
      a:- altq.first;
      while a=/= none do
        if a.duetime>duetime then begin  al:- a;  a:- none;  end
        else a:- a.suc;
      if al=/= none then precede(al)
         else into(altq);
      reactivate cpu;
   end *** intoaltq ***;

procedure outaltq;
   begin
      real t;
      ref(altalg) p,s,a;

      out;
   end *** outaltq ***;
```

```
comment this is the first-chance scheduler;
   process class sked;
      begin
         ref(altalg) a;          ref(prialg) p;
         real t,start;          integer priority;
         real array exectime[0:4];

  loop: if not altq.empty then
            begin comment alternates are available, run the first;
               a:-altq.first;  t:=a.timeleft;
               priority:=4;  start:=time;

               hold(t);

               t:=time-start;  a.timeleft:=a.timeleft-t;
               exectime[priority]:=exectime[priority]+t;
               if a.timeleft< fuzz then
                  begin a.out; activate a at a.duetime end;
            end

         else if not priq[3].empty then
            begin comment primaries are available, run the first;
               p:-priq[3].first;  t:=p.timeleft;
               priority:=3;  start:=time;

               hold(t);

               t:=time-start;  p.timeleft:=p.timeleft-t;
               exectime[priority]:=exectime[priority]+t;
               if p.timeleft<fuzz then activate p;
            end
         else
            begin comment no alternates or primaries, let cpu go idle;
               priority:=0;  start:=time;

               passivate;

               t:=time-start; exectime[priority]:=exectime[priority]+t;
            end;
        go to loop;
      end  *** sked ***;
```

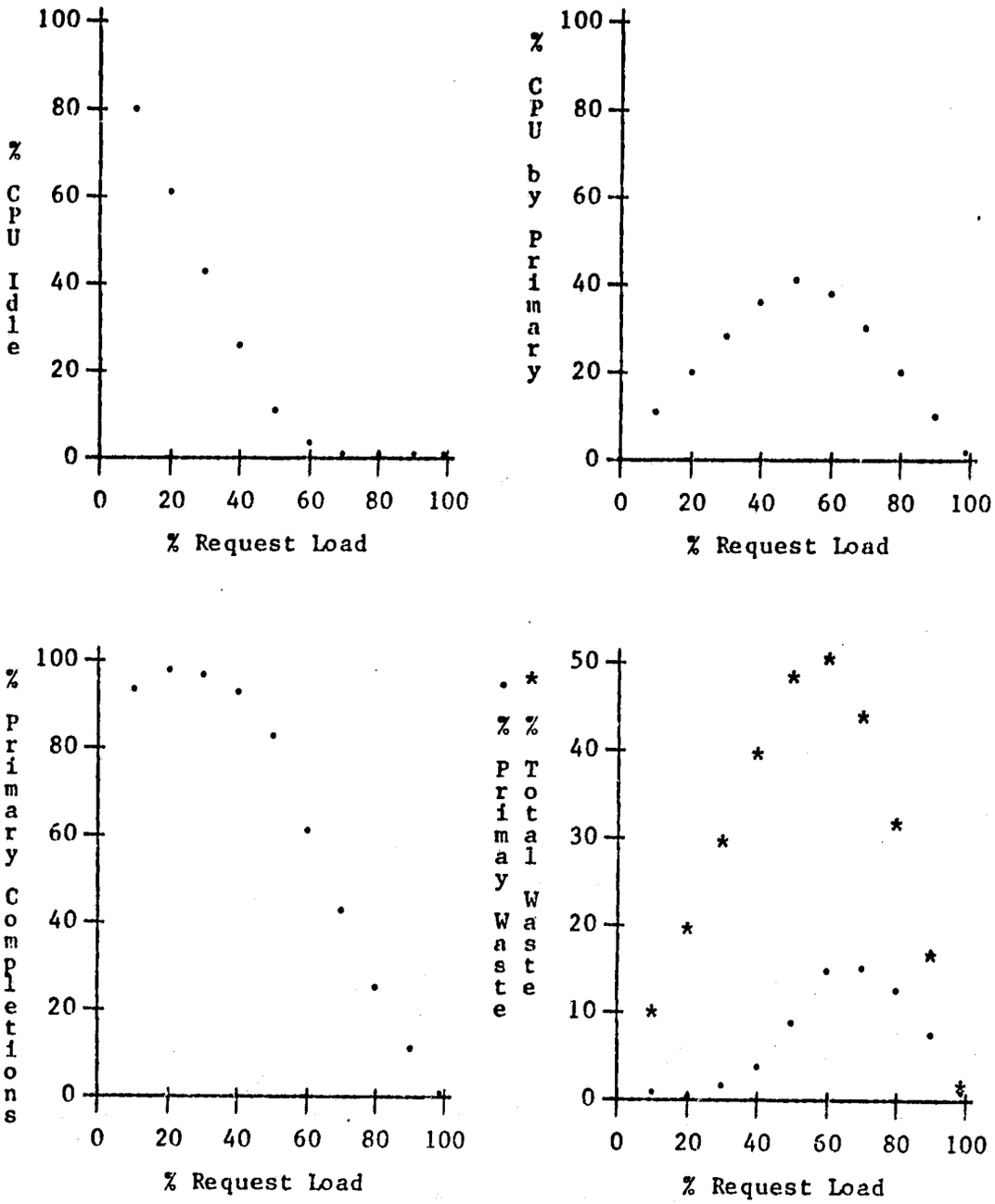
Rate-Monotonic Scheduler

```
procedure intopriq;
   begin
      ref(altalg) q;          ref(prialg) p,p1;
      p:- priq[imp].first;
      while p=/= none do
         begin
            q:- p.assocalt;
            if assocalt.resp<q.resp then p:- p.suc
            else begin  p1:- p;  p:- none;  end
         end;
      if p1=/= none then precede(p1)
                    else into(priq[imp])
   end *** intopriq ***;

procedure intoaltq;
   begin
      ref(altalg) a,a1;
      real t;

      a:- altq.first;
      while a=/= none do
        if a.resp>resp then begin  a1:- a;  a:- none;  end
        else a:- a.suc;
      if a1=/= none then precede(a1);
         else into(altq);
      reactivate cpu;
   end *** intoaltq ***;

procedure outaltq;
   begin
      real t;
      ref(altalg) p,s,a;

      out;
   end *** outaltq ***;
```

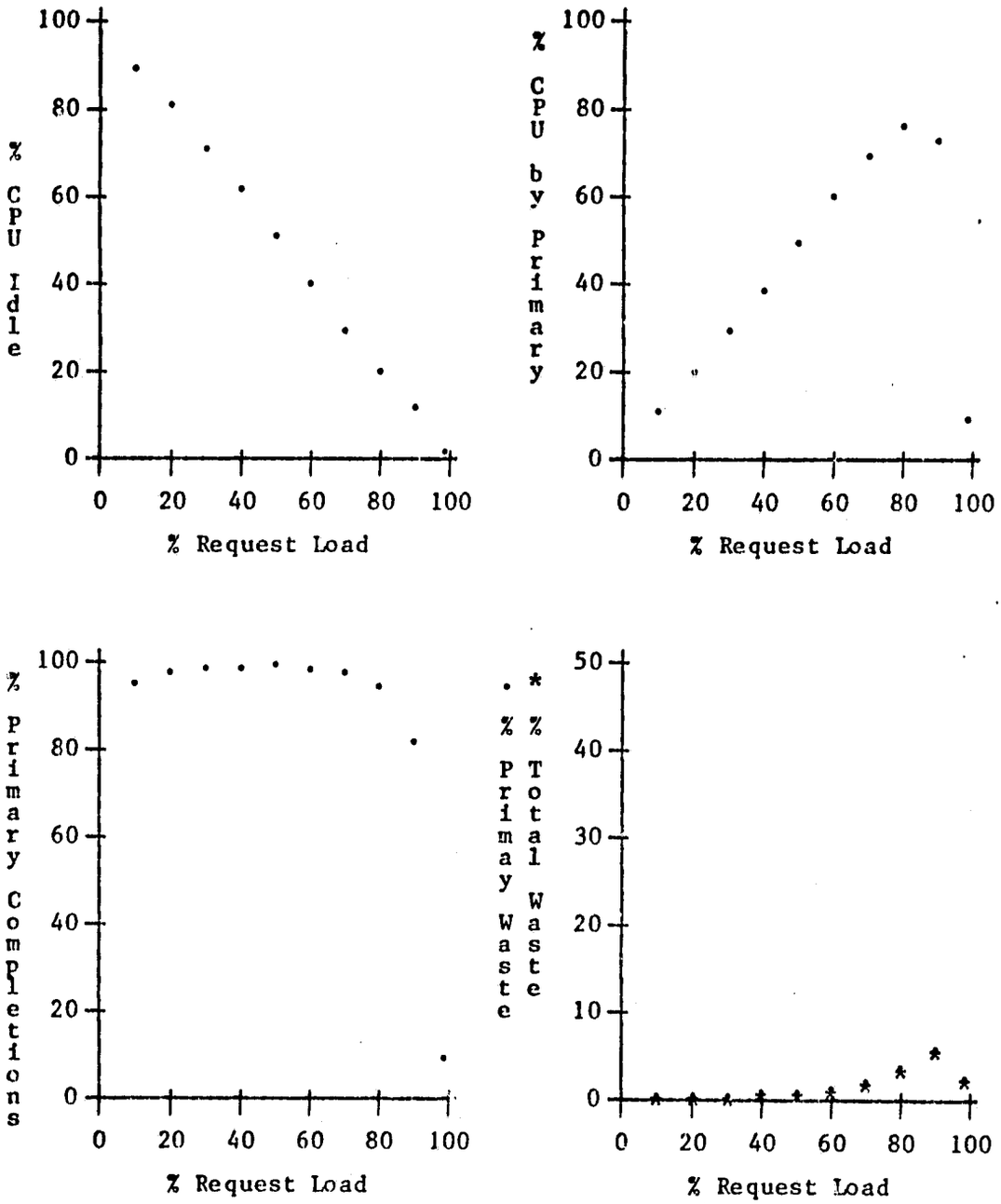
```
comment this is the rate-monotonic scheduler;
   process class sked;
      begin
         ref(altalg) a;          ref(prialg) p;
         real t,start;          integer priority;
         real array exectime[0:4];

  loop: if not altq.empty then
            begin comment select an alternate to run;
               a:-altq.first;  t:=a.timeleft;
               priority:=4;  start:=time;

               hold(t);

               t:=time-start;  a.timeleft:=a.timeleft-t;
               exectime[priority]:=exectime[priority]+t;
               if a.timeleft< fuzz then
                  begin a.out; activate a at a.duetime end;
            end

         else if not priq[3].empty then
            begin comment no alternates, so find a primary;
               p:-priq[3].first;  t:=p.timeleft;
               priority:=3;  start:=time;

               hold(t);

               t:=time-start;  p.timeleft:=p.timeleft-t;
               exectime[priority]:=exectime[priority]+t;
               if p.timeleft<fuzz then activate p;
            end
         else
            begin comment no primary or alternates, let cpu go idle;
               priority:=0;
               start:=time;

               passivate;

               t:=time-start;
               exectime[priority]:=exectime[priority]+t;
            end;
          go to loop;
      end  *** sked ***;
```

APPENDIX C: Simulation Data Plots

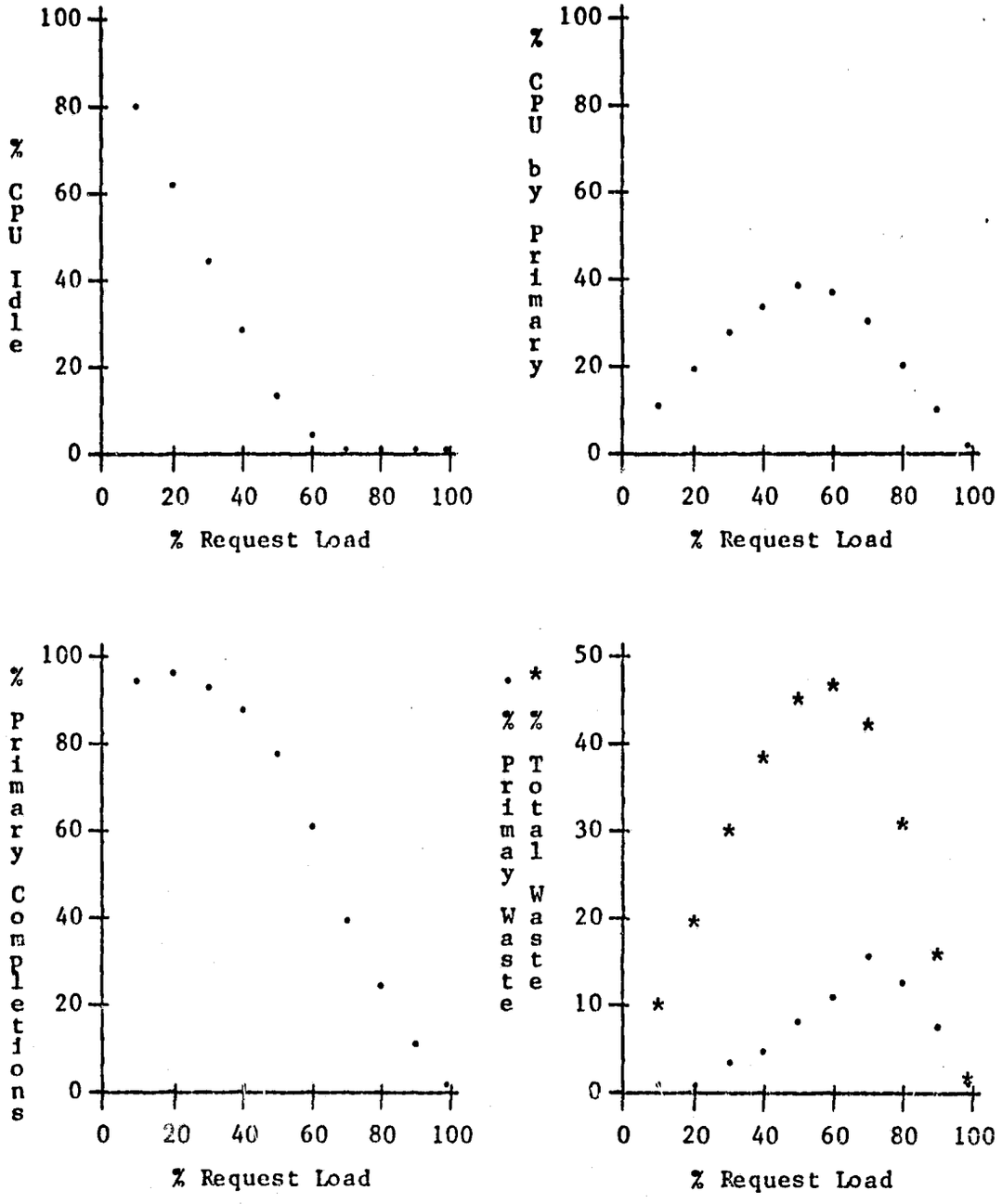
This appendix contains several example data plots from the simulations. Four graphs are presented for each set of runs. The first of these plots percentage of 'cpu' idle time versus request load. The second plots percentage of 'cpu' time consumed by primary algorithms (ignoring those abandoned primaries). Third we plot the percentage of deadlines met by primary algorithms. The last graph plots time wasted by abandoned primaries and total time wasted (this includes both abandoned primaries and alternate executions whose results were never used). A brief description of the mix of tasks and the scheduler used appears at the bottom of each page.



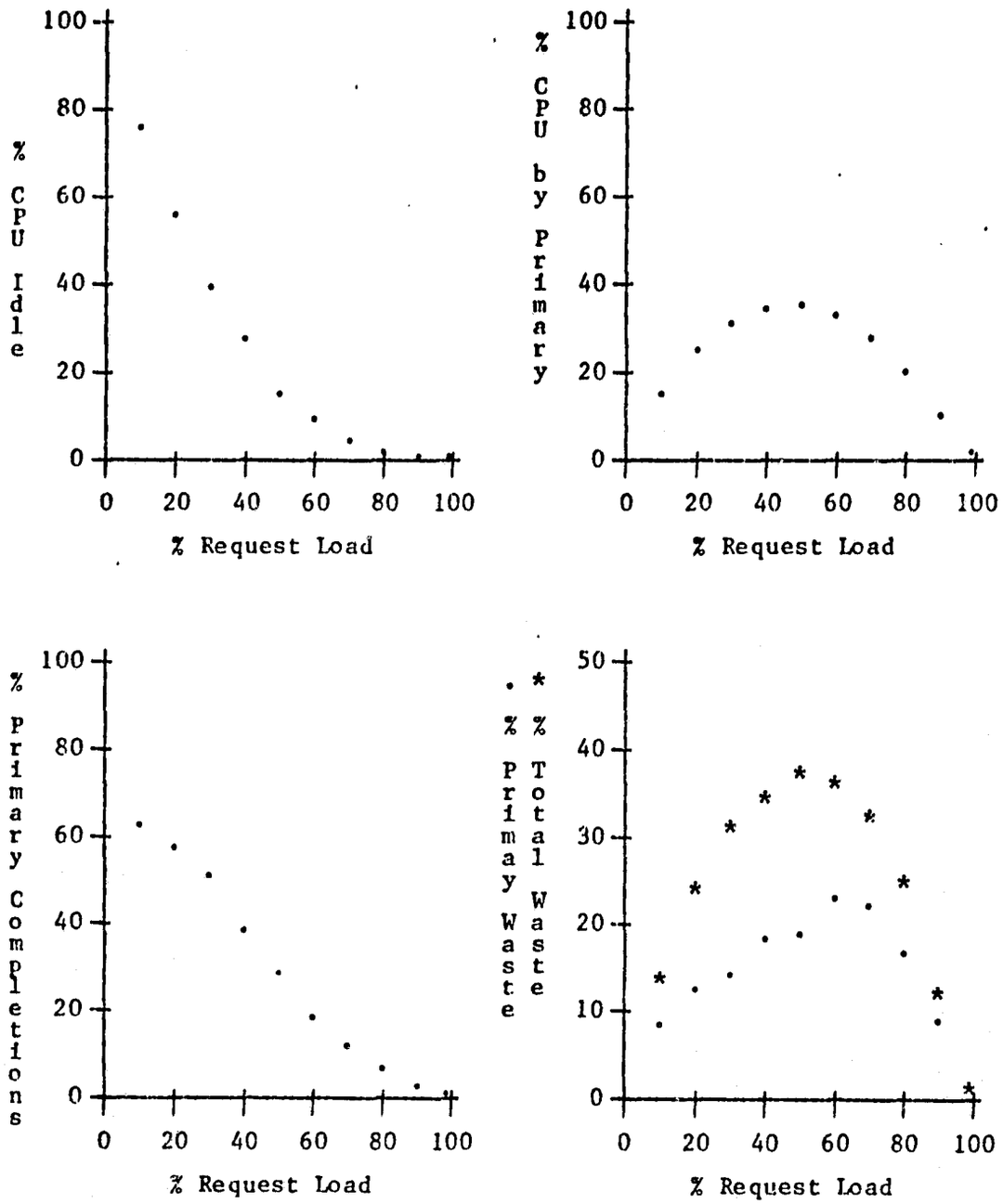
This simulation consisted of 10 identical tasks with response and request periods of 100; primary and alternate service periods were 10. The first-chance scheduler was used.



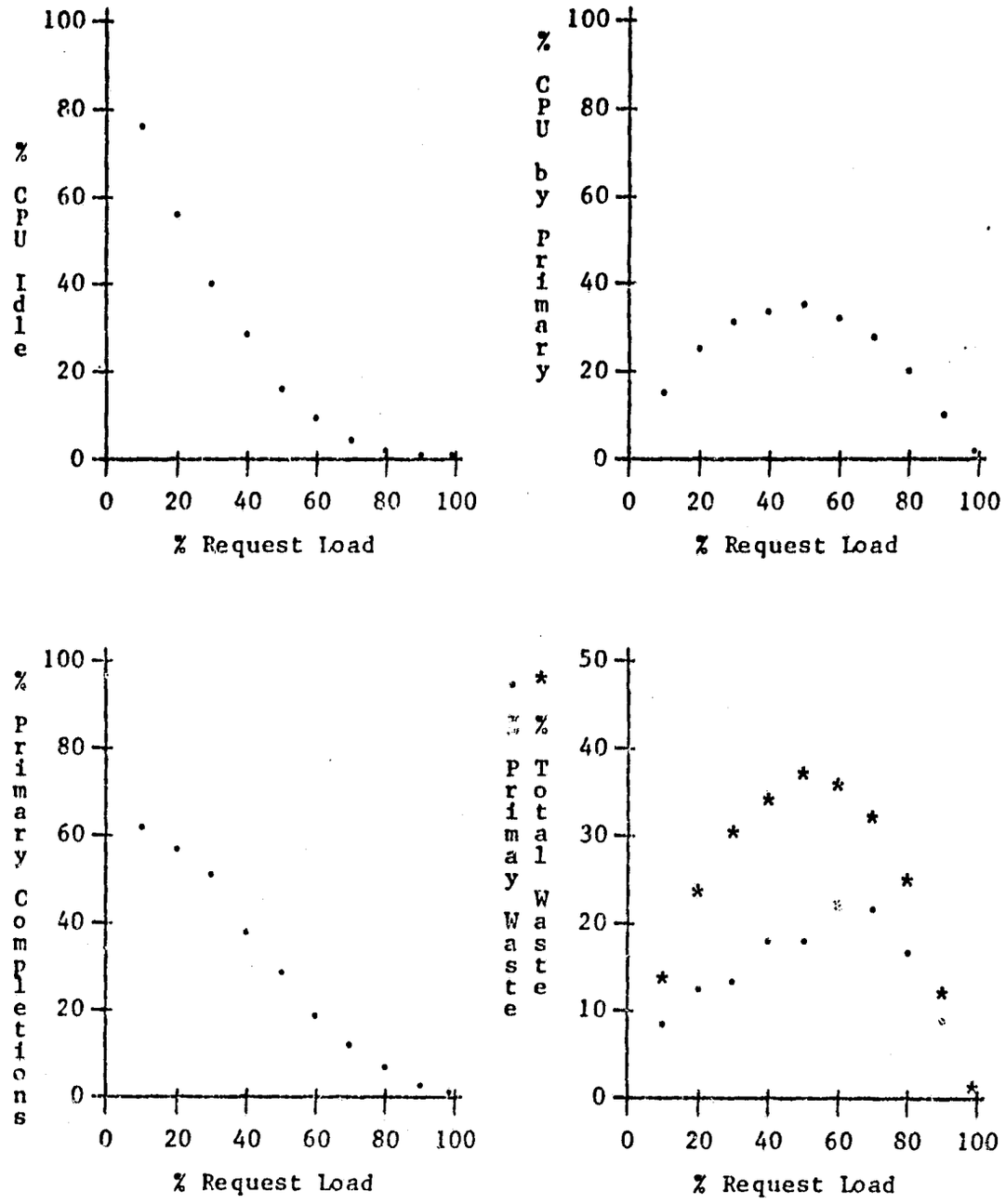
This simulation consisted of 10 identical tasks with response and request periods of 100; primary and alternate service periods were 10. The last-chance scheduler was used.



This simulation had the following task mix: 1 task with response and request periods of 50, primary and alternate service periods of 10; 8 tasks with response and request periods of 100, primary and alternate service periods of 10. The first-chance scheduler was used. Condition 2 is not satisfied by this mix, and consequently the last-chance scheduler could not be utilized.



This simulation had a task mix of 4 tasks with the ratio of alternate service to response equal to .25. The primary service period was equal to the alternate service period. The first-chance scheduler was used.



This simulation had a task mix of 4 tasks with the ratio of alternate service to response equal to .25. The primary service period was equal to the alternate service period. The rate-monotonic scheduler was used. Deadlines were missed at request loads above 90%.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-79-998	2.	3. Recipient's Accession No.
4. Title and Subtitle A FAULT TOLERANT DEADLINE MECHANISM			5. Report Date December 1979 6.
7. Author(s) K. Horton			8. Performing Organization Rept. No. UIUCDCS-R-79-998
9. Performing Organization Name and Address Department of Computer Science University of Illinois U-C Urbana, IL 61801			10. Project/Task/Work Unit No. 11. Contract/Grant No.
12. Sponsoring Organization Name and Address			13. Type of Report & Period Covered 14.

15. Supplementary Notes

16. Abstracts

The deadline mechanism extends Randell's recovery block concept to include real-time systems. Its application can aid the development of fault-tolerant real-time systems. The report develops the deadline mechanism from the recovery block concept, proposes several scheduling strategies, and discusses the results of simulating the mechanism using each strategy.

17. Key Words and Document Analysis. 17a. Descriptors

real-time systems
fault-tolerant systems
fault-tolerant real-time systems
deadline scheduling
recovery blocks

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
unlimited	UNCLASSIFIED	52
	20. Security Class (This Page) UNCLASSIFIED	22. Price