CMU-CS-84-130

Validation of Fault-Free Behavior of a Reliable Multiprocessor System FTMP: A Case Study

Ed Clune, Zary Segall, Daniel Siewiorek
Carnegie-Mellon University
Pittsburgh, PA

This research was sponsored by the National Aeronautics and Space Administration, Langley Research Center under grant NAG-1-190. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NASA, the United States Government or Carnegie-Mellon University.

Table of Contents

List of Figures

Abstract

This report outlines a methodology for validating a reliable multiprocessor system. This validation methodology is being used, in part, on the Fault Tolerant Mutli-Processor (FTMP) at NASA/Langley's AIRLAB facility. Specific experiments run on FTMP are discussed. These experiments were part of a methodology designed to test and verify the *fault-free* performance of the system at many levels, from hardware through system software.

This report also discusses the analysis of several baseline experiments, that is, experiments that give basic information about the system. Results of the baseline experiments are a prerequisite for measuring FTMP performance in more complex experiments. From the baseline experiments it was determined that the clock was constant and independent of workload in the tested cases; that instruction execution times are virtually constant; that the R4 frame size is 40mS with some variation; and that the frame stretching mechanism has some flaws in its implementation that allow the possibility of an infinite stretching of frame duration.

Future measurements are planned. Some will broaden the results of these initial experiments. Others will measure the system more dynamically. The implementation of a synthetic workload generation mechanism for FTMP is planned to enhance the experimental environment of the system.

1. Introduction

An aircraft of the 1990's will have computer systems that must function correctly for the aircraft to fly. Many studies have been performed on fault tolerant avionics computers. One such study by NASA, in its Aircraft Energy Efficiency (ACEE) program [9], requires that an aircraft computer failure probability should be less than 10^{-10} per hour. Systems have been built with this goal in mind (SIFT and FTMP [9, 4, 3]). Techniques must be developed for measuring the performance and reliability of these systems.

Carnegie-Mellon University has been involved in developing methods of validating and measuring multi-processors. In addition, NASA has held several workshops to determine system validation procedures. One workshop in particular [6] produced a detailed list of validation tasks to verify a system in an orderly manner.

This theory is being tested and extended on the Fault Tolerant Multi-Processor (FTMP). These experiments were performed at NASA-Langley Research Center. The computer systems are in a laboratory called the Avionics Integrated Research Laboratory (AIRLAB). AIRLAB is a facility for developing technologies and methodologies to evaluate and integrate avionics and control functions of future aircraft and to establish a store of performance evaluation and reliability evaluation statistics.

Some background is required to understand the experiments that were performed and how they relate to the overall goals of this research. Section 2 explains FTMP and discusses the theoretical environment and methodology for fault-free validation of FTMP. This section also describes the present state of the environment and validation process. Section 3 explains the experiments to measure some of the processor and operating system characteristics of FTMP. In Section 4, the experimental results are presented and conclusions drawn. A summary and discussion of future work is presented in Section 5.

2. Background

This section first describes the Fault Tolerant Multi-Processor (FTMP). Next, it discusses the theoretical basis for the experiment environment and the methodology for evaluating fault tolerant multi-processors. Then, the present working environment and state of FTMP validation is explained.

2.1 The Fault Tolerant Multi-Processor (FTMP)

The Fault Tolerant Multi-Processor (FTMP) has been discussed in several papers and manuals [3, 4]. This section will only describe those details necessary for understanding the experimental results. For more details the interested reader is referred to the literature. Figure 2-1 depicts FTMP at the software level (as seen by the application programmers). There are up to three triads, each with local memory. A triad consists of three processors that the programmer sees as a single processor. A bus connects the triads to global or main memory, I/O devices, a real-time clock and several latches needed for fault handling. The triads only execute independently when accessing local memory.

Work is performed by tasks. A task is a process that can be started independent of other tasks. Each triad will run tasks according to a schedule. Each task is assigned a time limit. If a task cannot be completed within the time limit, the task is stopped and the next task started.

Tasks are run within frames. Frames also act as a synchronization mechanism between triads. One of the triads becomes the leader and starts a frame for that triad and signals all of the other triads to start the frame. In the time allotted by the frame, the group of working triads must execute all of the tasks they are assigned. The tasks are in a global linked list with each pointing to the next task (except the last which has a null pointer). The individual triads access the global list to pick up a task. If there is more than one triad, some tasks will be executed in parallel. When there are no more tasks available for a triad to execute the triad becomes idle until the end of the frame. At that time, a triad becomes leader and starts a new frame.

In FTMP there are actually three frame sizes, each having a different frequency of execution as seen in Figure 2-2. Each triad has separate pointers to tasks for each rate group. The frame sizes are:

- R4, the basic frame size
- R3, equivalent to 2 R4 frames
- R1, equivalent to 4 R3 frames, the 'major' frame

Task execution becomes more complicated with multiple frame sizes. One triad still signals the start of the R4 frame, however, every second R4 frame it also starts an R3 frame and every eight R4 frames

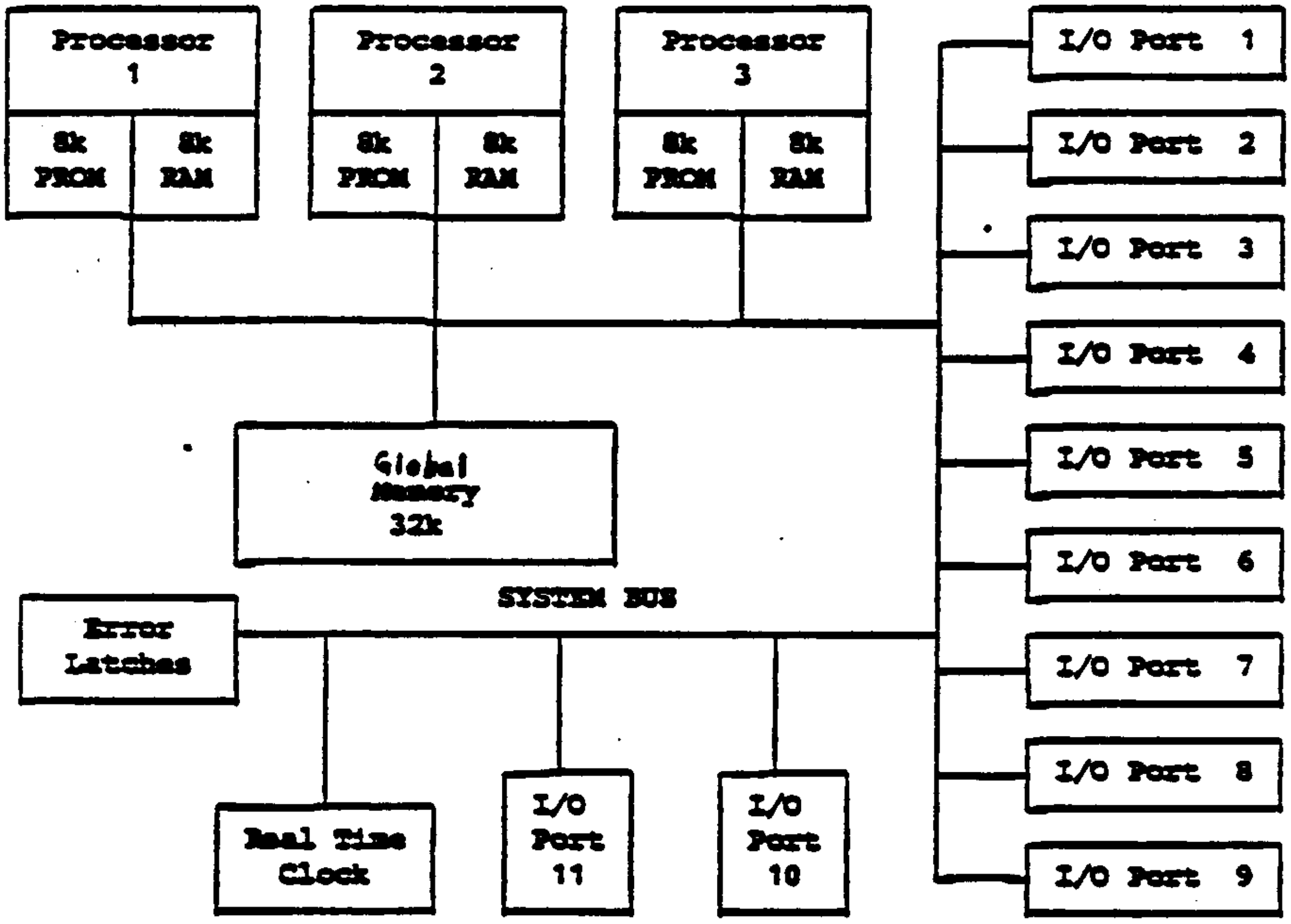

Figure 2-1: FTMP System

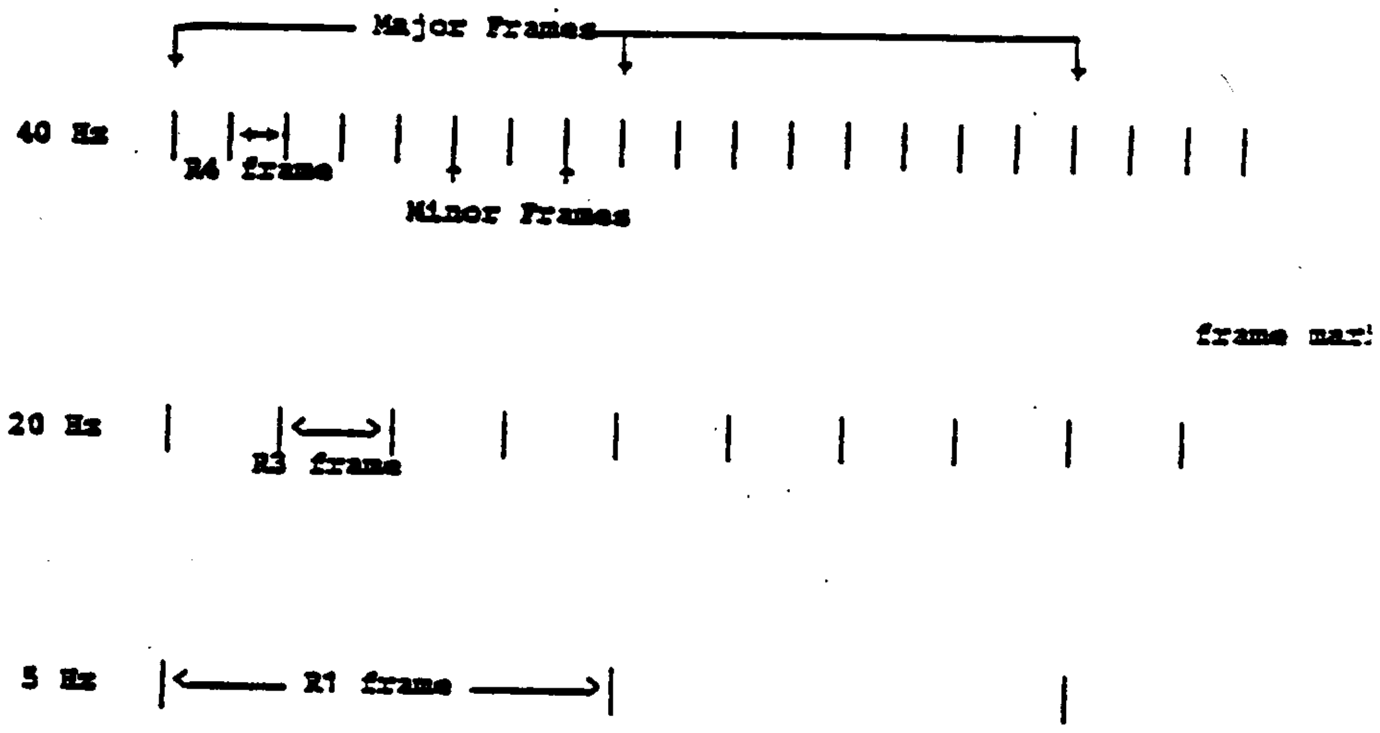

Figure 2-2: Frame Structure

it starts an R1 frame. The order in which a triad executes tasks for the different frame groups is fairly simple. First, it executes all of the R4 tasks, then (still in the same R4 frame) it executes R3 tasks. If the R3 tasks do not finish before the next R4 frame, execution of the R3 task is suspended and another R4 frame is started. Again, when all of the R4 tasks are done, the R3 tasks are continued. If the R3 tasks are finished, the R1 tasks are started. If these tasks are not finished before the beginning of the next R4 frame, they are suspended and started after the R4 tasks are done in the next frame. If another R3 frame starts before the R1 tasks finish, the current R1 task is suspended in the triad until time is available in a frame.

There is another interesting item concerning frames. According to the documentation, if a task needs more time in a frame, the frame can be stretched as illustrated in Figure 2-3. An R4 frame is stretched by a specific amount and R3 and R1 frames are stretched by giving them more R4 frames. The third experiment uncovered some interesting properties of this stretching mechanism.

Time is kept using a global clock. The clock has a resolution of .25mS (that is, each clock tick is .25mS). The clock and I/O devices are accessed by using a function called HREAD. A complimentary function is HWRITE. HREAD allows a program to transfer bytes between a device and the local memory of a triad. Transfers between local and global memory occur by invoking the

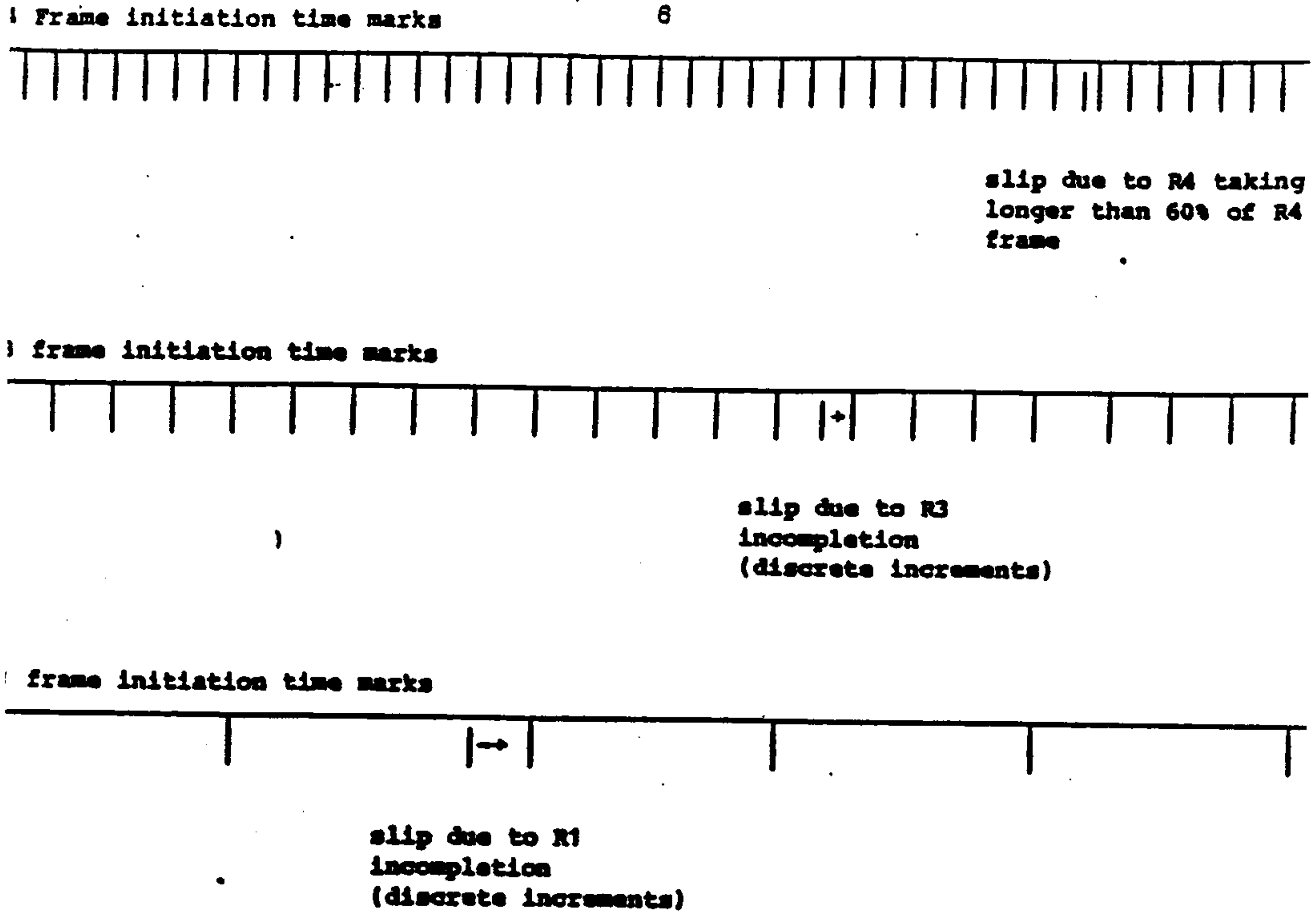
Frame initiation time marks
6
slip due to R4 taking longer than 60% of R4 frame
frame initiation time marks
slip due to R3 incompletion (discrete increments)
frame initiation time marks
slip due to R1 incompletion (discrete increments)

Figure 2-3: Frame Stretch Mechanisms

functions RD and WRT. Knowing the amount of time that these transfers take and how the time can vary are crucial to understanding the limits of system performance.

Several computing systems are involved in running the experiments. Programs for FTMP are written in a language called AED and assembly language. The compiler, assembler and linker for these languages reside on an IBM 4331. Object files are transferred to a VAX/750. Special interface programs on the VAX are used to load, read and write global memory locations in FTMP. Also, a batch facility on the VAX allows experiments to be run unattended. An HP terminal displays the status and other features of FTMP while it is running. Recently, it became possible to remotely access FTMP through the VAX so that experimenters do not have to be present at AIRLAB to conduct experiments.

2.2 Experiment Environment

This section will briefly outline the environment proposed for the fault-free validation of FTMP. This environment has been developed for the performance evaluation of Cm* at CMU.

Multiprocessor systems are enormously complex. In order to make them easier to comprehend, it is necessary to divide the system into several levels. One can then proceed from the most primitive level upwards to the highest conceptual level by introducing a series of abstractions. Each abstraction contains only information important to its particular level, and suppresses unnecessary information about lower levels. The levels in a digital system frequently coincide with the system's physical boundaries since the concept of levels was utilized by the system's designers to manage complexity. Once details at one level are comprehended, only the functionality provided for the next higher level need be considered. Figure 2-3 depicts one possible set of levels of abstractions.

Level	Sublevel	Typical Components
Multiprocessor		Processor, memory, switches
Program	Application Software	Display, navigation, flight control
	Executive Software	Message system, task scheduler, memory allocator
	Instruction Set	Memory state, processor state, effective address calculation, instruction execution
Hardware	Logic	Gates, flip-flops, registers, sequential machines

Figure 2-4: Levels of Abstraction in Multiprocessor Systems

Our experience at CMU indicates multiprocessors go through a series of stages. A stage is defined by the amount of functionality available to the user. This functionality, in turn, determines the complexity and sophistication of experiments that can be conducted.

There are several activities in the life of an experiment. First, the code has to be designed and written. Next, it must be compiled, followed by loading, debugging, measurement, and analysis.

Another view of the stages of a system's life is the number of these activities that are directly supported by the system for the user.

The following are three representative stages in the evolution of a typical multiprocessor system.

2.2.1 Stage 1 - Standalone

The system is completed through the instruction set level of abstraction. That is, the instruction set has been defined and the hardware has been implemented. There is virtually no software to support user applications. The only software utility would be a loader whereby programs compiled on another machine can be loaded into the system under test. Experiments are limited to simple, regular, compute bound algorithms. Only a limited number of parameters may be varied, and this variation requires rewriting of the source code of the experiment. There are several attributes to Stage 1 experiments. The programmer must be a hardware expert since there is little software to provide a higher level virtual (abstract) machine. Hence the program is tied closely to the hardware. The user must specify where code is placed, define the memory map, and write code to initialize the memory, create processes, manage resources, and collect data.

Typical experiments in Stage 1 include:

- **Hardware Saturation.** Programs consist of two or three instruction loops with variation in placement of code and data. The capacity of various system hardware resources is determined as well as the impact of contention for those resources.
- **Speedup due to Algorithm/Data Variation.** Experiments seek the impact of synchronization for data, as well as variation due to data dependencies and size of data.
- **Errors.** Diagnostic programs can be continuously run and monitored on the system. Distribution of diagnostic detected errors can be studied.

2.2.2 Stage 2 - Operating System (OS)

The user is presented the abstraction provided by the executive software. This software provides basic functions such as resource management and scheduling. In programming experiments, the user employs operating system primitives. Hence, the user needs a substantial operating system expertise. Also, characteristic for this phase is the discrete incremental nature of the experimentation process; each experiment represents one point in the design space.

The attributes of Stage 2 applications can be stated as follows:

- very regular, data bound with limited variation of parameters
- the general program organization has a Master process controlling a collection of Slave processes doing the actual computation
- code is replicated

- heavy use of OS mechanisms

Typical experiments are:

- **Measurements of the cost-per-feature of the operating system's functions.** Experiments exercise statically each OS function on a one by one basis. Examples include: memory management, communication primitives, synchronization, scheduling and exception handling.
- **Measurements of different implementation of parallel algorithms.** The impact of using various strategies in parallel program organization, data structure and resource allocation is studied.

2.2.3 Stage 3 - Integrated Instrumentation Environment

At this stage hardware and software have been provided for generating experimental stimulus, dynamically observing hardware and software activities, and analyzing results [8]. With this enhanced support, the user can experiment at the application level of abstraction with full variation of parameters. A major characteristic of this stage is the provision of stimulus generation, monitoring, data collection and analysis grouped under a unique user interface. Also the OS, the support software and the user application are uniformly instrumented enabling improved behavior visibility. Only with this capability, the interaction between OS, support software and user application became measurable with acceptable effort. Hence, the programmer could be a relative system novice. Experiments at this stage have the following attributes:

- Measurements of dynamic behavior of OS and applications.
- Measurements are continuous. Program could be monitored on-line and sometimes in real-time.
- Studies of different virtual machines.
- Studies of different logical intercommunication structures.
- Scaling application performance with respect to different virtual machines.

Examples of experiments at this stage include:

- Comparison of various OS policies as reflected by classes of applications.
- Tuning a virtual machine for a specific application.
- Designing application oriented architectures.
- Study of multiprocessor intercommunication strategies.
- Validation of fault-free performance of an emulated system.
- Study of the architectural effectiveness and efficiency.
- The handling of faults represents additional load for the avionics system. The fault capabilities represent another aspect of system functionality. Whereas a system without faults may be able to meet all of its deadlines, the addition of fault handling workload may cause schedule slippage and/or violations of realtime constraints.

A key part of the Stage 3 methodology is the specification and generation of a controlled parallel workload [8]. Such a workload for avionics applications is given in [1]. The workload is represented as a special purpose parallel data-flow graph. A run-time experimentation environment provides capability of controlling, varying and measuring the workload without having to recompile or re-debug the parallel program.

2.3 Proposed Methodology

NASA held several workshops to determine system validation procedures. One in particular [6] produced a detailed list of validation tasks to verify a system in an orderly manner. The list was as follows:

1. Initial Checkout and Diagnostics
2. Programmer's Manual Verification
3. Executive Routine Verification
4. Multiprocessor Interconnect Verification
5. Multiprocessor Executive Routine Verification
6. Application Program Verification and Performance Baseline
7. Simulation of Inaccessible Physical Failures
8. Single Processor Fault Insertion
9. Multiprocessor Fault Insertion
10. Single Processor Executive Failure Response Characterization
11. Multiprocessor System Executive Fault Handling Capabilities
12. Application Program Verification on Multiprocessor
13. Multiple Application Program Verification on Multiprocessor

The first six tasks verify functionality and the next seven verify fault handling performance. Several of these tasks are in process or have been completed. This paper specifically deals with a set of performance baseline experiments (Task 6) for FTMP.

2.4 Present and Future Experiment Environment

A signifcant amount of work was required by AIRLAB personnel to bring the system environment up to Stage 2. At present, each experiment generally requires some code compilation, followed by linkage and downloading of the whole FTMP binary file. Experiments can be designed with modifiable variables so that some variation can be made without having to go through the entire code development cycle. The experiments described in this paper used the modifiable variable approach.

A more specialized workload generation mechanism is being developed for use on real-time multiprocessor systems (FTMP in particular [1]). With this mechanism in place, experiments can be run in an environment somewhere between Stage 2 and Stage 3. This model considers tasks of a specific organization and deals with a simple set of parameters. The system is assumed to be made up of a bus with several processors (each with local memory), one global memory, and I/O. Operating system tasks are considered part of the system under measurement. Traffic on the bus is stricted to I/O and Inter-Process Communication (IPC), each of which access memory.

In this real-time model, tasks are made up of five sections. These sections include read in I/O data, read in IPC data, perform some representative operation, write out I/O data and write out IPC data. The amount of work performed in each section can be varied by parameters.

The workload structure was designed for simplicity so that variations in the workload parameters and the resulting measurements could be easily understood. The system parameters consist of total I/O, total IPC, and total instruction execution per second. Each system parameter is divided between functions as a percentage of the total work each function performs. Each function is in turn made up of tasks which divide the work of the function as evenly as possible. Measurement of the throughput, system utilization and interaction of the system is done by using the system clock to measure when a task begins and ends.

2.5 Present State of Validation Process

At present the individual processors and operating system are assumed valid. The baseline experiments are meant to.determine static properties such as the time to execute an operating system call or the amount of time left in a frame upon completion of an avionics function. Several areas of baseline experiments are possible:

- Instruction Level
  - o Assembly and High-level language instruction times.
- Operating System
  - o OS primitive and overhead times
  - o Interrupt procedure times
  - o Memory access time
  - o Bus access and contention delays
  - o Fault tolerance overheads
- System and Application
  - o Frame utilization characteristics (including OS overhead and bus contention delay and fault tolerance overhead)

The specific experiments that are reported upon in this paper are:

- Clock Read Delay. In order for subsequent experiments to be valid, the delay and variation in reading the clock must be determined.
- Processor Performance for Simple Operations. This is a measure of the amount of time required by the processor to perform simple AED instructions, for example 'A = 1' or 'A = B + C'.
- R4 Frame Iteration Rate. The measurement of the R4 frame under nominal conditions as well as when stressed by long tasks.

3. The Experiments

Most of the experiments were organized as in Figure 3-1.

```
Begin
  EXEC = Read(CMU.EXEC);
  If CMU.EXEC <= SomeCount Then
    begin
      RTCNUM = Read(CMU.RTCNUM);
      Hold = Read(RT.CLOCK);
      For X=1 to RTCNUM do
        begin
          SomeInstructions;
        end;
      Hold1 = Read(RT.CLOCK);
      Write(Hold,CMU.TIME(1));
      Write(Hold1,CMU.TIME(2));
      EXEC = EXEC + 1;
      Write(EXEC,CMU.EXEC);
    end;
End.
```

Figure 3-1: Basic Experiment Task Algorithm

When a task starts, a global variable called CMU.EXEC is read from global memory. If it is above a certain value (which depends on the experiment) the task is terminated. If it is not, a second variable, CMU.RTCNUM, is read from global memory. CMU.RTCNUM is the number of iterations that a loop must execute. In most cases, the global time is read, an instruction is repeated a number of times (defined by CMU.RTCNUM) and time is read again. These numbers are then stored in the global array CMU.TIME. Finally, CMU.EXEC is incremented.

The time limit for each experiment task must be large enough that the task can finish. Also, some of the experiment tasks must finish before the 60% mark of the R4 frame has been reached. According to the documentation, after that time an interrupt will occur. The interrupt would invalidate any time measurements.

Using the VAX/FTMP interface program called CTA and a batch command script, the values in the FTMP global memory can be read and stored in a file on the VAX. CTA can also set FTMP memory locations. Therefore, a command script can set CMU.EXEC to 0, wait, read the global array and repeat as many times as desired.

It is important to note, that the experiments allow the number of iterations to be changed using CTA. For example, if the number of iterations is found to be too small to obtain useful results, CMU.RTCNUM can be increased using CTA and the experiment can be run again. Changes can be made without having to recompile, relink and reload FTMP. This saves a great deal of time.

3.1 Clock Read Time Delay

In order for any subsequent experimental results to be considered valid, the characteristics of the clock must be determined. The delay and variation in reading the clock must be determined, as well as the causes of any variations. If these variations cannot be characterized or minimized, any further experiments using the clock would be suspect. For example in Cm* multiprocessor system, there was as much as 4.6% difference in clock frequency, and substantial variation of clock read delays [5]. An example of possible characteristics of clock read delay would be a constant offset that could be subtracted from any future experiment results using the clock.

On FTMP the time is read with the instruction 'HREAD(RT.CLOCK,variable,2);'. In the experiment task, 16 iterations, each of 5 clock reads were made with the time before starting and the time of the last read being stored in global memory. A second task was created that did exactly the same as the first task except that it did not write to global memory. This second task was placed so that it would start execution immediately after the measurement task. If two triads were in use, the second task would execute in parallel with the first and add contention for the clock.

The experiment was repeated about 100 times for three situations

1. Triad 1 running alone,
2. Triad 2 running alone,
3. Triad 1 and 2 running simultaneously (contention for the clock),

The first two runs determined single triad clock read time with no contention, and variation between triads. The third case determined how the clock contention effects the clock read time.

3.2 Instruction Times

The times for the following AED instructions were measured:

1. 'Null'	
2. A = 1;	(integer assign)
3. A1 = 1;	(real assign)
4. A2 = 1;	(long assign)
5. A = B + C;	(integer add)
6. A1 = B1 + C1;	(real add)
7. A2 = B2 + C2;	(long add)
8. A = B*C;	(integer multiply)

Each of these instructions was executed in a loop 100 times along with the instruction 'A = 1'. The 'A = 1' instruction was added because the compiler would not accept a null statement for the first instruction. The 'Null' statement was included so that the overhead from clock reading and loop control can be eliminated from the other instructions, leaving only the time for instruction execution. This task was executed 308 times.

3.3 Measuring R4 Frame Size

There were three parts to this experiment. The first part was to measure the nominal R4 size. Upon starting, the first R4 task reads the real-time clock. If the global variable CMU.EXEC is set correctly, this time will be stored. This will be done eight consecutive times, giving eight relative R4 frame sizes. The experiment was run about 100 times each for one triad and two triads.

The second part of the experiment was to determine how the system behaved when an R4 frame was stretched. Only one triad was used. The time limit for the task was set to a very large number (so that a task would not abort before it was finished). Finally, RTC.NUM had to be set to several values that would stretch the frame. These values were 2000, 3000 and 5000. Data was recorded about 100 times for each iteration value.

The last part of the experiment was to determine the effect on the system of an R4 frame with an infinite number of tasks. One of the words of control information associated with a task was a pointer to the next task. An infinite string of tasks could be generated by having a task point to itself as the next task. One R4 task was caused to execute over and over in this way. Another task was checked to see if it ran once the R4 task started repeating.

4. Results

4.1 Read Time Clock Delay

The clock read overhead was virtually constant for a single triad configuration. The data never varied more than a clock tick (.25 mSec). For the two different triads the results were:

```
For Triad 1 -- 14.0 ± .012 mSec / 16 iterations[1]
               .874 ± .00073 mSec / iteration

For Triad 2 -- 14.0 ± .0091 mSec / 16 iterations
               .875 ± .00057 mSec / iteration
```

Each iteration has 5 clock reads plus loop overhead. Loop overhead per iteration is 15.7μSeconds (see Experiment 2). This is subtracted from the iteration time, then the result is divided by 5.

```
Triad 1          874 ± .73 μSec
                 -15.7±.11
                 ------
                 858.3 ± .84 / 5 = 172 ± .17 μSec

Triad 2          875 ± .57 μSec
                 -15.7±.11
                 ------
                 859.3 ± .68 / 5 = 172 ± .14 μSec
```

A read with no contention on the bus requires 172μSeconds. Although there is an indication of some variation between triads, it is not significant and within the margin of error for a 95% confidence interval.

In the second measurement, two triads were started, each executing roughly the same code so that contention for the bus is created.

For 2 Triads --
14.1 − + .023 mSec for 16 iterations,
173 − + .31μSeconds per read.

It is evident that the contention for the clock at this rate does not affect the delay in reading the clock greatly (less than 1%). The Contention is large enough that the range of the 95% confidence intervals for the single triad read time and double triad read time do not overlap. These results do not take into account other contention for the bus like memory access or I/O device access.

The reason that this variation is so small is that the section of code in the read procedure that

[1] All intervals are 95% confidence intervals. Refer to Ferrari [2] for a description of confidence intervals and how they are calculated.

actually uses the bus is a small percentage of the whole clock read procedure. Since both contending procedures are exactly the same when in the iteration section, they will tend to be synchronized so that only one will actually request control of the bus at a time. The slight variation from the single triad case could be due to slight variations in the execution rates of the different processors so that occasionally the two triads do conflict. However, this would seem to be very minor.

On the whole, the real-time clock on FTMP should serve as a reliable measurement device with predictable delays that can be factored out of experiments. This is especially true in the single triad case. However, this assumes that the experimenter has complete control of all of the tasks. If an experimenter on the system with multiple triads, lets one triad run uncontrolled, the clock results may not be reliable. The range of system activities under which the clock times are repeatable should be explored further.

4.2 Instruction Measurement

The result of the instruction measurements were as follows:

AED Instruction	clock ticks per 100 instr(ave.)	µS per instruction		
		w/Null task overhead	w/o overhead	predicted by Rockwell
1) Null	12.3±.013	30.7 ±.013	---	---
2) A=1;(Int Assn)	18.3±.013	45.7 ±.013	15.0 ± .026	8.3
3) A1=1;(Real Assn)	18.4±.013	46.1 ±.014	15.4 ± .027	8.3
4) A2=1;(Long Assn)	19.6±.013	49.1 ±.014	18.4 ± .027	12.3
5) A=B+C;(Int Add)	23.0±.013	57.7 ±.004	27.0 ± .017	22.3
6) A1=B1+C1;(Real Add)	23.2±.013	58.0 ±.011	27.3 ± .024	22.3
7) A2=B2+C2;(Long Add)	27.4±.013	68.6 ±.014	37.9 ± .027	30.0
8) A=B*C;(Int Mult)	25.1±.013	62.9 ±.010	32.2 ± .023	27.4

Overhead in this case means loop and clock read overhead, plus the time to execute 'A = 1'. The range is for a 95% confidence interval.

The results showed little variance. The time per frame varied only by .25mSecond (one clock tick) for each AED instruction. The instructions took longer than suggested by the times given by the assembler and Draper Labs documents [7]. The predicted times according to the document are actually the times under best conditions[2]. This makes the predicted times of marginal value in real-time applications. In order to get a complete view of the instruction execution times, all of the important AED instructions must be measured on the actual machine.

[2]Best conditions are when the instruction is in ROM and data is in the local cache memory.

The overhead needed to measure the instruction (the iteration time and the two clock read times) can be found by subtracting the Null instruction from the time for the instruction 'A = 1'. If the overhead is assumed to consist of only the loop instructions then the amount of overhead per instruction iteration is 15.0 − + .039 μSeconds. This is derived by subtracting the time for 'A = 1' from the 'Null' instruction time, giving the overhead per instruction iteration. The resulting confidence interval is the sum of the 'Null' and 'A = 1' confidence intervals. Because the 'A = 1' interval is

This overhead is useful for calculations in other experiments.

Using 'A = B + C' as an average AED instruction, a rough order of magnitude of the number of instructions that can be executed in an R4 frame and the rough high level throughput of a triad can be calculated.

```
(40mS per R4 frame)/(27.0μS per instruction)
= 1500 instructions per R4 frame

1/(27.0μS per instruction)
= 37KOPS High Level Throughput
```

The instruction 'A = B + C;' actually used four assembly instructions. Therefore, a rough assembly level throughput would be 150KOPS.

4.3 Measuring R4 Frame Size

In the first part of the experiment the time that represented one R4 frame varied considerably. Since time was measured at the beginning of the first R4 task of a frame, these variations are probably due to variations in the execution time of the R4 dispatcher. There may be predictible variation caused by the starting of R3 and R1 tasks. However, this could not be determined without a large number of consecutive R4 frame start times, which was not done.

The nominal R4 frame measures in the single and double triad cases are shown:

```
Single Triad
  40.0 mSeconds average
    .741 mSeconds Standard Deviation
  37.75 -- 42.25 mSec. Range

Double Triad
  40.0 mSeconds average
    .623 mSeconds Standard Deviation
  37.75 -- 42.25 mSec. Range
```

The distributions of frame sizes are shown in Figures 4-1 and 4-2. The distribution looks approximately normal except that the frame sizes near the average occur less frequently than would

```
Size freq
(mS)
----------
37.8( 2) *
38.0( 3) *
38.3( 11) **
38.5( 5) *
38.8( 35) ****
39.0( 30) ***
39.3( 42) *****
39.5( 14) **
39.8(146) ***************
40.0(126) *************
40.3(133) **************
40.5( 19) **
40.8( 45) *****
41.0( 23) ***
41.3( 48) *****
41.5( 11) **
41.8( 7) *
42.0( 1) *
42.3( 2) *
```

Figure 4-1: Single Triad R4 Frame Distribution

```
Size freq
(mS)
-----------
37.8( 1) *
38.0( 0)
38.3( 2) *
38.5( 5) *
38.8( 5) *
39.0( 17) **
39.3( 60) ******
39.5(190) *******************
39.8( 59) ******
40.0( 29) ***
40.3( 87) *********
40.5(120) ************
40.8( 92) **********
41.0( 15) **
41.3( 11) **
41.5( 5) *
41.8( 1) *
42.0( 0)
42.3( 1) *
```

Figure 4-2: Double Triad R4 Frame Distribution

In the second part of the experiment, the R4 frame was stretched. The results of the stretching were:

```
2000 iterations
  80.8 mSec. Ave.
  .480 mSec. Standard Deviation
  80.5 -- 83 mSec. Range

3000 iterations
  108 mSec. Ave.
  .480 mSec. Standard Deviation
  107.75 -- 110.5 mSec. Range

5000 iterations
  163 mSec. Ave.
  .481 mSec. Standard Deviation
  162.25 -- 165 mSec. Range
```

In all of these runs at least 70% of the data points were in the two smallest frame sizes of the total range. The rest of the data points were spread out over the rest of the range as seen in Figures 4-3,4-4 and 4-5. The task that is stretching the frame can be considered to take constant time since the major portion is a loop of 'A = 1' which has been shown to be very predictable (see experiment 2). This would indicate that the variation might be due to some kernal procedures or the R4 dispatcher. The large number of data points in the first two consecutive frame sizes might indicate a basic frame size when no unusual situations occur. The other data might suggest exceptions that caused some extra instructions to be executed. The range of data is about 2.5mSeconds. Using the approximations from experiment 2, the amount of extra code that would be executed during these unusual situations is on the order of 100 AED instructions or 400 assembly level instructions.

When the average times were plotted against the iteration rate, a linear relation emerged as depicted in Figure 4-6. The step function (also shown on the graph) was expected because the documentation discusses a timer interrupt that, according to our interpretation, was supposed to happen every 24mSeconds. In fact; after the first timer interrupt, 24mSeconds into the R4 frame, the timer was not used until all of the R4 tasks finished. Therefore, the size of the frame would increase linearly above 40mS.

The final part of the experiment was to determine the behavior of the FTMP system when it executed an 'infinite' number of R4 tasks. In the experiment, an R4 task pointed to itself as the next task. If there were no mechanism for aborting a frame, as would be expected from the previous section of the experiment, the R4 frame would continue forever. This could be demonstrated by watching the

```
 size  freq
(mS)
-----------
80.5(238) ************************
80.8(272) ****************************
81.0( 40) ****
81.3( 33) ****
81.5( 30) ***
81.8( 52) ******
82.0( 20) **
82.3(  5) *
82.5(  5) *
82.8(  4) *
83.0(  1) *
```

Figure 4-3: Stretched Frame -- 2000 Iterations

```
 size  freq
(mS)
----------
107.8(235) ************************
108.0(261) ***************************
108.3( 38) ****
108.5( 42) *****
108.8( 36) ****
109.0( 51) ******
109.3( 23) ***
109.5(  9) *
109.8(  2) *
110.0(  2) *
110.3(  0)
110.5(  1) *
```

Figure 4-4: Stretched Frame -- 3000 Iterations

```
 size  freq
(mS)
------------
162.3(216) **********************
162.5(304) *******************************
162.8( 37) ****
163.0( 17) **
163.3( 38) ****
163.5( 53) ******
163.8( 22) ***
164.0(  5) *
164.3(  1) *
164.5(  5) *
164.8(  1) *
165.0(  1) *
```

Figure 4-5: Stretched Frame -- 5000 Iterations

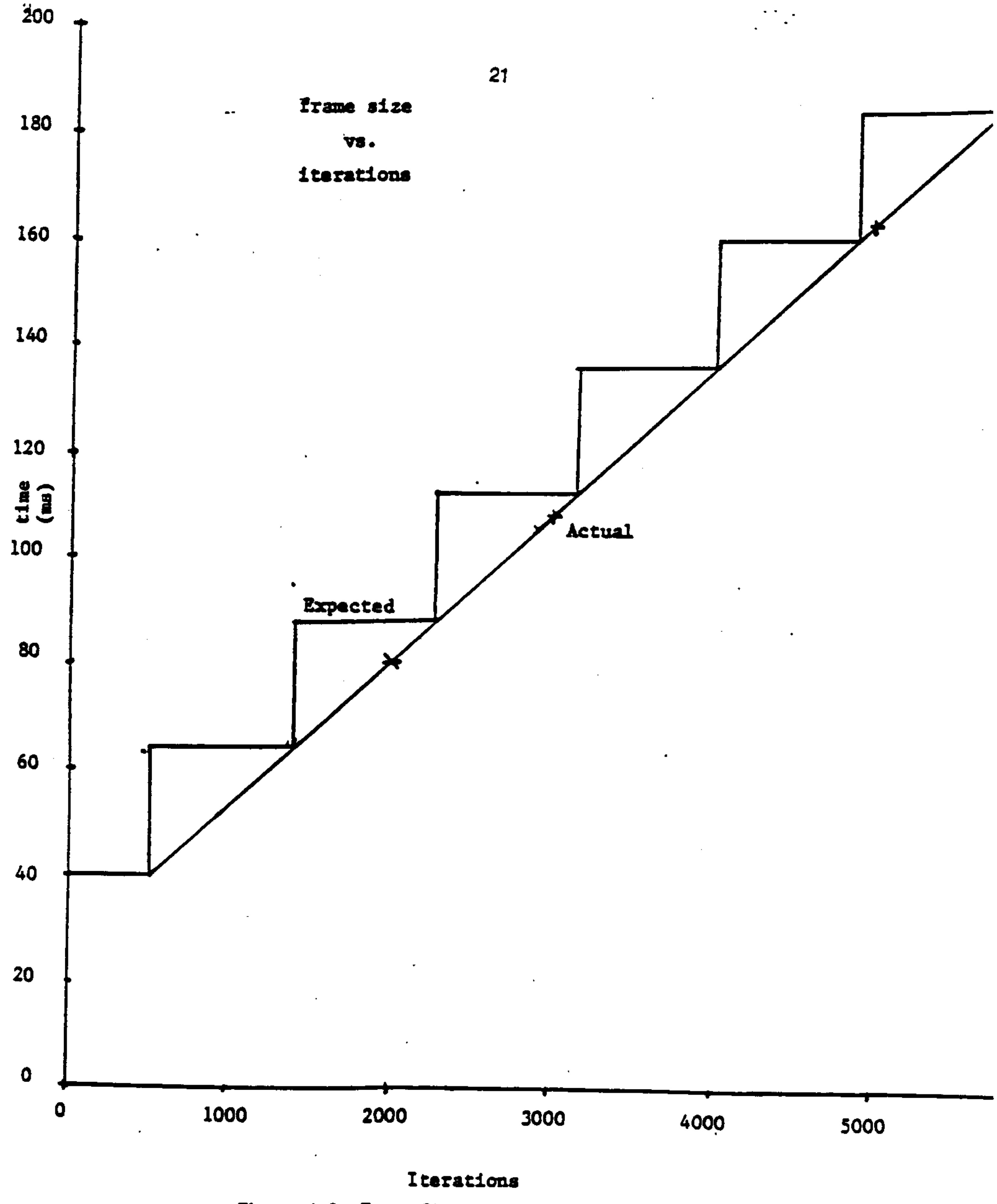

Figure 4-6: Frame Size (mSeconds) vs. Iteration Count

FTMP system display, driven by an R1 task. When the system is running this display is updated constantly (the time is updated every two seconds). When the 'infinite' list of R4 tasks was set up and started the display immediately stopped updating. When the recursive task pointer was nullified, resulting in a finite list, the display began updating again. This shows that an 'infinite' list of R4 tasks will not be aborted.

This last test points out a flaw in the scheduling software. Although tasks are regulated by giving them time limits, frames are not limited in this manner. A frame of any rate is simply stretched until all of the tasks within the frame can finish. This mechanism is not reliable in at least two situations. The first was described above, in which all other tasks were locked out by one task that pointed to itself. Another possibly hazardous situation would be a task with its time limit set too high. If, in most cases, the task takes much less time than the limit, this error may not be noticed. However, if some untested section of code starts a long, virtually infinite loop, the system will hang (at least at the frame size that the task uses) until that task has stopped. In a real-time application this is equivalent to failing.

5. Summary and Future Work

This paper described three experiments that are part of a methodology for fault-free validation of reliable multiprocessors. These experiments were concerned with baseline measurements run on the system. The major results of these experiments were:

1. The real-time clock is a reliable measurement device and was used in several experiments.
2. The instruction execution times are constant and reproducible. The measured times are slower than the documented best times.
3. The frames are nominally 40 milliseconds long. There is a variation of up to 2.5 milliseconds in the measurements. This is on the order of one hundred AED instructions.
4. The frame stretching mechanism allows linear increase in the size of the frame depending on the number of instructions to be executed, not stepwise increase as expected from the documentation.
5. Frame stretching continues until all tasks finish or abort. This is unreliable in some cases.

More work needs to be done to fully characterize the FTMP system. This is especially true of instruction and procedure call measurements. Major omissions from the present results were the call/return times for different types of procedures and the system reaction to arithmetic faults. Other AED instructions should also be measured to get a more complete evaluation of the system.

Enhancement of the experiment environment is planned. The goal of the enhancement is to have the capability of running several different experiments on FTMP by changing certain values in the global memory. With this environment it is hoped that information can be collected on the time to run various sizes and types of tasks in combinations. Information on scheduling and other operating system overhead might also be obtained with this environment.

6. Acknowledgement

We wish to acknowledge the help of personnel of AIRLAB at NASA/Langley Research Center. We would especially like to thank Carlos Liceaga, Frank Hill, Dan Coppen, Brian Lupton, George Finnelli and Dale Holdon. We would also like to thank Matt Reilly for his initial work on the FTMP system and Frank Feather for his help in the data analysis.

References

[1] Draper Labs.
AIPS System Requirements.
Technical Report, Charles Draper Laboratory, 1983.

[2] Ferrari, D.
Computer System Performance Evaluation.
Prentice-Hall, Inc., 1975.

[3] *FTMP Manual -- Volume I, II, III*
1981.

[4] Hopkins, A.L., et. al.
FTMP - A Highly Reliable Multiprocessor.
IEEE Trans. on Computers , October, 1978.

[5] Thomas H. Kong.
Measuring Time for Performance Evaluation of Multiprocessor Systems.
Master's thesis, Carnegie-Mellon University, November, 1982.

[6] Research Triangle Institute.
Validation Methods Research for Fault-Tolerant Computer Systems-- Preliminary Working Group II Report.
Technical Report, NASA-Langley Research Center, 1979.

[7] Rockwell International.
CAPS FTMP Instructions -- Execution Times.
Internal Letter.
FTMP Note #54.

[8] Z. Segall, A. Singh, R. T. Snodgrass, A. K. Jones, D. P. Siewiorek.
An Integrated Instrumentation Environment for Multiprocessors.
IEEE Transactions on Computers C-32(1), January, 1982.

[9] Wensley, J.H., et. al.
SIFT: A Computer for Aircraft Control.
IEEE Trans. on Computers , October, 1978.