NASA Contractor Report 177967

NASA-CR-177967
19860001381

FAULT-FREE BEHAVIOR OF RELIABLE MULTIPROCESSOR SYSTEMS: FTMP EXPERIMENTS IN AIRLAB

Ed Clune, Zary Segall, and
Daniel Siewiorek

CARNEGIE-MELLON UNIVERSITY
Pittsburgh, Pennsylvania

Grant NAG1-190
August 1985

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

1 Report No.	2 Government Accession No	3 Recipient's Catalog No
NASA CR-177967		

4 Title and Subtitle	5 Report Date
Fault-Free Behavior of Reliable Multiprocessor Systems: FTMP Experiments in AIRLAB	August 1985
	6 Performing Organization Code
7 Author(s) Ed Clune, Zary Segall, and Daniel Siewiorek	8 Performing Organization Report No
	10 Work Unit No
9 Performing Organization Name and Address Carnegie-Mellon University Pittsburgh, PA	11 Contract or Grant No NAG1-190
	13 Type of Report and Period Covered Contractor Report
12 Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546	14 Sponsoring Agency Code 505-34-13-32

15 Supplementary Notes

Langley Technical Monitor: George B. Finelli

16 Abstract

This report describes a set of experiments which were implemented on the Fault Tolerant Multi-Processor (FTMP) at NASA/Langley's AIRLAB facility. These experiments are part of an effort to formulate and evaluate validation methodologies for fault-tolerant computers. This report deals with the measurement of single parameters (baselines) of a fault free system.

The initial set of baseline experiments lead to the following conclusions:

1. The system clock is constant and independent of workload in the tested cases;
2. The instruction execution times are constant;
3. The R4 frame size is 40mS with some variation;
4. The frame stretching mechanism has some flaws in its implementation that allow the possibility of an infinite stretching of frame duration.

Future experiments 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.

17 Key Words (Suggested by Author(s))	18 Distribution Statement
Validation Fault-tolerant Multiprocessors Performance measurement Synthetic workload Fault-free	Unclassified - Unlimited Subject Category 62

19 Security Classif (of this report)	20 Security Classif (of this page)	21 No of Pages	22 Price
Unclassified	Unclassified	32	A03

Table of Contents

List of Figures

List of Tables

Executive Summary

This is a report on research by Carnegie-Mellon University for NASA-Langley Research Center under contract NAG-1-190 on the validation of fault-tolerant avionics multiprocessors This report covers the initial phase of experimentation.

In this period a series of basic performance measurements were conducted on the Fault Tolerant Multi-Processor (FTMP) as part of a process to evaluate validation methodologies. The results of these experiments and proposals for future work are presented in this report.

A paper, based on this work, entitled "Validation of Fault-Free Behavior of a Reliable Multiprocessor System — FTMP: A Case Study" was presented at the American Control Conference in June 1984.

Abstract

This report describes a set of experiments which were implemented on the Fault Tolerant Multi-Processor (FTMP) at NASA/Langley's AIRLAB facility These experiments are part of an effort to formulate and evaluate validation methodologies for fault tolerant computers. This report deals with the measurement of single parameters (baselines) of a fault free system.

The initial set of baseline experiments lead to the following conclusions:

1 The system clock is constant and independent of workload in the tested cases,
2 The instruction execution times are constant;
3 The R4 frame size is 40mS with some variation;
4. The frame stretching mechanism has some flaws in its implementation that allow the possibility of an infinite stretching of frame duration

Future experiments 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]

[1]This research was sponsored by the National Aeronautics and Space Administration, Langley Research Center under contract 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

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, 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[5, 6, 11]) Techniques must be developed for measuring the performance and reliability of these systems.

Since a probability of failure of 10^{10} per hour translates to less than one failure per million years of operation, it is not feasible to wait for enough accumulated operational hours to demonstrate compliance with the goal prior to the release of the aircraft computer A comprehensive validation methodology can greatly reduce the amount of time required to determine if a system meets its design goals. An overall validation methodology has many components including theorem proving, mathematical modeling, and physical experimentation. NASA has held several workshops to develop a system validation procedure One workshop in particular [8] produced a detailed list of validation tasks to verify a system in an orderly manner

Theorem proving and mathematical modeling are often based on a simplification and abstraction of the physical system These simplifications are required to reduce the complexity of the mathematics to a tractable level Experimentation is a key element in a validation methodology since it serves to validate the model and abstractions assumed in the mathematical treatments as well as to discover unanticipated phenomena.

The foundations for an experimental validation methodology have been developed and are being tested at the Avionics Integrated Research Laboratory (AIRLAB) at the NASA Langley Research Center 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

In parallel, Carnegie-Mellon University had developed several multiprocessor systems including C mmp, a system with 16 processors communicating with 16 memories through a crossbar switch [12], and Cm*, a 50 processor system with a hierarchical processor-memory switch [10] Over the past decade researchers at CMU were evolving experimental methodologies for evaluating these multiprocessor systems AIRLAB provided an opportunity to apply and extend the experimental methodologies to real time fault tolerant multiprocessor systems The generality of the experimental

methodology could be demonstrated by its application to four diverse multiprocessor systems while producing actual measurements which compared and contrasted these architectures.

The remainder of this document is organized as follows. Section 2 gives background on the validation methodology used as the basis of the baseline experiments It also introduces the Fault Tolerant Multiprocessor (FTMP), on which the baseline experiments were performed, and gives a short description of the types of baseline experiments that were run on FTMP Section 3 describes the basic experimental structure and the variants used to measure different baseline parameters In Section 4, the results of the experiments are presented and conclusions drawn A summary of the experiment results and a discussion of future work are presented in Section 5.

2. Background

This section contains information necessary to understand the motivation for the experiments discussed in the next section. Included are a description of the validation methodology used as a basis for the experiments and a description of the Fault Tolerant Multi-Processor (FTMP).

2.1 Proposed Methodology

NASA held several workshops to determine system validation procedures One in particular [8], produced a detailed list of validation tasks to verify a system in an orderly manner. The methodology was based on a building block approach in that confidence would be built up in an incremental manner through the understanding and measurement of primitive activities Once these primitive activities were characterized, more complex experiments would be devised to explore the interaction of primitive activities as well as more complex activities constructed from these primitive activities This orderly progression insures uniform coverage as well as maximizes the ability to locate the cause of an unexpected phenomenon A modified version of this list is shown in Figure 2-1.

1 Fault Free Evaluation

- a Initial Checkout and Diagnostics
- b. Programmer's Manual Verification
- c. Executive Routine Verification
- d Multiprocessor Interconnect Verification
- e Multiprocessor Executive Routine Verification
- f Application Program Verification and Performance Baseline

2 Fault Handling Evaluation

- a Simulation of Inaccessible Physical Failures
- b. Single Processor Fault Insertion
- c Multiprocessor Fault Insertion
- d. Single Processor Executive Failure Response Characterization
- e Multiprocessor System Executive Fault Handling Capabilities
- f. Application Program Verification on Multiprocessor
- g Multiple Application Program Verification on Multiprocessor

Figure 2-1· System Evaluation List

The first set of six tasks verifies the fault free functionality of the system while the next set of seven verifies fault handling capabilities The reader is referred to [8] for a more detailed explanation of the above listed tasks The experiments described in this paper deal only with the set of fault free performance evaluation tasks The experiments run on FTMP were actually involved in performance baselines (part of task 1f) although verification at other levels was accomplished as well (for example, Executive Routine Verification, task 1c) while running baseline experiments

2.2 Experiment Environment

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 2-1 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 2-2: Levels of Abstraction in Multiprocessor Systems

Our experience at CMU indicates multiprocessors go through a series of evolutionary 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 This functionality can usually be defined in terms of the activities in the life of an experiment First, the code has to be designed and written Next, the code must be compiled, followed by loading, debugging, measurement, and analysis Consequently, a 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 baseline 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 baseline experiments in Stage 2 include·

- **Measurements of the cost-per-feature of the operating system's functions** Experiments statically exercise 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 [9] 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 Furthermore, the effort to conduct advanced experiments becomes manageable 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 advanced experiments that can be conducted in Stage 3 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.
- 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 [9] Such a workload for avionics applications is given in [2] 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 Present and Future Experiment Environment

A significant 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 by changing values in memory without having to go through the entire code development cycle An example of a modifiable variable is the number of iterations in a loop 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 restricted 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 executions 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

Prior to presenting a detailed definition of a set of baseline experiments, we will first describe the experimental vehicle so that the reader can observe how individual baseline experiments have to be modified to take into account the specific implementation of an architecture

2.4 The Fault Tolerant Multi-Processor (FTMP)

The Fault Tolerant Multi-Processor (FTMP) has been discussed in several papers and manuals [5, 6] This section will only describe those details necessary for understanding the experimental results. For more details the interested reader is referred to the literature.

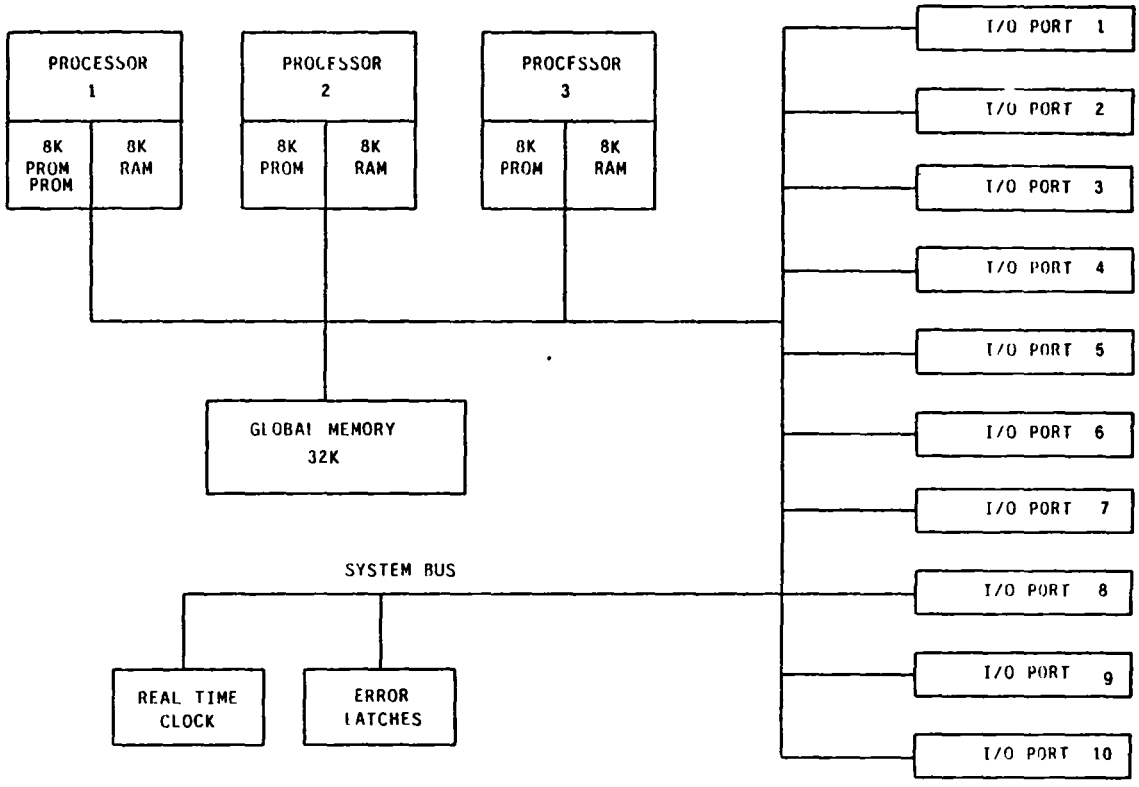

Figure 2-3: FTMP System

Figure 2-3 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 Due to the real-time nature of the application, triads do not necessarily execute the same tasks in the same order 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 select 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 2-4 Each triad has separate pointers to tasks for each rate group.

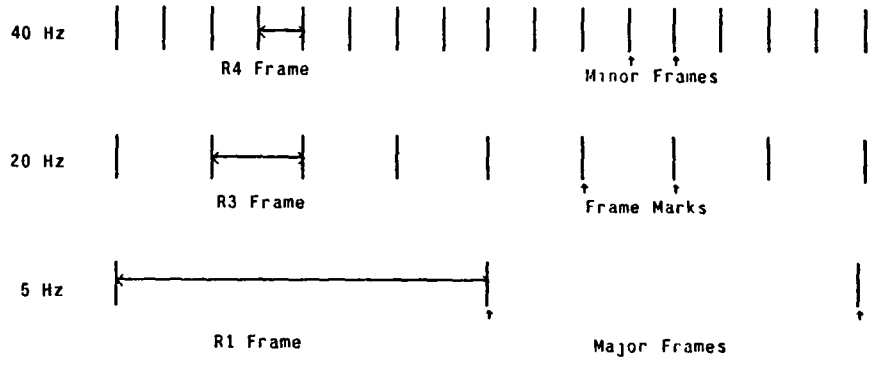

Figure 2-4: Frame Structure

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 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

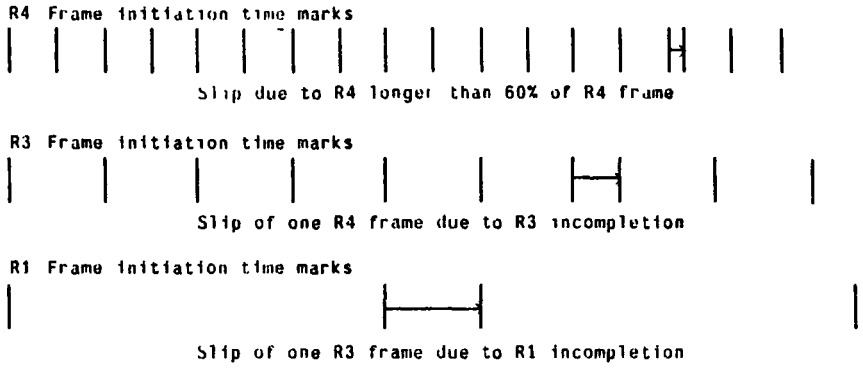

Figure 2-5: Frame Stretch Mechanisms

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 2-5 An R4 frame is stretched by a specific amount and R3 and R1 frames are stretched by giving them more R4 frames. The third baseline 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 functions RD and WRT Knowing the amount of time that these transfers take and how the time can vary are crucial to understanding the accuracy of measurements.

Several computer systems are used to run 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 4341 Load files are transferred to a VAX-11/750 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.

2.5 Proposed Experiments

A candidate set of baseline experiments was organized according to the levels of abstraction depicted in Figure 2-2:

- Instruction Set Level

- Assembly and High-level language instruction times.

- Executive Software Level

  - Executive primitive and overhead times
  - Interrupt procedure times
  - Memory access time
  - Bus access and contention delays
  - Fault tolerance overheads

- System and Application Level

  - Frame utilization characteristics (including OS overhead and bus contention delay and fault tolerance overhead)

The specific baseline 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

The goal of the experiments described in this report is to obtain simple performance measures of processor and operating system functions. The method used to do this consists of three steps:

- Record start time
- Perform operation(s)
- Record end time

Variations in implementing this approach are due to the constraints of the FTMP system The experiments use a framework as in Figure 3-1, which is described in the following paragraphs A framework related to this one has been used to implement a synthetic workload environment on FTMP [3]

```
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 experimental 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 clock interval times

Using the VAX/FTMP interface program called CTA and a batch command script, the values in the global array can be read and stored in a file CTA can also set 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 Thus a great deal of experimentation overhead is saved.

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 the Cm* multiprocessor system, there was as much as 4 6% difference in clock frequency, and substantial variation of clock read delays [7] 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 Referring to the framework in Figure 3-1, 'SomeInstructions' is replaced by 5 consecutive clock reads and RTCNUM becomes 16 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 be the second task to start execution 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 contention for the common clock resource 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 Again, referring to Figure 3-1, 'SomeInstructions' is replaced by 'A = 1,' and the instruction being measured and RTCNUM becomes 100 This task was executed 308 times.

3.3 Measuring R4 Frame Size

There were three parts to this experiment In the first part, time from the real-time clock was read at the beginning of the first R4 task Referring to Figure 3-1, this time was stored in the CMU TIME array if appropriate (depending on the value of CMU EXEC) and the instruction 'A = 1,' was executed a specified number of times as determined by CMU RTCNUM.

If CMU EXEC was above the value eight, the task would finish without doing anything else Eight consecutive R4 TASK1 start times were stored in each iteration of the experiment The objective was to determine the R4 frame duration The experiment was run about 100 times for a single 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 a rate group with an infinite number of tasks This could be easily done because each task had control information associated with it. One of the words of information 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 For the two different triads the results were

```
Triad 1:    55.9    _+  047    ticks/ 16 iterations (95% confidence[2])
            14.0    _+ .012    mSec / 16 iterations
              .874  _+ .00073  mSec / iteration

Triad 2:    56.0    _+  036    ticks/ 16 iterations
            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 (μSec)	Triad 2 (μSec)	
874 ± 73 -15 7 ± 11	875 ± 57 -15 7 ± 11	Initial Data Overhead
858.3 ± 84 /5	859 3 ± 68 /5	Number of Reads
172 ± .17	172 ± 14	Clock Read Time

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 The result for two triads was

56 3 ± 091	Ticks / 16 Iterations
173 ± 31	μSec / Clock 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%) However, the contention is large enough that the range of the 95% confidence

[2] All intervals are 95% confidence intervals assuming normal distribution for the variables Refer to Ferrari [4] for a description of confidence intervals and how they are calculated

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 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

Instruction	Clock Ticks per 100 Instr.(ave)	μSec per Instruction, w/ Overhead	μSec per Instruction, w/o Overhead	μSec per Instruction, Predicted
Null	12 3	30 7 ± 013		
Integer Assignment	18.3	45 7 ± 013	15 0 ± 026	8 3
Real Assignment	18 4	46 1 ± 014	15 4 ± 027	8 3
Long Assignment	19 6	49 1 ± 014	18 4 ± 027	12 3
Integer Addment	23 0	57 7 ± 004	27 0 ± 017	22 3
Real Add	23 2	58 0 ± 011	27 3 ± 024	22 3
Long Add	27 4	68 6 ± 014	37 9 ± 027	30 0
Integer Multiply	25 1	62 9 ± 010	32 2 ± 023	27 4

Table 4-1: Instruction Results

The result of the measurements are shown in Table 4-1 The three times given for each instruction are as follows The first column is the time to execute each instruction including the overhead of reading the clock, maintaining the loop, and the time to execute 'A = 1,' The second column adjusts the time from the first column by subtracting out these overheads The third column represents time per instruction predicted by the assembler The range is for a 95% confidence interval

The results showed little variance The number of 'clock ticks' per frame varied only by one for each AED instruction. The instructions took longer than suggested by the times given by the assembler and Draper Labs documents. The predicted times according to the document are actually the times under best conditions 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.

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 $\pm$ 039 μSeconds This overhead is useful for calculations in other experiments.

Another aspect of the looping overhead is the error due to the clock resolution On average this turns out to be half a clock tick This value would be subtracted from any absolute time average to give the actual average time that was measured

Using 'A = B + C' as an average high level 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.

$$\frac{40mS/R4Frame}{27.0\mu S/Instruction} = 1500 Instructions/R4Frame$$

$$\frac{1}{27.0\mu S/Instruction} = 37 KOPS(AED) Throughput$$

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

Triads	Average Time (mSeconds)	Standard Deviation (mSeconds)	Range (mSeconds)
Single	40.0	741	37.75 - 42.25
Double	40.0	623	37.75 - 42.25

Table 4-2: Frame Measurement Results

R4 frames varied considerably in size (the amount of time between consecutive frames) from one

frame to another. There may be cyclic variation, however it is hard to determine from the method used to obtain the data. The nominal R4 frame measures in the single and double triad cases are shown in Table 4-2 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 be expected. The reason for this is unclear.

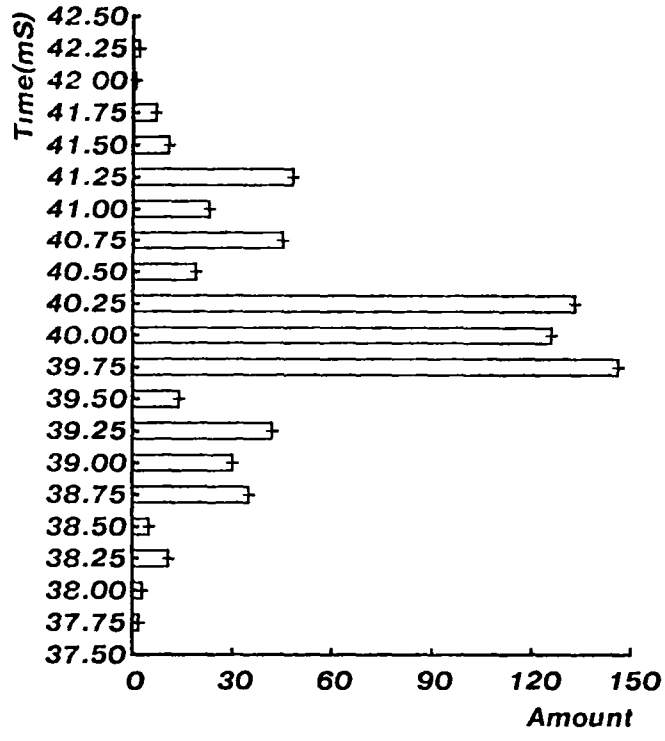

Figure 4-1: Single Triad R4 Frame Distribution

In the second part of the experiment, the R4 frame was stretched The results of the stretching are shown in Table 4-3 In all of these runs, the time measured for a frame was usually close to the average (within a few clock ticks) with some taking several ticks longer and none taking more than 2 clock ticks less than the average (see Figures 4-3,4-4 and 4-5) The reason for this distribution is again unknown, but it is probably due to the operating system and dispatcher variations rather than the task that runs within the frame (see experiment 2) The actual variations compare to roughly nine instructions per tick This could be the difference due to one conditional (if — then — else) statement.

When the average times were plotted against the iteration rate, a linear relation emerged (see Figure 4-6). From the documentation, a step function increase was assumed with a step of 24 mSeconds This is also shown on the graph When the actual code was read, the linear increase was to be

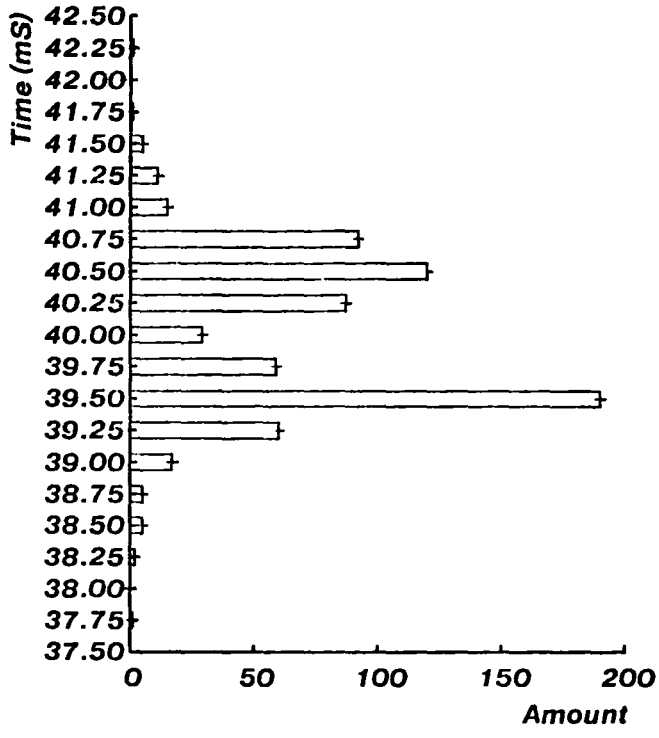

Figure 4-2. Double Triad R4 Frame Distribution

Frame Size (Iterations)	Average Time (mSeconds)	Standard Deviation (mSeconds)	Range (mSeconds)
2000	80 8	480	80 5 - 83 0
3000	108	480	107 8 - 110 5
5000	163	481	162 3 - 165 0

Table 4-3: Frame Stretching Results

expected The reason for the supposed step function was a timer interrupt that was to happen every 24 mSeconds In fact, after the first timer interrupt, 24 mSeconds into the R4 frame, the timer was not used until 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 system when an infinite set of R4 tasks was started In the experiment, an R4 task pointed to itself as the next task If there were no mechanism for aborting a frame, the R4 frame would continue forever This could be shown, by attempting to use another task while the R4 task continues to loop For this experiment the task that was used to test whether the system was running was an R3 task that failed and restored processors. Normally, it took only a few seconds from entering a request to reconfiguring the system However,

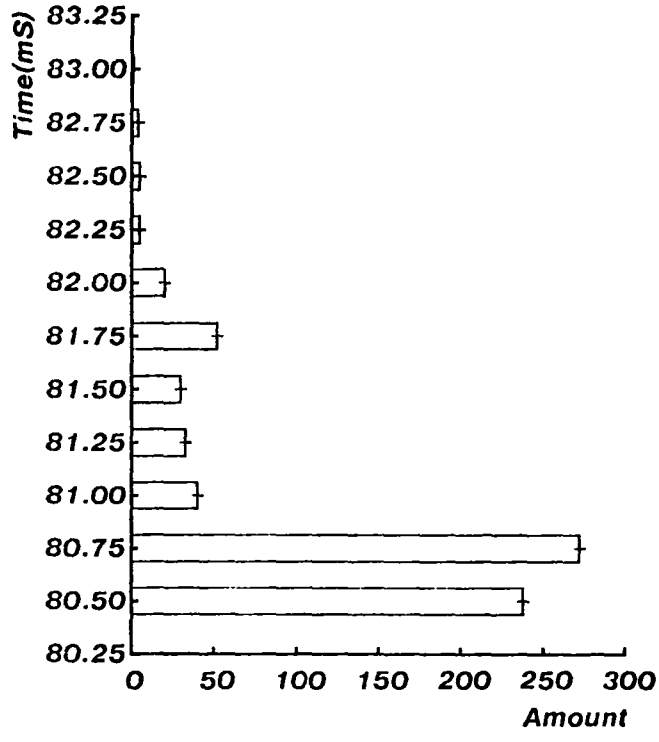

Figure 4-3· Stretched Frame — 2000 Iterations

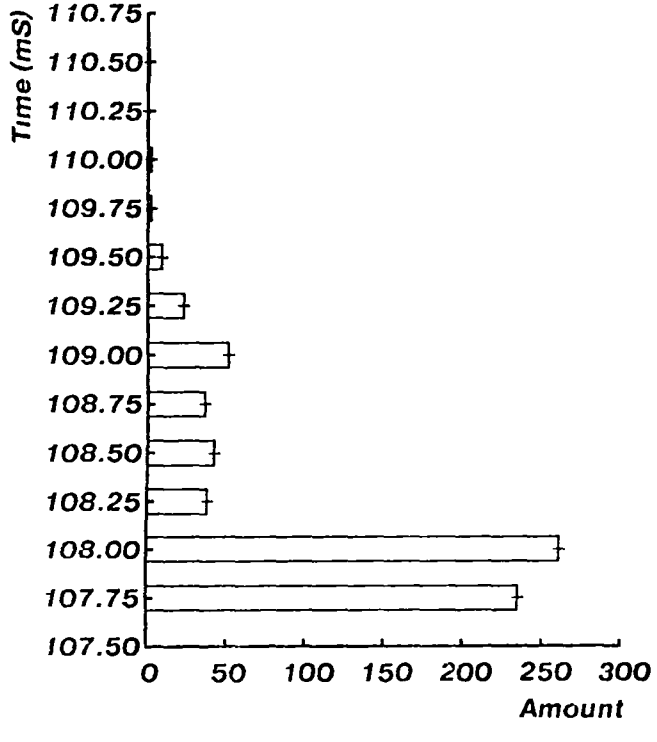

Figure 4-4 Stretched Frame — 3000 Iterations

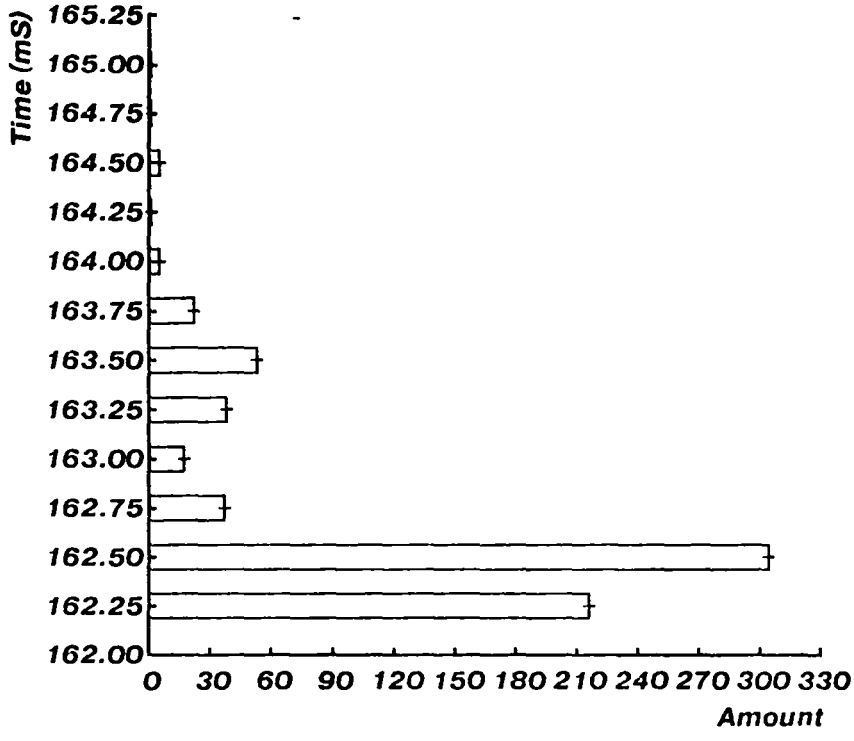

Figure 4-5: Stretched Frame — 5000 Iterations

when the R4 task began to repeat infinitely, the R3 task could not execute at all When the infinite loop was stopped (by nullifying the R4 pointer), the R3 task ran immediately.

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 the code starts a long, virtually infinite loop, the system will hang (at least at that rate group) until that task has stopped In a real-time application this is equivalent to failing

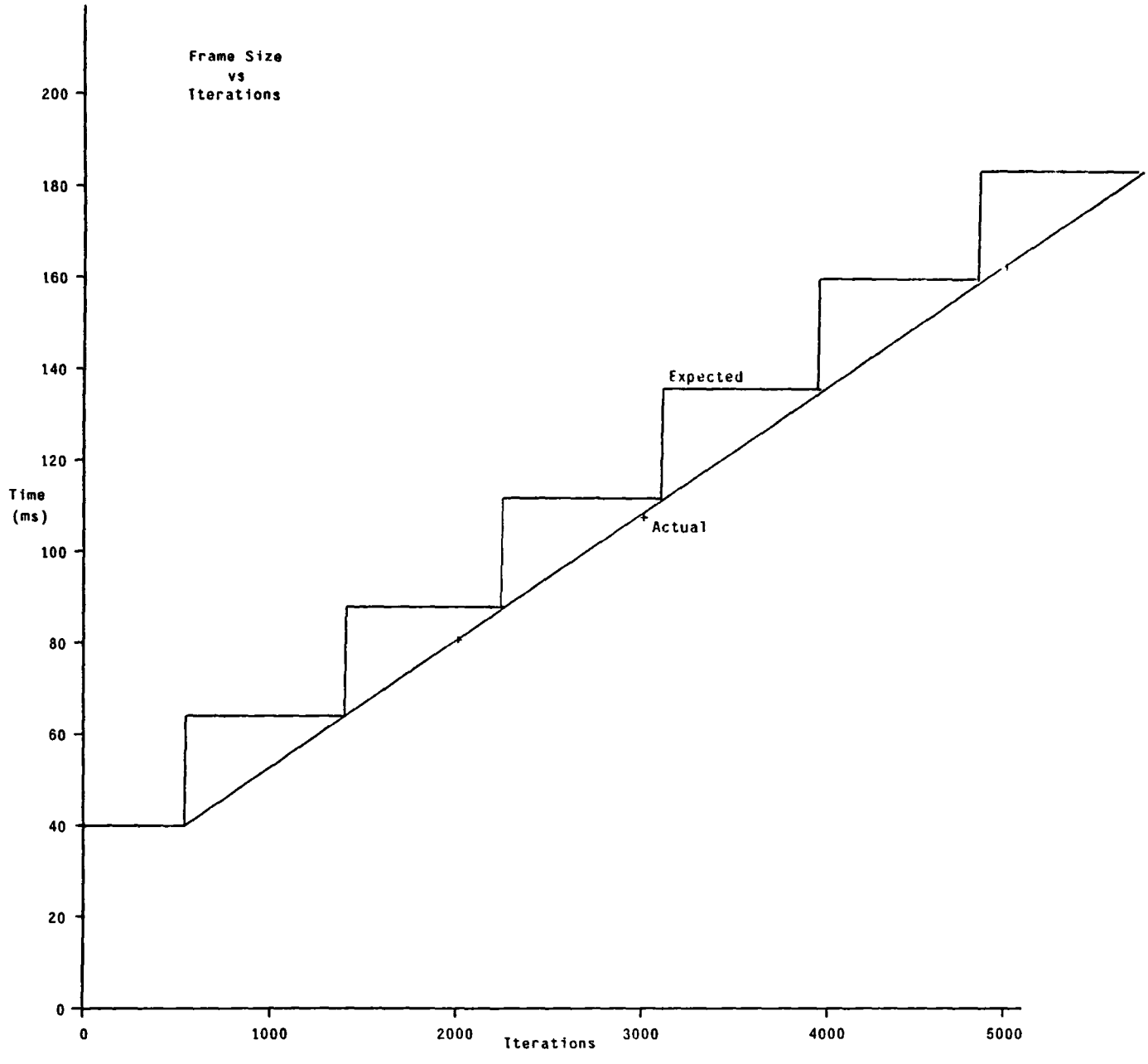

Figure 4-6: Frame Size (mSeconds) vs Iteration Count

5. Summary and Future Work

This paper described three experiments that were designed within the framework of a validation methodology The methodology was derived earlier and is undergoing changes as experience increases The experiments were concerned with baseline measurements of the running system The major results of these experiments were:

1 The real-time clock is a reliable measurement device and can be used in timing 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 many clock ticks in all measurements.

4 The stretching mechanism allows a linear increase in the size of the frame depending on the number of instructions to be executed, not a stepwise increase as expected from reading 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 of 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 only changing certain values in memory With this environment it is hoped that information can be collected on the time to run various sizes and types of tasks in many combinations Information on scheduling and other operating system overhead might also be obtained with this environment

6. Acknowledgment

We wish to acknowledge the help of all of the people of AIRLAB at NASA/Langley Research Center. We would especially like to thank Carlos Liceaga, Frank Hill, Dan Koppen, Brian Lupton, George Finelli and Dale Holden 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] Clune, E.
Analysis of the Fault Free Behavior of the FTMP Multiprocessor System: Baseline Measurements and Synthetic Workload.
Master's Project, Carnegie-Mellon University, September, 1984.

[2] Draper Labs.
AIPS System Requirements.
Technical Report AIPS-83-50, Charles Draper Laboratory, 1983.

[3] Feather, Frank E.
Validation of Fault-Free Behavior of a Reliable Multiprocessor System. Workload Implementation.
Master's Project, Carnegie-Mellon University, March, 1985.

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

[5] *Development and Evaluation of a Fault Tolerant Multiprocessor (FTMP) Computer, Volume I, II, III, IV*.
Contract Reports 166071, 166072, 166073, 166074.
Draper Laboratories, 1983.

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

[7] Kong, T. H
Measuring Time for Performance Evaluation of Multiprocessor Systems.
Master's Thesis, Carnegie-Mellon Universiy, November, 1982.

[8] Research Triangle Institute.
Validation Methods Research for Fault-Tolerant Computer Systems-- Preliminary Working Group II Report.
NASA Conference Publication 2130, NASA-Langley Research Center, 1979.

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

[10] Swan, R. J., S. H. Fuller, D. P. Siewiorek.
Cm*: A Modular, Multi-Microprocessor.
Proc AFIPS NCC, vol. 46, 1977.

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

[12] Wulf, W. A., C. G. Bell.
C.mmp: A Multi-Mini-Processor.
Proc AFIPS FJCC, vol. 41, pt. 2, 1972.

End of Document