Fault Free Performance Validation of a Fault-Tolerant Multiprocessor: Baseline and Synthetic Workload Measurements

Edward W. Czeck, Daniel P. Siewiorek, Zary Segall

21 November 1985

Department of Electrical and Computer Engineering
Carnegie-Mellon University
Schenley Park
Pittsburgh, Pennsylvania 15213

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 author 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

Abstract

1. Introduction

2. Background

- 2.1 Proposed Validation Methodology
  - 2.1.1 NASA Workshops
  - 2.1.2 Performance Definitions
- 2.2 Fault-Tolerant Multiprocessor Structure
  - 2.2.1 FTMP Hardware
  - 2.2.2 FTMP Scheduling and Dispatcher Strategy
    - 2.2.2.1 Overview
    - 2.2.2.2 Uniprocessor-Multiprogrammed
    - 2.2.2.3 Multiprocessor-Multiprogrammed Dispatcher Strategy
  - 2.2.3 FTMP Fault Detection Software
- 2.3 Experimental Environment
- 2.4 Background and Previews to Experiments

3. FTMP Experiments

- 3.1 Instruction Execution Time Measurements
  - 3.1.1 Experimental Setup
  - 3.1.2 Experimental Results
  - 3.1.3 Instruction Combinations Executions Times
- 3.2 Block Transfer Times
  - 3.2.1 Experiment Set-Up
  - 3.2.2 Results of Block Transfers
- 3.3 Dispatcher-Scheduler Software Overhead
  - 3.3.1 Start-up Dispatcher Time
  - 3.3.2 IPC 'Kick' Times
  - 3.3.3 Intra-Task Group Switching
  - 3.3.4 Inter-Task Group Switching
  - 3.3.5 Frame Size and Frame Slippage
  - 3.3.6 Dispatcher-Scheduler Software Overhead Summary
- 3.4 Software Overhead for Fault Detection and Isolation

4. Performance Estimates

5. Future Work

6. Conclusions

I. Instruction Execution Times

II. Block Transfer Execution Times

List of Figures

Figure 2-1: FTMP Structure, Programmers Model. 9
Figure 2-2: FTMP LRU and Bus Interface [Draper 82]. 10
Figure 2-3: FTMP Task Frame Structure. 12
Figure 2-4: FTMP Dispatcher Scheduler Strategy, Showing Two Consecutive Frames. 14
Figure 2-5: FTMP Experimental Environment. 16
Figure 2-6: Representation of a Synthetic Workload Task. 18
Figure 3-1: FTMP Experiment Task Algorithm. 19
Figure 3-2: AED Instruction Execution Time Summary. 20
Figure 3-3: Read and Write Execution Times as a Function of Block Size, 1 Triad. 23
Figure 3-4: Read Execution Times as a Function of Block Size, 1 Triad and 2 Triads. 24
Figure 3-5: Write Execution Times as a Function of Block Size, 1 Triad and 2 Triads. 25
Figure 3-6: Typical Distribution for a Bus Service Routine with Competing Triads. 26
Figure 3-7: R4 Dispatcher Execution Times, 1, 2, and 3 Triads. 28
Figure 3-8: IPC 'Kicks' Timing Diagram. 29
Figure 3-9: Time and Variation Between the Starts of the Application Tasks on Different Processor Triads. Emulating IPC Kick Times. 30
Figure 3-10: Intra-Task Group Switching Times in Milliseconds. 32
Figure 3-11: Inter-Task Group Switching Times in Milliseconds. 34
Figure 3-12: Actual Frame Size for R4 Tasks. 35
Figure 3-13: Actual Frame Size for R3 Tasks. 36

List of Tables

Table 2-1: Performance Evaluation Matrix. 6
Table 3-1: Instruction Combination Execution and Predicted Times. 21
Table 3-2: Dispatcher Software Overhead Summary. 38
Table 3-3: Software Overhead for Fault Tolerance. 39
Table 4-1: Performance Estimates for FTMP. 41
Table I-1: Instruction Executions Time: Integer 47
Table I-2: Instruction Executions Time: Real 48
Table I-3: Instruction Executions Time: Long Integers 48
Table I-4: Instruction Executions Time: Boolean 49
Table I-5: Instruction Executions Time: Miscellaneous Operators 49
Table II-1: Block Transfer Times, Read from System to Local Memory 51
Table II-2: Block Transfer Times, Write from Local to System Memory 51

Introduction

Abstract

Today's aircraft use on board computers to perform isolated functions. Soon aircraft must utilize ultrareliable computer systems to provide flight critical functions. This project outlines a *Validation Methodology* for ultrareliable computer systems and applies the methodology to the Fault Tolerant Multiprocessor (FTMP) system at NASA Langley's AIRLAB. The validation experiments applied to FTMP measured the operating system and fault tolerant software overheads on FTMP. The real time operating systems consumed 63% of the system throughput whereas the fault tolerant software required only 5%. Although the validation methodology was applied strictly to FTMP the methodology proved effective by uncovering both system implementation dependencies and undesirable behavior in the FTMP system.

1. Introduction

Aircraft today employ computers to perform isolated functions. If a computer fails, its tasks would be assumed by the aircrew without loss of life or cargo. Soon the designs of aircraft will require an on board real time computer to perform flight critical control functions. If such a computer were to fail the craft would be unable to fly. One study by the National Aeronautics and Space Administration (NASA) in its Aircraft Energy Efficiency (ACEE) Program required the probability of failure be less than 10^{-10} for a ten hour mission. Two multiprocessor systems designed to these specifications, SIFT [Wensley 78] and FTMP [Hopkins 78], have been delivered to NASA's Avionic Integrated Research Laboratory (AIRLAB).

This specified failure rate translates to less than one failure per million years of operations. Conventional validation methods, such as life testing, would be impracticable for a system designed to these constraints. Studies at NASA were conducted to determine system validation and verification methodologies, [NASA 79a, NASA 79b]. Two approaches were chosen: the first involved mathematical models and verification, and the second involved experiments to test the functionality, behavior, performance, and fault handling capabilities of the system.

The goal of this project is to apply the experimental methodology, developed for NASA, to a Fault-Tolerant Multiprocessor, FTMP, at NASA's AIRLAB. This report covers the following:

- A background of the experimental methodology, along with the FTMP structure, and present state of validation.
- A description of the experiments on FTMP and their result.
- A summary of the performance estimates based on the experiments.
- Future work for FTMP and an overall conclusion.

Background

2. Background

2.1 Proposed Validation Methodology

Underlying any methodology, there must be a set of guiding philosophies. Over the last decade, C-MU has dedicated over 100 man years of effort in the design, construction and validation of multiprocessor systems. A partial list of the experimental guidelines developed during the last decade include:

- The experimental validation methodology is successively refined as experiments uncover new information and the methodology is applied to new multiprocessor systems.
- Experiments are designed to validate behavior that is documented, as well as behavior that is not documented.
- Experiments are conducted in a systematic manner; since the search is for the unexpected, there is not shortcut to thorough testing.
- Experiments should be repeatable.
- The feasibility of performing various experiments is tempered by what is available in the experimental environment. More sophisticated experiments may have to be postponed until the experimental environment is provided with more tools.
- A building block approach should be used wherein one variable is changed at a time, so the cause of unexpected behavior is easy to isolate.
- Testing should take advantage of the structural (abstract) levels used in the design of the system.

With a Fault Tolerant, ultra-reliable system other problems arise that make the validation task difficult. Some of these problems are: [NASA 79b]

- Life testing is inappropriate.
- System design complexity makes it difficult to perform failure effect analysis, instrument and measure all relevant parameters, and use exhaustive testing approaches, since there are a large number of states and failure modes possible.
- Large scale integration makes access to control and observation points difficult as well as determining a confidence level for fault coverage.

2.1.1 NASA Workshops

NASA held several workshops to determine validation procedures. One [NASA 79b] in particular produced a detailed list of a validation procedure. The procedure is based on a building block approach. Primitive activities are characterized first. Once these activities are understood, complex experiments involving the interaction of primitive activities, as well as complex activities built from the basic primitives, may be run. The orderly progression insures uniform, thorough coverage and maximizes the ability to locate the cause of unexpected phenomena. The steps in the proposed methodology include:

1. Initial checkout and diagnostics.
2. Programmer's manual validation.
3. Executive routine validation.
4. Multiprocessor interconnect validation.
5. Multiprocessor executive routine validation.
6. Application program verification and performance baseline measurements.
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 failure response characterization.
12. Application program verification on multiprocessor.
13. Multiple application program verification on multiprocessor system.

The first six tasks in the list validate the fault free baseline functions of the system, items seven through eleven characterize the fault handling capabilities of the processors, and the last two validate the total integrated environment of the system.

2.1.2 Performance Definitions

Performance is measured in functions per unit time or the time needed to complete a specific task [Siewiorek, Bell, and Newell 82]. The notion of performance exists throughout the digital design hierarchy, from the circuit level (switching times), to the system (application task time) level. With this definition and the validation methodology a performance evaluation matrix can be created, Table 2-1. The vertical axis is the design hierarchy, while the horizontal axis is definitions or characterizations of performance.

	Behavior	Throughput	Utilization	Delay
Application	Correct Function In Integrated Environment.	Application Task Times. Flight Control, etc.	Idle Time.	Variation Caused by Shared Data, Increased Load.
Executive, Operating System.	Correct Operation of Scheduler, Dispatcher, etc..	Operating System Primitives Times.	O.S. Primitives Frequency of Use.	Variation Caused by Hardware and Data Contention.
Instruction Set, Hardware.	Correct Operation of Interrupts, etc..	Instruction, And Resource Times.	HW. Resource Frequency of Usage.	Variation Caused by Hardware Contention.

Table 2-1: Performance Evaluation Matrix.

In detail the entries for the table are described as follows:

- Instruction Set, Hardware Level:
    - Behavior: The operation of hardware primitives, such as interrupt and exception handling characteristics.

  - ○ Throughput: The time to execute basic primitives, instruction times, bus access, interrupts etc..
  - ○ Utilization: The frequency and percent usage of the hardware resources.
  - ○ Delay: Delay and variation caused by hardware contention.

- Executive, Operating System Level:
  - ○ Behavior: Validate operation of the executive software.
  - ○ Throughput: The execution time of dispatcher-scheduler, message systems, and other O.S. primitives.
  - ○ Utilization: The frequency and percent usage of the executive and operating system level resources.
  - ○ Delay: Executive primitive contention and delay due to hardware constraints, and common O.S. databases.

- Application Level:
  - ○ Behavior: Actions of the system and application software in the fully integrated environment.
  - ○ Throughput: The execution times of application (user) tasks and the total useful work accomplished by the system.
  - ○ Utilization: Frequency and percent usage of the combined operating system, hardware resources, and application tasks in relation to the total usable time. (i.e. Total available throughput less overhead times.)
  - ○ Delay: Variation caused by shared databases, hardware contention, and temporary work overloads.

Each element in the matrix is not singular and evaluation measures can overlap. The matrix can be used for both fault free and faulty performance measurements. In general, there is a building block approach, starting with baseline experiments and moving upto more complex experiments: the same approach referred to in the guiding philosophies of the validation methodology.

2.2 Fault-Tolerant Multiprocessor Structure

The Fault Tolerant Multiprocessor, FTMP, is a hardware redundant multiprocessor system designed as a prototype for use in an ultra-reliable avionics environment. The architecture is discussed in [Hopkins 78]. This section gives an overview describing the hardware structure, and fault tolerant features of FTMP. Also discussed is the dispatcher-scheduler strategy, comprehension of the strategy is needed to fully understand the executive level experiments.

2.2.1 FTMP Hardware

Figure 2-1 gives the software appearance of the FTMP system. Each virtual processor is a triad consisting of three synchronized processors[1] executing the same code independently and conducting a hardware vote on the results. Each processor contains a local PROM, used to hold frequently used executive code, and a local RAM to store working stacks, data, less frequently used executive code (such as self tests), and application task code. Code and data are paged into local memory from global memory as needed. The system's memory is triplex redundant. Data written into system memory is the voted result from each processor in the triad. Data read from system memory is the voted result of each memory module in the memory triad. The system bus is a quintuply redundant serial bus, with three active lines. Active elements are allowed to transmit on only one line; while the receiving unit votes on information transmitted on these lines. The error latches are registers used to hold voter disagreements until subsequent error processing. The I/O ports have system bus addresses and are used to communicate with the external environment (aircraft actuators and sensors, display terminal etc.).

The system is configured with active processor, memory and bus elements. On a failure of one of these elements, the active element is replaced with a spare; system integerity is maintained through the hardware voters. In the case of a processor failure with no spare processor available, the triad is retired with the non-failed units becoming spares. The workload of the retired processor triad is continued on the remaining triads. During normal processing, the active and spare elements are rotated to allow the detection of faults on all elements of the redundant system.

The implementation of the architecture maps the components into ten Line Replaceable Units, LRUs. Each LRU contains a processor element, a global memory element (16K words), a clock element, an I/O element and necessary bus interfaces. The system is dispatched with ten active LRUs. This allows the initial configuration to have: three processor triads, with one spare, two 16K word memory triads with two spares for each memory triad,[2] and four clock networks

[1] For clarity in this report, the term processor refers to a single processor in the system, whereas virtual processor, processor triad, or triad refers to a synchronized processor triple.

[2] Some early FTMP documentation [Draper 82] shows 48K, three memory triads, instead of two memory triads.

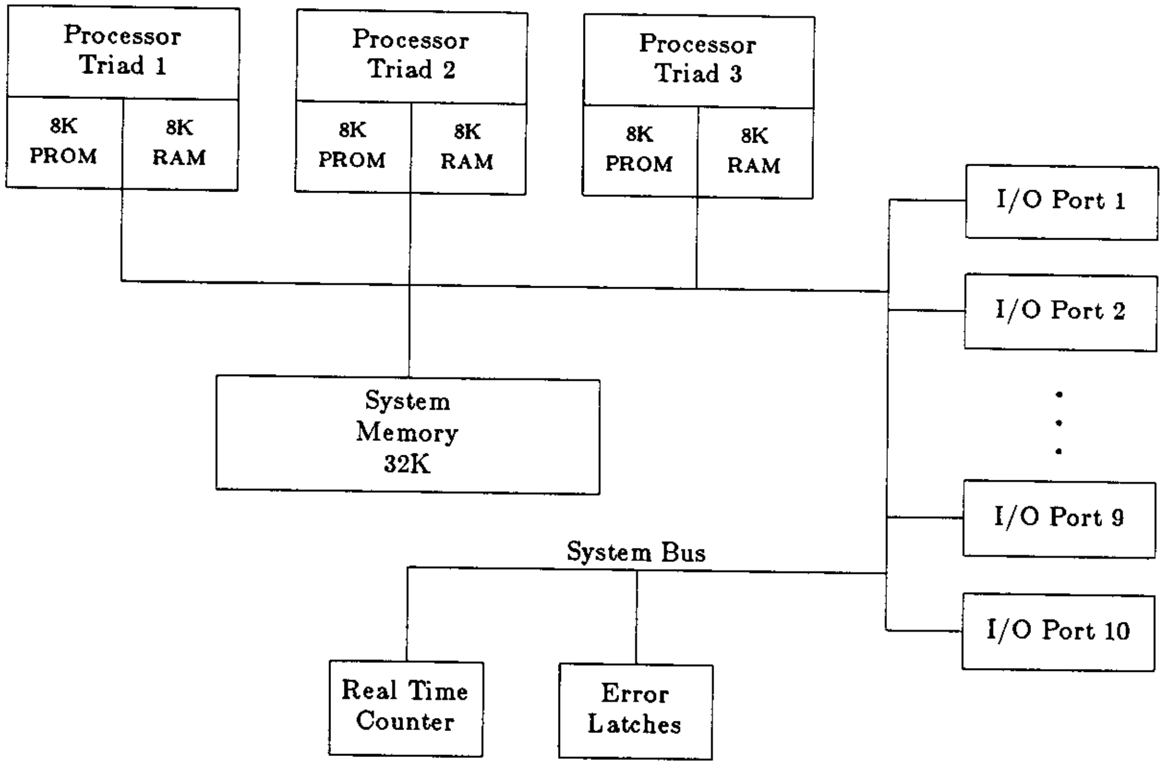

Figure 2-1: FTMP Structure, Programmers Model.

with six backups.[3] The implementation of a LRU is shown in Figure 2-2. As seen in the figure, each LRU is composed of a processor region containing the CPU and local memory and the slave region containing a global memory module, real time clock, status, control and communication registers and an I/O port. In each LRU there is a bus interface unit controlling the processor and slave regions' access to the system bus and hence the system configuration. Also shown in the figure is the system bus composed of four buses each quintuply redundant. The functions of the four buses are:

- Transmit Bus: Transmits address and data from the processor to system bus devices. One processor from each triad is assigned to a unique active bus. The information on the bus is address and data for a write from a processor to a system device, and is address only for a read from a system device.
- Receive Bus: Transfers data from system bus devices to the processor triad during a read by a processor triad.

[3] Failure of a clock in an LRU causes failure of all the LRU elements.

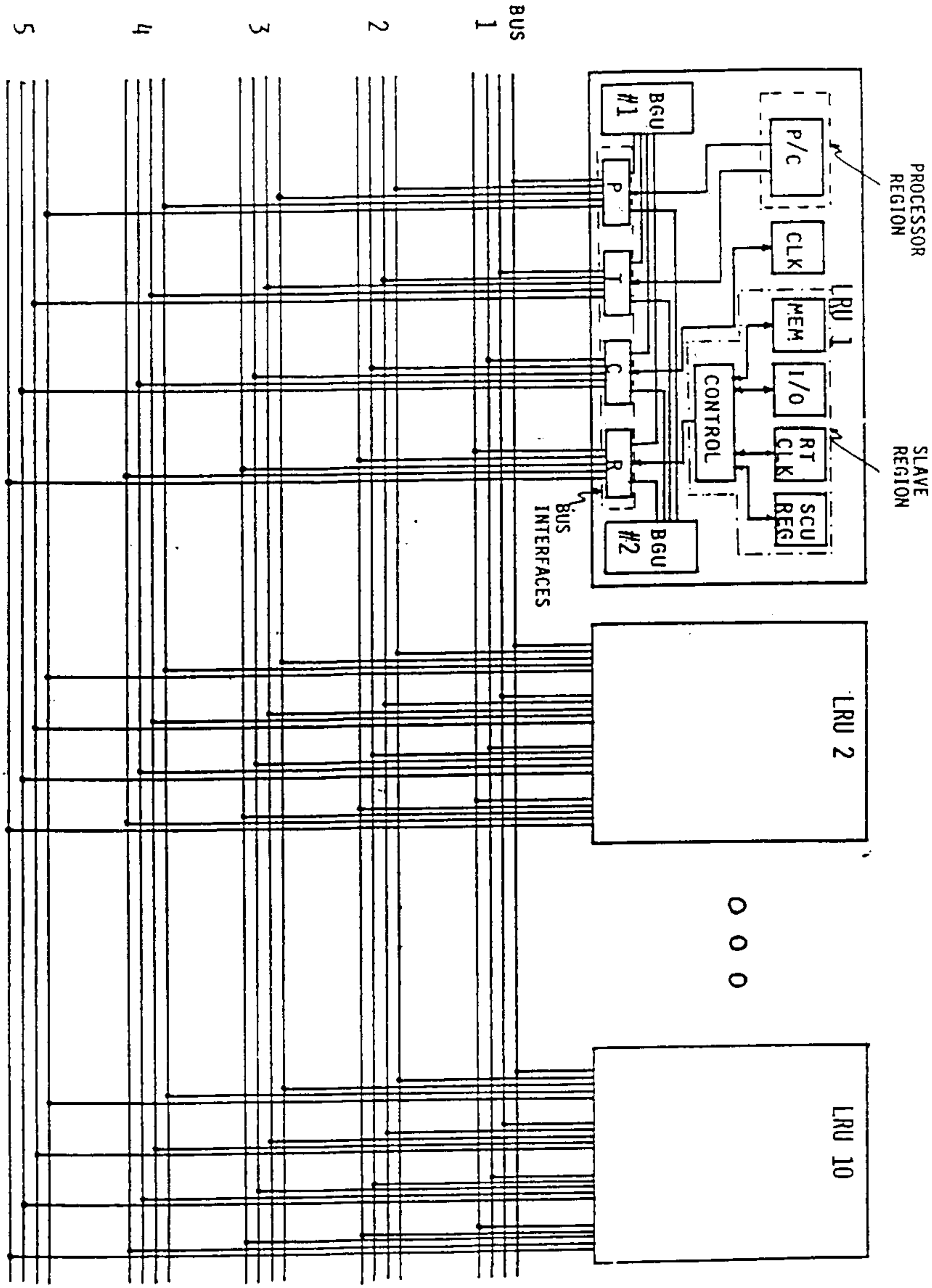

Figure 2-2: FTMP LRU and Bus Interface [Draper 82].

- Poll Bus: The system bus arbitrator. The arbitration scheme allows one triad to gain and keep control of the Poll and Transmit buses during a bus request.
- Clock Bus Quad: The clock distribution network keeps the system synchronized. Each LRU's clock is synchronized against the majority vote of the four active clock bus lines. Four LRUs are enabled to transmit their clock onto the four active buses.

A critical section of the system is the bus interface and bus guardian units. The bus interface unit protect system integerity by halting the propagation of errors beyond the bus/bus device boundary through a hardware vote. The Bus Guardian Units, BGUs, control the receive and transmit enables for a processor thus configuring the system. The BGUs are regulated by a triad write to the BGU's control register commanded by system configuration control software. The BGUs are potentially a single point of failure, therefore the BGUs are duplicated within each LRU. Consent of both BGUs is necessary before the element is allowed to transmit on the bus.

2.2.2 FTMP Scheduling and Dispatcher Strategy

FTMP was designed as a real time computer system intended to execute tasks at a fixed iteration rate or frequency to meet hard deadlines. This section covers the Scheduler-Dispatcher strategy used in FTMP to meet these deadlines. The strategy is presented in three parts, an overview and definition of the task frame structure, a uniprocessor multiprogrammed strategy, and finally a multiprocessor multiprogrammed strategy. [Draper 83a]

2.2.2.1 Overview

An FTMP task is a single "program" or thread of execution. Tasks in the real time world run at regular intervals called task iteration rates. All tasks need not be run at the same iteration rate, or frequency. (e.g. The status display does not have to be updated as often as an aircraft's control surface.) Tasks are grouped into common rate groups and are executed within a time period called a frame. FTMP has three iterations rates:

- R4: the fastest set at 25 Hz or 40 milliseconds per frame.
- R3: an intermediate rate set at 12.5 Hz or 80 milliseconds per frame.
- R1: the lowest priority, set at 3.125 Hz or 320 milliseconds per frame.

Figure 2-3 shows the time relations of the frame structure, including the 1:4:8 ratio between the frames. A major frame is the complete cycle of eight R4 frames, four R3 frames or one R1 frame.

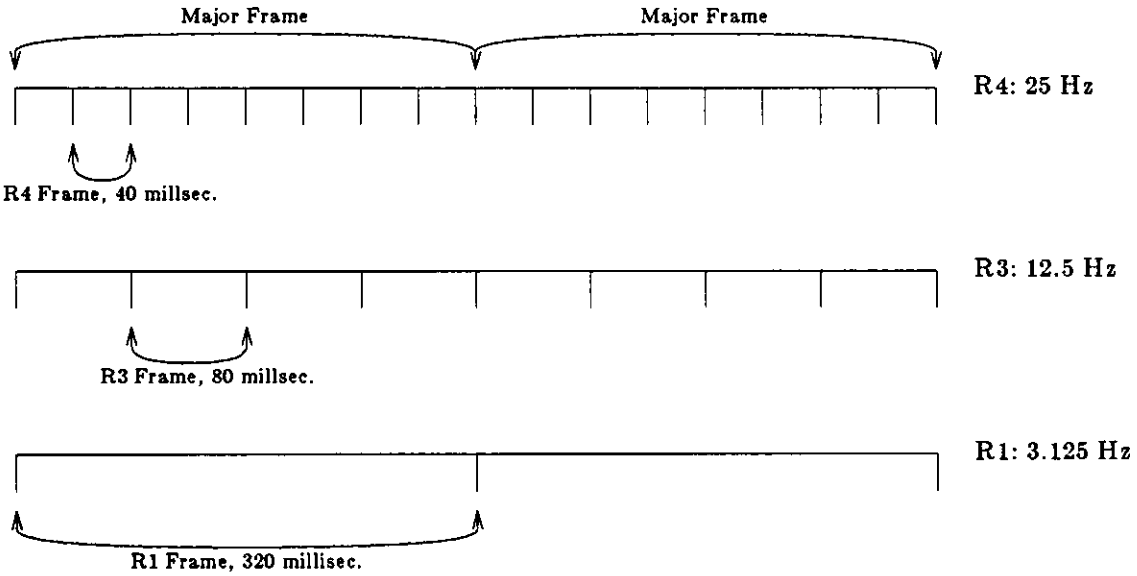

Figure 2-3: FTMP Task Frame Structure.

2.2.2.2 Uniprocessor-Multiprogrammed

The multiprogrammed uniprocessor real time environment, for the FTMP frame structure, dispatches and schedules tasks in the following sequence:

1. Assume an initial state where the processor is idling, waiting for an interrupt to start the frame.

2. A timer interrupt occurs and the interrupt handler starts the R4 dispatcher. This interrupt occurs at regular intervals to signal the start of the R4 frame.

3. The dispatcher does necessary housekeeping. In particular FTMP's dispatcher marks the lower iterations rates to start, does I/O for the tasks and issues reconfiguration commands. FTMP marks lower rate groups for execution by stringing together the Processor State Descriptor, PSD of the dispatchers.[4]

4. Once the dispatcher is finished with its housekeeping, it begins work on the first application task of the highest rate group, R4. The tasks to be executed are located in a task queue data base.

5. When this task is complete, control is passed back to the dispatcher. The dispatcher finds the next task to execute and the processor begins work on this task. This process continues until all the tasks in the rate group are completed.

[4] This can be thought of as a string of procedure calls or interrupts, where the returning location of any procedure may be changed dynamically.

6. At the completion of all the tasks in the primary rate group, R4, control is passed to either the next lower rate group dispatcher, a previously interrupted task or the idle state. In Figure 2-4 control is passed to the R3 dispatcher in the first frame. The control is passed by transferring control to the next task in the PSD chain.
7. The lower rate group, R3, dispatcher selects an application task from its task queue and starts execution of the task. The selection and execution of tasks continues until all tasks have been executed, then control is passed down the PSD chain to the lower priority dispatcher, R1 or an application task. If all tasks have completed for the frame, the idle process will execute[5].
8. At some point in this process a timer interrupt occurs signaling the start of the next R4 frame. The task executing, R3-Application task 2 in Figure 2-4, is suspended until the R4 (and possibly R3) tasks of this frame complete. The R4 dispatcher begin its execution, does housekeeping and then works on the R4 application tasks.
9. When all the R4 tasks are completed, controlled is passed to the previously interrupted task or pending task, such as the R3-application task 2 shown in Figure 2-4.
10. This process continues with each major frame.

Figure 2-4 summarizes the dispatcher scheduler strategy in a time diagram. A few potential problems arise in this strategy, as sketched below.

- The R4 tasks may not finish in an R4 frame because of temporary system overload or a long dispatcher execution time. FTMP avoids this problem by not arming the timer interrupt until the completion of all R4 tasks. This however causes problems because the frame could be slipped, or extended causing the system to be unable to meet its real time constraints.
- Hogging of the CPU cycles by the highest rate groups is another problem. FTMP addresses this issue by delaying, slipping, the start of the next R4 frame such that 10 milliseconds is set aside for the lower task groups. This again may cause system failure because of missed deadlines.
- When the workload is too large for the system, the lower rate group tasks do not complete before the scheduled start of their next frame. Before dispatching the lower rate group tasks, the scheduler checks if the previous frame's tasks have completed. If the tasks have not completed, the start of the frame, R3 or R1, is delayed one R4 frame, again risking the possibility of missed deadlines.

In summary the dispatch scheduler is satisfactory for normal operation, but for a high workload real time deadlines may be missed.

[5] The idle process is the last task in the PSD chain. It will never complete.

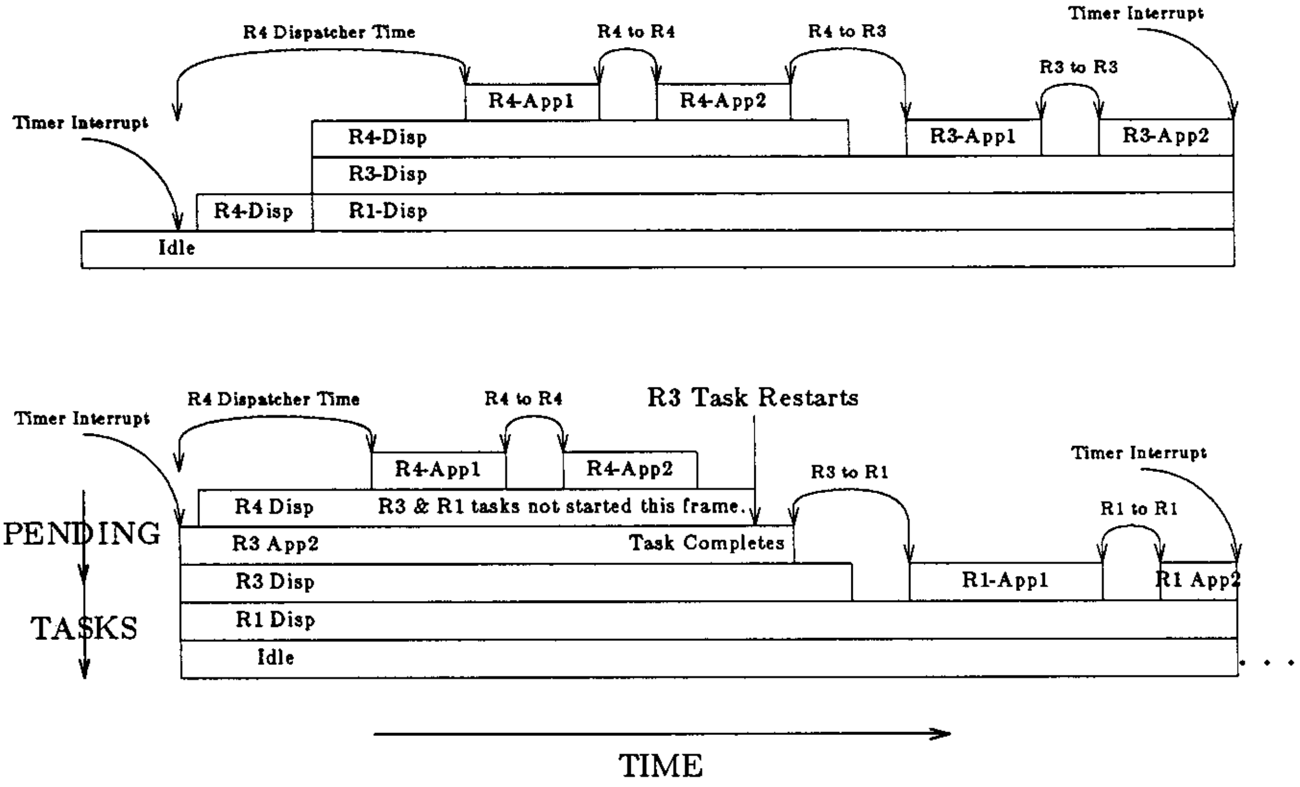

Figure 2-4: FTMP Dispatcher Scheduler Strategy, Showing Two Consecutive Frames.

2.2.2.3 Multiprocessor-Multiprogrammed Dispatcher Strategy

The uniprocessor multiprogrammed environment can be expanded into the multiprocessor domain with little modification. The necessary modifications to the uniprocessor strategy to the FTMP multiprocessor are the following:

- Only one triad's timer interrupt is armed to start the R4 frame. This is called the *R4 responsible* triad and is the last triad to finish work on the previous R4 frames.
- The R4 responsible triad during its dispatcher execution "Kicks", through an Interprocess Communication, the start of the R4 frame in the other processor triads.
- The global task queues must be locked by semaphores during critical sections of the dispatcher code. This prevents two triads from executing the same task.
- Frame are started and slipped in the same process as the uniprocessor case.

2.2.3 FTMP Fault Detection Software

Errors[6] are detected with the hardware majority voters of the bus interface units. An error is corrected by voting, to maintain system integerity, and the error is marked in the error latches for further processing. The processing of the error latches is done under software control by the task System Configuration Controller, SCC. This task runs under the lowest iteration rate, R1, and completes the following during its execution:

- Reads the error latches, tests for reasonability[7] , and compacts the error latches into four words, one for each bus quintuple, for further processing.
- If no errors are detected, SCC will rotate active and shadowing elements (processors, memories, and buses). The rotating of active and spares elements occurs once every ten seconds. Or if the elements are not to be rotated, self tests are executed to expose latent faults in the voters, bus guardian units, and buses.
- If errors were detected from the error latches fault isolation occurs. The possible source(s) of errors are determined, and isolated by swapping with shadowing units. For the next four iterations the program remains in this state to discover if an error reoccurs.
    - If the error reoccurs and the source can be determined from the past error(s), the faulty unit is retired and a spare brought on-line.
    - If the error does not reoccur, a transient error routine is entered to assign demerits to all possible faulty units. If the total of demerits for a unit cross a threshold the unit is retired.

2.3 Experimental Environment

Figure 2-5 shows the experimental (test) environment for FTMP. The following steps must be taken to create and run experimental tasks on FTMP. The application task code is created on the VAX and shipped to the IBM for compilation. The IBM returns to the VAX a listing of error and assembly code. The object code is kept on the IBM for linking at a later time. The system memory tables are modified to include the application code in the task queue, and allocate global memory for the task. The tables are assembled on the IBM in the same method as the application task. A link file is then sent to the IBM for linking together the executive routines, application tasks, and system memory tables. A listing of global variable locations,

[6] An Error is the manifestation of a fault that causes a change in the data, whereas a fault is any deviation from the intended design. Faults can be classified as hard, permanent faults or soft, transient faults, either type of fault may or may not cause an error.

[7] A receive bus line may be faulty, causing either an error latch to be set or an error in the reading of the error latch or both.

task code locations and errors is sent down from the IBM along with the load module for FTMP. The load module is down-loaded to FTMP via the PDP-11 emulation on the VAX and the test adapter. The experiment is then debugged using the test adapter. Once the experiment is debugged, the test adapter is used to set flags and iterations values in the experimental tasks, and to dump data from FTMP's system memory to the VAX for further analysis.

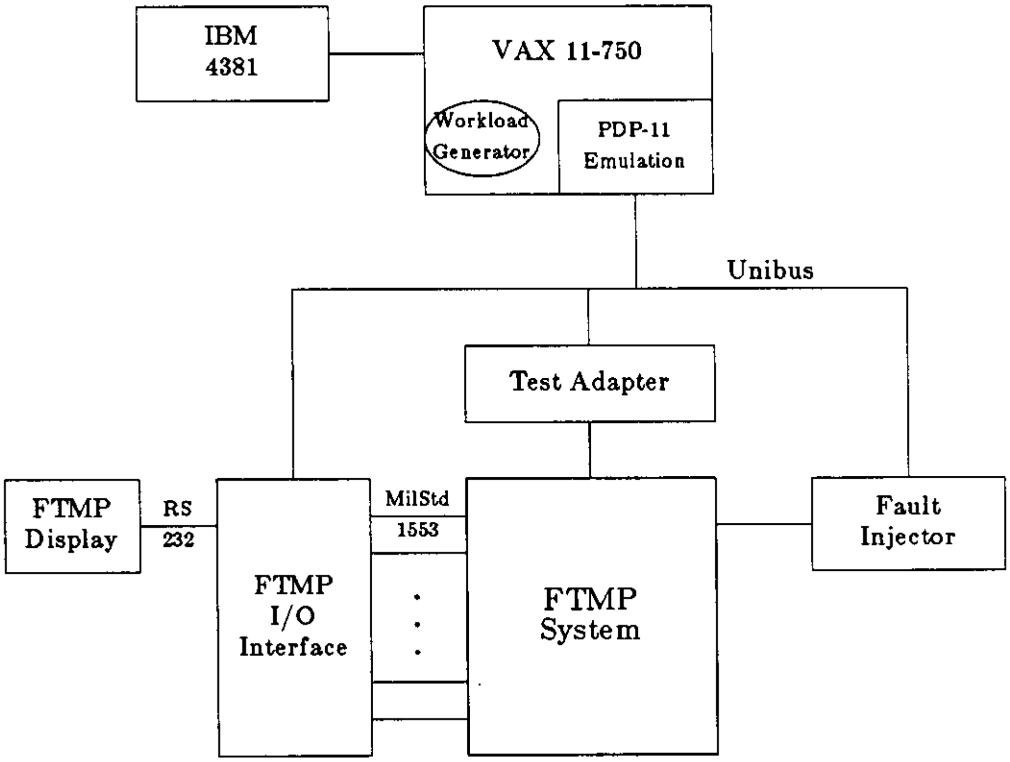

Figure 2-5: FTMP Experimental Environment.

In an effort to shorten the experimental turn around time a synthetic workload generator was proposed by [Clune 84] and developed by [Feather 85]. A synthetic workload is a set of programs designed to exercise a computer system to check its performance and behavior under artificial conditions. The actual environment that the computer operates in is called its natural workload. Some of the advantages to using a synthetic workload over a natural workload are:

- The synthetic workload is easy to create and debug, whereas a natural workload may have to be created and its set of inputs defined.
- Experiments are easily repeatable, corresponding to the experimental design philosophies.

- Experiments are easily controlled using the workload parameters.
- The workload can be adapted to other systems for performance comparisons.

Conversely, disadvantages to using a synthetic workload are:

- The system must be dedicated when using a synthetic workload, whereas with a natural workload data can be collected while useful work is being done.
- The synthetic workload is only an approximation of the natural workload.

A natural task includes a mixture of the following five actions.

1. Read Sensor data.
2. Read Inter-process Communication (IPC) data.
3. Operate on the Sensor and IPC data.
4. Write Actuator Commands.
5. Write IPC Commands.

The synthetic model of a single natural task for FTMP is illustrated in Figure 2-6. Loops represent the amount of work each of the five actions is to perform in the task. The controllable parameters are thus the loop counters. The counters are configured during experimental setup. In FTMP's implementation the real time clock is read at the start of the task, and at the end of each of the actions. The clock times are then stored in system memory for transfer to the VAX.

At the application level, there is more than one task on a multiprocessor. The performance, behavior and interaction of the tasks can be modeled by combining several single synthetic tasks. At the application level the system's synthetic workload parameters include:

- The number of tasks and their frequency of execution (FTMP's real time frame structure). Each task parameter is individually controllable.
- The number of triads executing on FTMP.
- The inclusion or exclusion of system executive tasks, such as Display and Configuration Controller.

2.4 Background and Previews to Experiments

At present, the following baseline experiments have been conducted on FTMP [Clune 84, Feather 85].

- Characterize the delay and variation in reading the real time clock. Characteristics of the real time clock are fundamental to the interpretations of other experiments.
- Measure the execution time of high level language instructions.
- Duration and variation of scheduling frames which impose real time deadlines on the system.
- Mechanism for extending the basic scheduling deadline.
- Responses to interrupts and exceptions.

This report covers FTMP experiments in the following areas.

```
Workload_Task() ;
Begin
        Read(P, Q, T, R, S ) ;
        Read(Time) ;
        For X = 1 to P  do
                Read_Sensor_Input ;       (Read Memory)
        Read(Time) ;
        For X = 1 to Q do
                Read IPC Data ;           (Read Memory)
        Read(Time) ;
        For X = 1 to T do
                Execute_Instructions ;    (A = B + C )
        Read(Time) ;
        For X = 1 to R do
                Write_Actuator_Command ;(Write Memory)
        Read(Time)
        For X = 1 to S do
                Write_IPC_Command ;       (Write Memory)
        Read(Time)
        Store(Clock_Times) ;
End;
```

Figure 2-6: Representation of a Synthetic Workload Task.

- Execution time of high level language instructions.[8]
- Execution time of instructions in combinations to see if the result is equivalent to the sum of the execution times when measure singlely.
- Time to transfer blocks of data to and from system and local memory.
- Characterize the dispatcher execution time from frame start to the execution of the first application task. Dispatcher execution time is marked in Figure 2-4.
- Characterize the IPC 'kick' used to start other triads working on the R4 frame.
- Task switching times between tasks of the same and different rate group. The task switching times are shown in Figure 2-4.
- Overhead and variation in fault detection and system configuration software.

The experiments in this report cover both baseline and executive level areas. The baseline experimental tasks were developed singularly for the experiment. The executive level experiments were conducted using the synthetic workload. The workload allowed task interaction times to be measured, with a variable amount of tasks, different task durations, and different FTMP configurations. All times measurements were taken from the real time clock on FTMP.

[8] [Clune 84] measured instructions using static local variables. This experiment uses the more efficient dynamic stack local variables.

3. FTMP Experiments

3.1 Instruction Execution Time Measurements

One element of the Performance Evaluation Matrix of Figure 2-1, is the throughput of hardware primitives. Instruction execution times and access times of hardware resources (Clock reads, bus access and transfers, etc.). were measured. Section 3.1 summizes the experiments used to measure the execution time of High Level Language (HLL) instructions.

3.1.1 Experimental Setup

Since the resolution of the real time clock (250 μ-seconds) is too course for measuring the execution time of a single instruction, the measurements are performed in an indirect fashion. The instruction under test is repeated 1000 times in a loop and the full loop execution time is measured. The loop and clock read overheads are then subtracted, yielding the execution time for 1000 iterations of the instruction under test. From the resultant times the execution time of a single instruction, is calculated as well as standard deviation and confidence intervals. A typical FTMP experimental task is shown in Figure 3-1.

```
Begin
        EXEC = READ( MM.CMU.EXEC );                 /* MM defines Main Memory       */
        If EXEC <=  some_count
        Begin
                ITCNT = READ( MM.CMU.ITCNT ) ;
                HOLD1 = READ( RT.CLOCK ) ;
                For I = 1 to ITCNT DO               /* Repeat the loop ITCNT times  */
                        Begin
                        some_instruction
                        End;
                HOLD2 = READ( RT.CLOCK ) ;
                WRITE(HOLD1, MM.CMU.TIMER1(EXEC) ) ;        /* Write Clock Times    */
                WRITE(HOLD2, MM.CMU.TIMER2(EXEC) ) ;        /*  into Main Memory.   */
                EXEC = EXEC + 1 ;                   /* Increment Flag               */
                WRITE(EXEC, MM.CMU.EXEC ) ;
        End;
End.
```

Figure 3-1: FTMP Experiment Task Algorithm.

The experimental procedure is as follows. A global memory flag is set using the test adapter, CTA. When the experimental task finds this flag set[9] the experiment is run. The clock

[9] Remember this is a real time computer executing the task at 25 Hz.

times are then stored in global memory array and the flag incremented (so not to overwrite the data). The test adapter reads the clock values after waiting a specific amount of time. The process is then repeated (under control of a command procedure) to obtain multiply data points. A data point is the complete execution of the task in Figure 3-1. Each instruction under test was run with this procedure to obtain 500 data points; the tasks were executed with only one FTMP triad running to rule out possibilities of bus contention.

3.1.2 Experimental Results

Figure 3-2 summarizes the results for representative HLL instructions involving 16 bit integers, 32 bit integers, and 16 bit fixed point.[10] Appendix I presents a tabular form of all the instructions tested. For the instructions tested, data points were within one clock tick, thus showing consistent results for the execution times of instructions. (This is reflected in the tight 95% confidence intervals given in Appendix I).

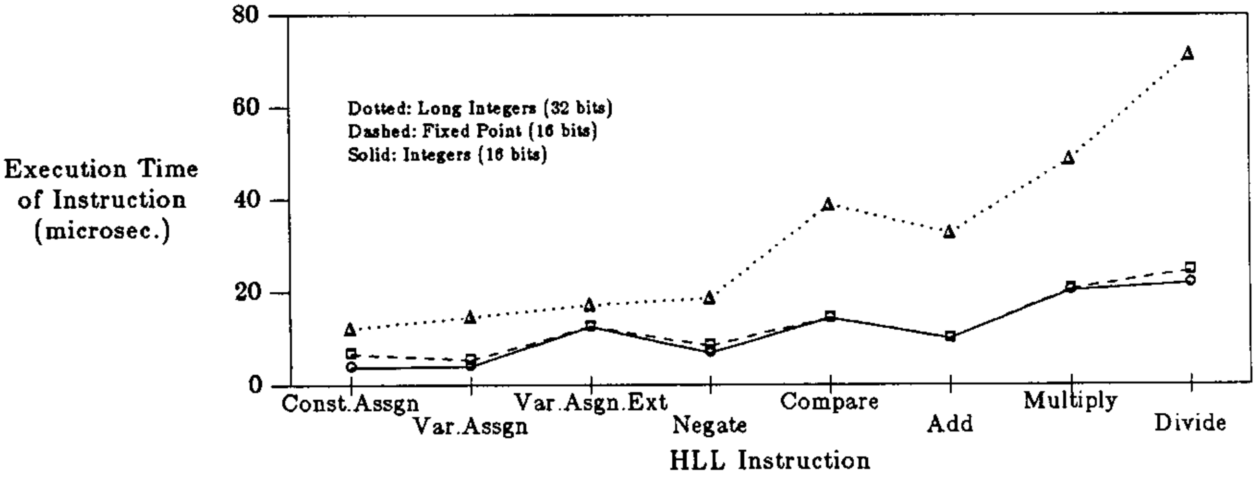

Figure 3-2: AED Instruction Execution Time Summary.

Some other issues discovered during the setup and execution of the experiment include:

- The compiler generates code for 3 different types of assembly loops, depending on loop contents and semi-colon placement.
- Loop instructions starting on a byte address have a longer execution time (+1.5 μsec/iteration) than those starting on word addresses.

[10] FTMP's fixed point definition is a 16 bit quantity with a assumed binary point before the most significant bit, also referred to a 16 bit fractional. There is no hardware support for floating point operations although the compiler provides software emulation of floating point operations. Only fractional operations were tested.

These behaviors demonstrates thorough testing is necessary to discover system and implementation dependencies.

3.1.3 Instruction Combinations Executions Times

The previous experiment measured the execution times of single representative HLL instructions for predicting the throughput of a processor. The next question to answer is whether the execution time of instructions in combination is the sum of the single execution times. This questions determines if the machine is predictable as well as searching for any compiler optimizations.

The experiment was set up in the same fashion as the single instruction measurements, with an instruction pair substituted for the single instruction. Each combination was executed 1000 times in a loop on a single processor triad. All data was within one clock tick; the results are summized in Table 3-1.

Instruction Pairs Execution Times Summary
(All times in micro-seconds, Ranges are 95% Confidence Intervals)

Instruction Pair	First Experiment Predicted Time	Single Loop Execution Time	Pair Execution Time	Percent Difference
B = 2; C = 2	8.0 ± .16	31.7 ± .30	8.0 ± .22	0
B = 2; C = 3	8.0 ± .16	31.7 ± .30	8.0 ± .22	0
B = 2; C = B	9.5 ± .15	30.2 ± .31	6.5 ± .22*	-31.5
B = 2; C = D	9.5 ± .15	33.2 ± .29	9.5 ± .21	0
B = 2; B = 3	8.0 ± .16	31.7 ± .30	8.0 ± .22	0
B = B	5.5 ± .21	29.2 ± .31	5.5 ± .22	0
B = C; D = C	11.0 ± .15	34.7 ± .31	11.0 ± .22	0
B = C; D = E	11.0 ± .15	34.7 ± .31	11.0 ± .22	0
B = C+D; E = B+F	20.0 ± .15+	43.2 ± .31	19.5 ± .22*	-2.5
B = C+D; E = F+B	20.0 ± .15+	44.7 ± .32	21.0 ± .22**	+5.0
B = C+D; E = C+D	20.0 ± .15	43.7 ± .30	20.0 ± .22	0.0
B = C+D; E = F+A	20.0 ± .15+	44.7 ± .32	21.0 ± .22**	+5.0
B = (C+D)+(C+D)	-	41.2 ± .32	17.5 ± .22	-
B = (C+D)+(E+A)	-	41.2 ± .31	17.5 ± .22	-

Table 3-1: Instruction Combination Execution and Predicted Times.

As seen in Table 3-1 the instruction execution times add as predicted, with two exceptions. The first exception (marked by a single asterisk), deals with compiler optimization of the assembly code. The compiler optimizes by replacing a store followed by a push of the same variable to a duplicate (top item on stack) followed by the store. The second exception (two asterisks) deals with the pairs containing a reference to the variable "F". The assembly code is the same except for the reference to "F", this reference required an extended reference because the allocation of variables on the stack placed the location of "F" beyond the simple reference limit.[11] Thus from this data it is safe to conclude the execution time of instructions in combination is the sum of the single execution times.

3.2 Block Transfer Times

All FTMP tasks require access to shared system memory or other common system resources. Access to these resources is achieved using System Executive Primitives. Executive primitives are basic functions used repeatedly by most user and executive tasks. FTMP divides these into four categories. [Draper 83a]

- System Bus Service Routines: These procedures are used to read to and write from devices on the system bus (except error latches). The read and write routines involve simplex and voted, high and low address space, and non-incremental and incremental transfers. Also included under this category are hog and release bus, and synchronization primitives.

- Error Latch Service Routines: These are specific bus service routines optimized for the error latches. Error latches are registers containing voter disagreement data.

- Timer Routines: These keep track of several interval timers in software. There is only one hardware timer per processor.

- Miscellaneous Primitives: Include lock and unlock semaphores, test and set routines, and IPC (Inter-process Communication) Kick function used to start R4 tasks in another processor triad.

3.2.1 Experiment Set-Up

This experiment considered two of the system bus service routines: RD and WRT. The functions are voted, low address, incremental read and write. These experiments were set-up similarly as the instruction execution time measurements; the differences being in a reduced iteration count while varying the amount of words read or written. The iteration count was lowered to 50 to reduce the total execution time of the task. The size of the block transfers was

[11] A reference to a variable is from the local environment pointer. 16 words may be referenced by a byte instruction, (a 4 bit instruction and a 4 bit operand). An extended reference is for variables further than 16 words from the environment pointer these requires two bytes of instruction, (8 bit instruction and 8 bit operand).

varied from 1 to 200 words[12] (1 word = 16 bits) The number of processor triads competing for the system bus was also varied. One triad showing execution time with no contention for the bus, and two triads running to show the effects of bus contention.

3.2.2 Results of Block Transfers

Figures 3-3, 3-4, and 3-5 show the execution times of reads and writes to be linear with respect to the size of the data block being transferred. The figures also show an increase in average execution time due to bus contention. This increase is nearly constant throughout the range of block sizes. (Tabulated data can be found in Appendix II.)

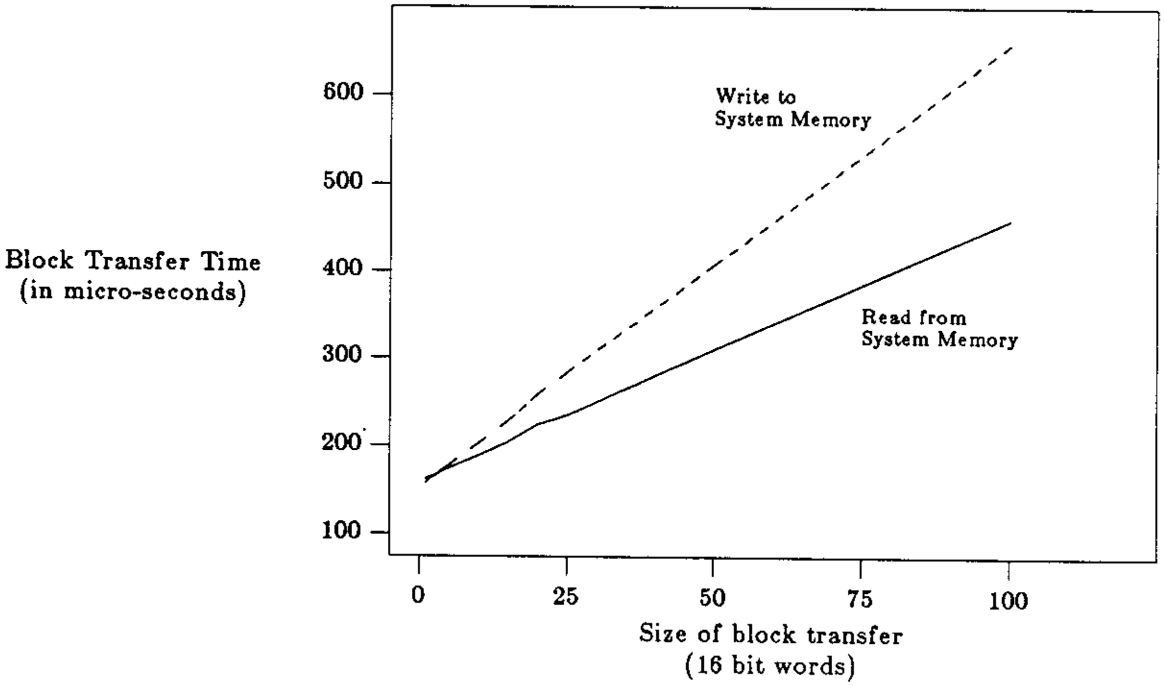

Figure 3-3: Read and Write Execution Times as a Function of Block Size, 1 Triad.

Of interest is the spread of data in the one and two triad case. With one triad, no bus contention, the measured times were within two clock ticks. (The sum of clock ticks to complete a task varied from 31, for a one word write, to 248, for a 200 word write.) With two triads competing for the bus, most of the measured times were within eight clock ticks, with a few

[12] A full page, 256 words, could not be transferred because of the size of the working stack for an application task.

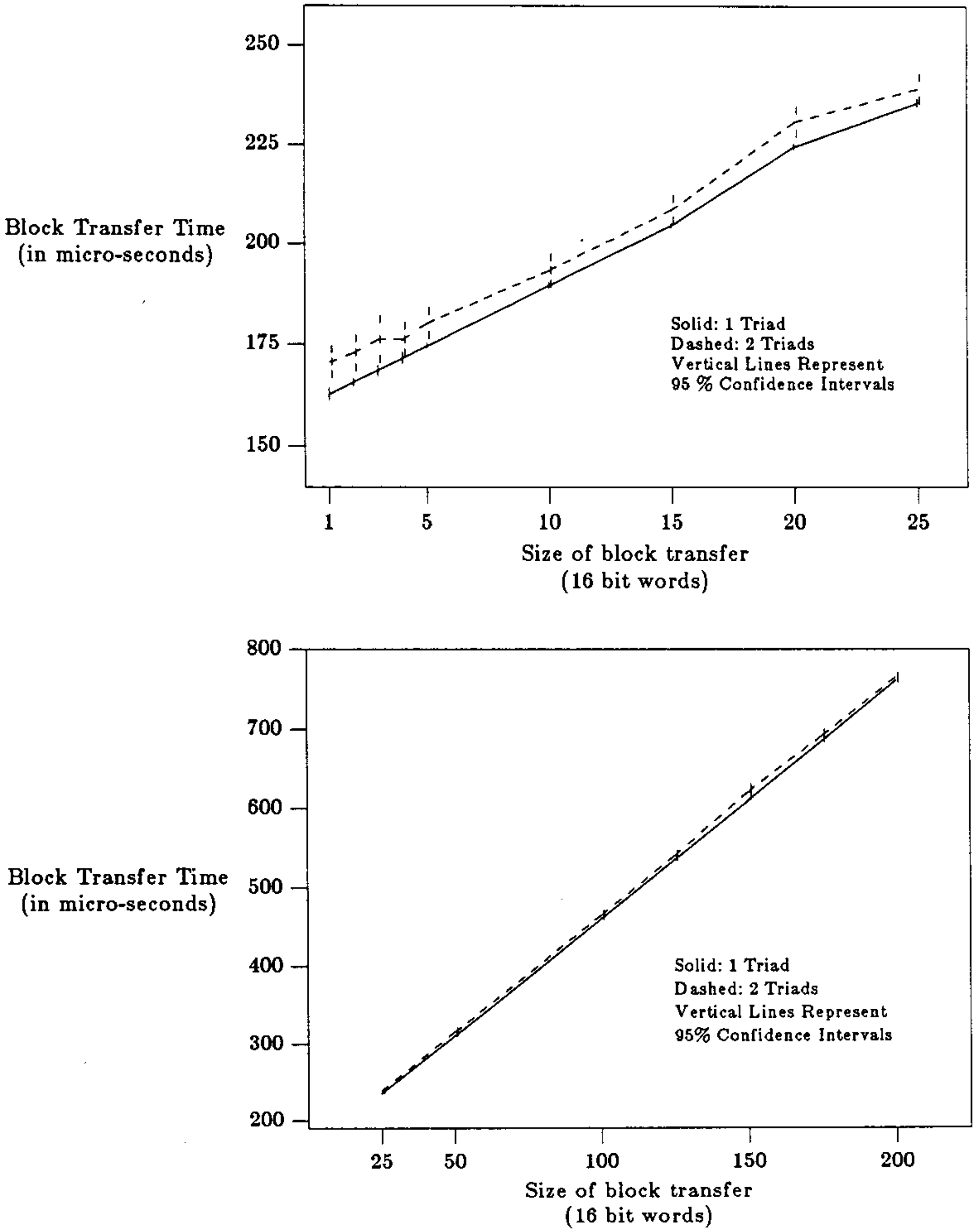

Figure 3-4: Read Execution Times as a Function of Block Size, 1 Triad and 2 Triads.

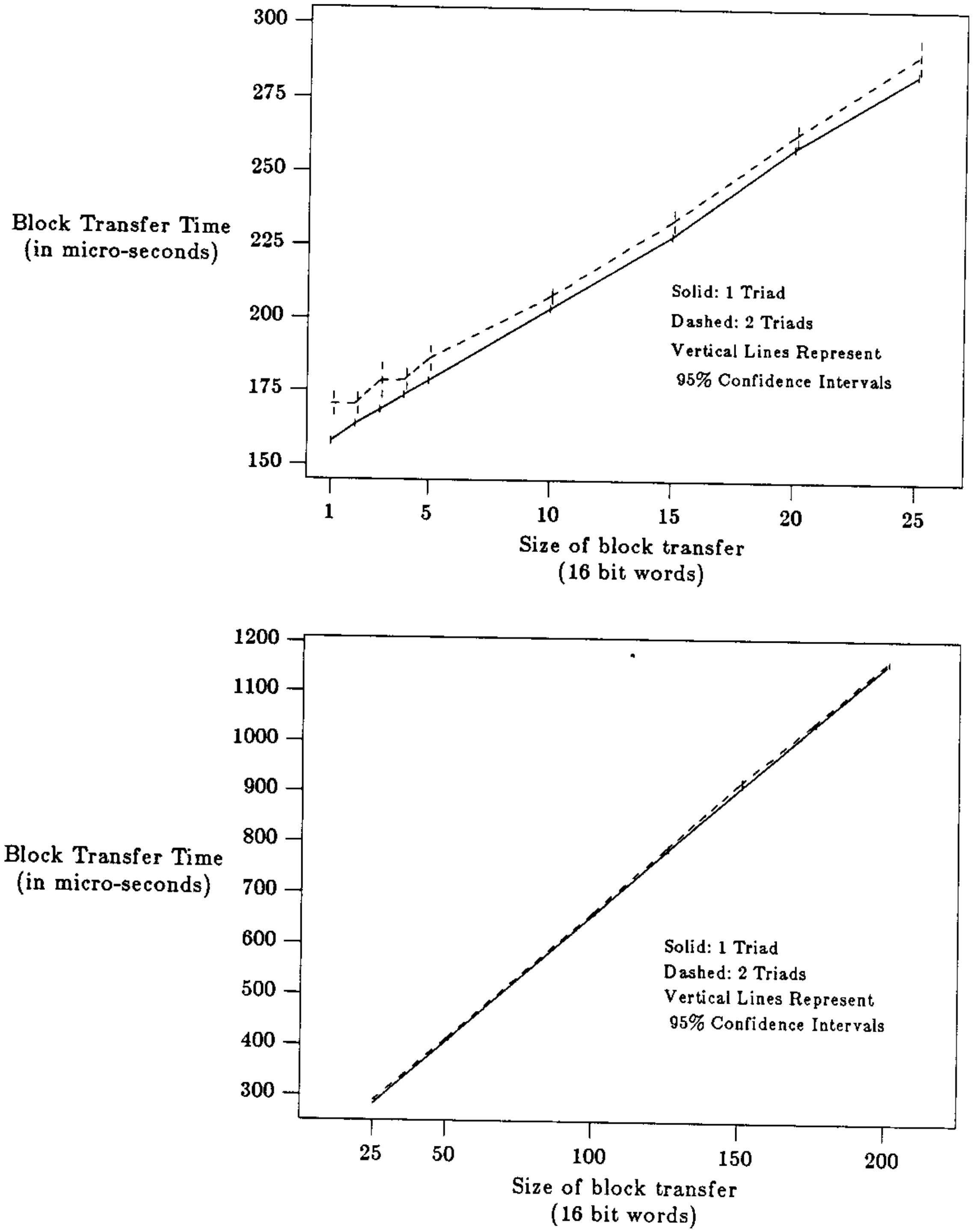

Figure 3-5: Write Execution Times as a Function of Block Size, 1 Triad and 2 Triads.

outlayers up to 20 clock ticks (5 msec) from the mean. This is illustrated in Figures 3-4 and 3-5, showing the increase in the 95% confidence intervals and in Figure 3-6 showing the distribution of a typical two triad read.

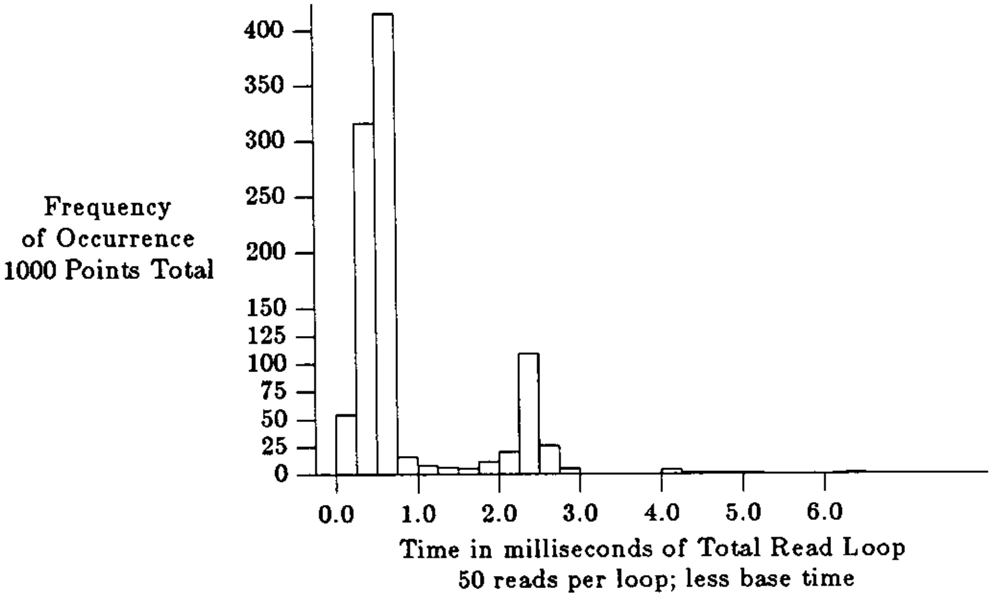

Figure 3-6: Typical Distribution for a Bus Service Routine with Competing Triads.

A final word regarding system bus service routines comes from the FTMP Executive Summary [Draper 82]. In the summary, the authors noted the bottleneck caused by system bus access (greater than 150 μsec overhead per access which equals about 15 HLL instructions) lowered the expected throughput of the system. [Draper 82] showed that one third of the dispatcher time is spent in overhead for bus service routines. This bottleneck could be reduced by microcoding of some of the I/O functions.

3.3 Dispatcher-Scheduler Software Overhead

The behavior and delay of the executive software are two more elements of the Evaluation Matrix of Table 2-1, is the behavior and delay of the executive software. This section covers the Dispatcher-Scheduler software overhead. The dispatcher-scheduler strategy was described in Section 2.2. The following experiments measured the software overhead of the dispatcher-scheduler tasks, (see Figure 2-4).

- R4 dispatcher start up times,
- IPC 'Kick' times,

- Intra-task group switching times, and
- Inter-task group switching times.

3.3.1 Start-up Dispatcher Time

In the dispatch strategy, a timer interrupt occurs in one triad to signal the start of an R4 frame. The interrupt handler initiates the R4 dispatcher; the dispatcher schedules pending R3 and R1 tasks[13] , kicks the other triads to start R4 tasks, and does necessary I/O. At some point the dispatcher activates[14] the application task. The behavior and execution time of the dispatcher from the interrupt to the start of the first application task was measured.

The synthetic workload generator provides a means to measure execution times and intertasks times of applications tasks, but does not provide a means to measure the behavior of the R4 dispatcher, hence a custom task was created. This task is the first R4 application task; it reads the real time clock at the start of its execution and the absolute time of the frame start stored in system memory (to determine the time of the interrupt). The task was repeated 256 times dumping the clock values into system memory for transfer to the VAX. This experiment was repeated for one, two, and three processor triads running.

Histograms of the results is given in Figure 3-7. As seen in from these graphs the data is grouped into three regions. The execution times vary in a cyclic pattern of 65, 14, 16, 67, ... milliseconds for one triad executing, and 65, 15, 15, 40, ... for two and three triads. This pattern is probably caused by the scheduling of the R3 and R1 frames, and possibly I/O servicing for the tasks and reconfiguring commands issued by the configuration controller, but executed by the dispatcher. Further study of the dispatcher will be necessary to fully characterize its behavior.

This behavior is unacceptable for a real time system. Assuming the frame start interrupt occurs at a constant rate, the actual time between consecutive application task starts will vary plus or minus 50 milliseconds from the intended starts. This may caused missed or late data causing the real time system to miss deadlines. (Data on the variation between frame starts and

[13] The dispatcher schedules the start of a minor frame, R3 or R1, by appending the processor state descriptor of the rate group dispatcher behind the active processor state descriptor, the R4 dispatcher.

[14] The term activate refers to a process where the dispatcher transfers control to another process. At the process completion control is returned to the dispatcher.

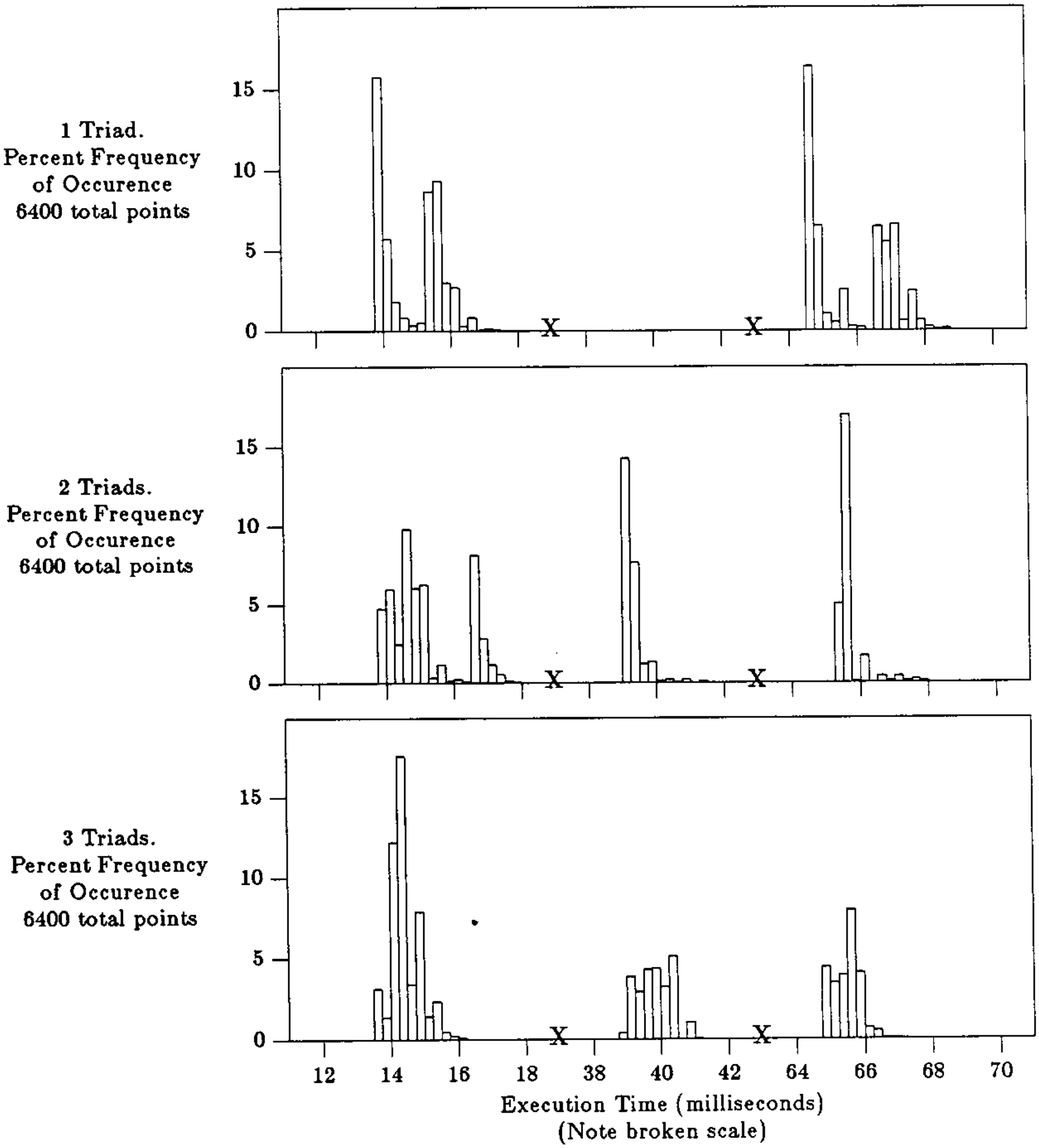

Figure 3-7: R4 Dispatcher Execution Times, 1, 2, and 3 Triads.

variation between tasks starts will be presented in Section 3.3.5.) The dispatcher takes a full R4 frame or longer to execute in one-half of the time, clearly an implementation problem.

Demonstrating any possible affects of system workload, and to show the frame start interrupt occurs at its set time, the experiment was repeated under the following conditions.

1. Decrease the work load to just the single application task being executed. The results show the same pattern and spread as the other tests.

2. Extend the frame size from 40 milliseconds to 250 milliseconds. This allowed all tasks to complete in a single R4 frame without slippage. Again the dispatcher behaved in the similarly. It was also noted that the frame starts occurred at 250 msec. with no variation[15] (i.e. the interrupt mechanism works correctly).

3.3.2 IPC 'Kick' Times

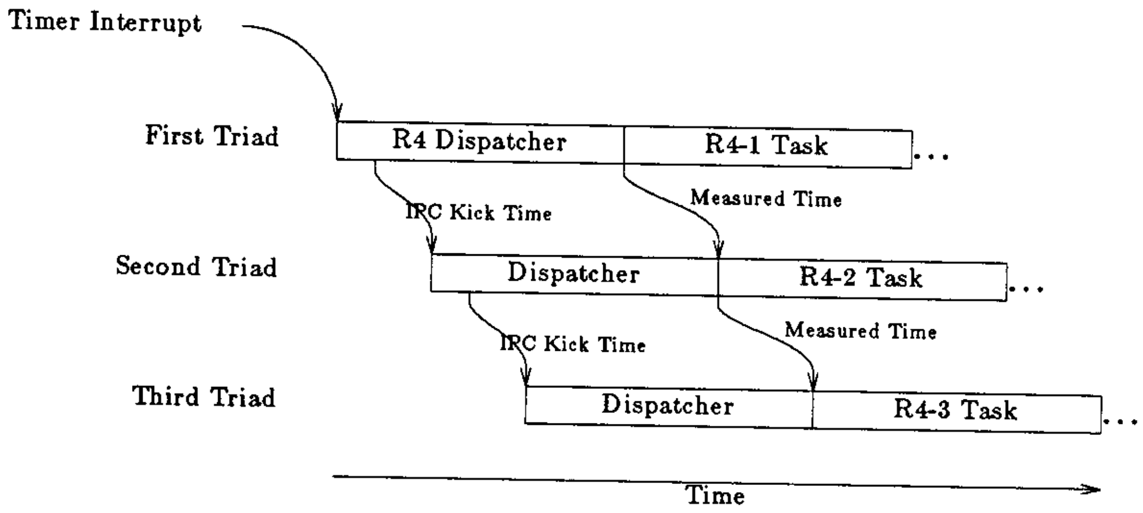

Figure 3-8: IPC 'Kicks' Timing Diagram.

One function of the R4 dispatcher is to 'Kick', through an IPC (Inter-Process Communication) interrupt, the start of the R4 frame in another triad. A timing diagram of this process is given in Figure 3-8. Again the workload generator does not allow the measurements of the time from the 'Kick' to the start of the application task; but the timing and behavior can be

[15] There was variation between frame starts at the 40 millisecond size. This variation is caused by frame slippage; the frame is extended 10 milliseconds past the last R4 task completion time. If the dispatcher takes 45 milliseconds the next frame cannot be started at the 40 millisecond mark.

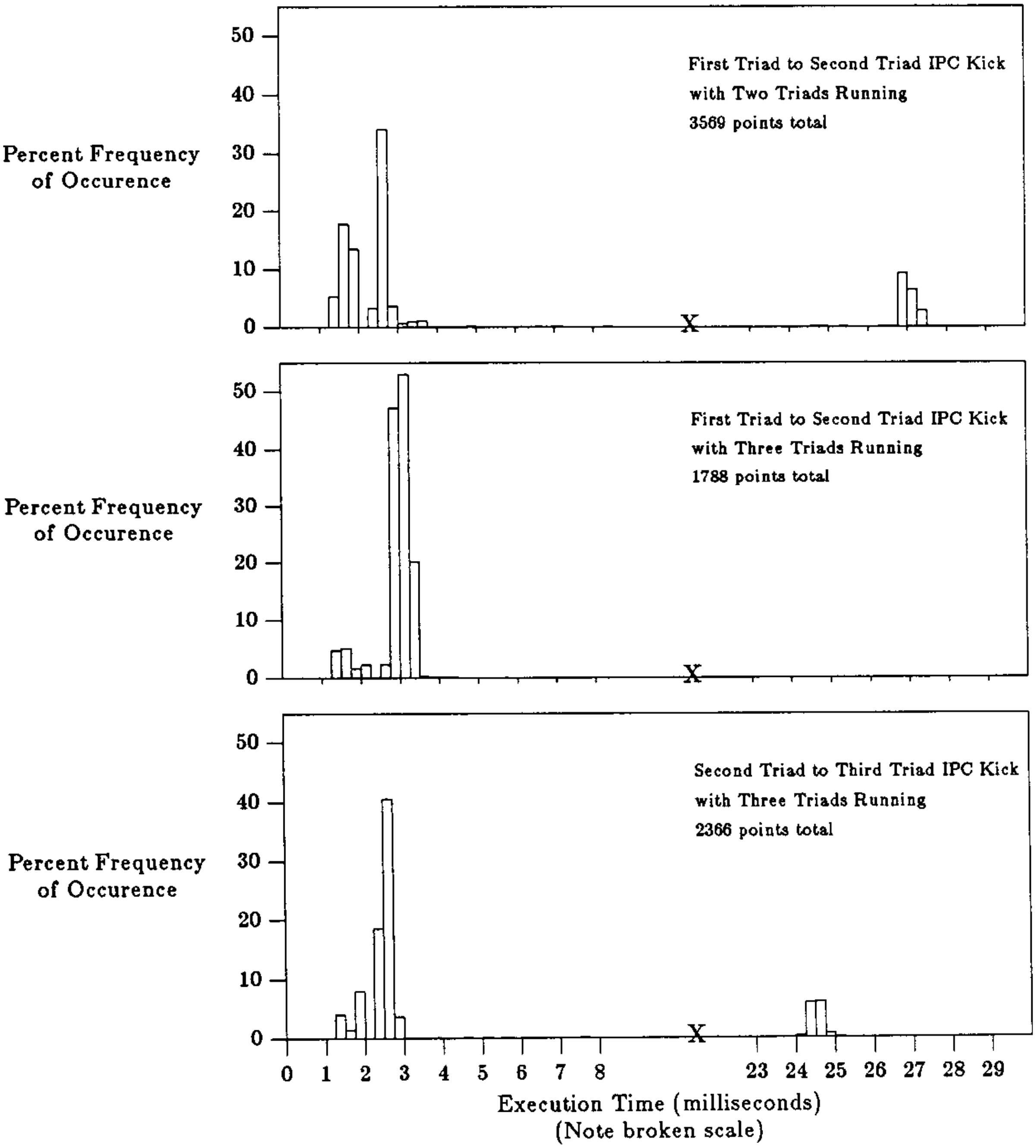

Figure 3-9: Time and Variation Between the Starts of the Application Tasks on Different Processor Triads. Emulating IPC Kick Times.

approximated by measuring the difference between the starts of the application tasks. The desired time to measure would be from the IPC kick of the first triad to the time when the R4 dispatcher begins execution, or from the frame start interrupt to the time when the second and third triad start their R4 dispatcher. An approximation to this behavior is the time between starts of the application task on different triads, labeled as "Measured Time" in the Figure 3-8

Histograms for the IPC 'Kicks' are given in Figure 3-9 with two triads executing, the times are grouped around 1.5, 2.5 and 27 milliseconds. With three triads executing the first to second triad "kick" is centered at 4.0 milliseconds with no outlayers beyond 5 milliseconds. However in the second to third triad "kick" there was a large group, about 10% at 24.5 milliseconds. This behavior is undesirable in real time systems. The long 'Kick' times allow the possibility of one triad running its first task, finishing, and then running a second task while the second triad is still idle. This behavior was mentioned in [Clune 84] and its cause should be investigated. Upon inspection of the dispatcher code, a few possibilities for this unwanted behavior arise. These possibilities include: a problem with the IPC 'Kick' mechanism, the possiblily of the dispatcher 'hanging' because of a lock or failed bus access, or an extended execution time during a system reconfiguration. Note that the outlayers involve between 10 and 18 percent of the data or about once per major frame.

3.3.3 Intra-Task Group Switching

At the end of the application task, control is passed back to the dispatcher. The dispatcher activates the next task in the queue (same rate group) or if all tasks have been dispatched, the dispatcher returns control to the previously interrupted or pending task (R3 or R1 dispatcher or application tasks). This experiment measures the intra-task group switching times including R4 to R4, R3 to R3 and R1 to R1 shown in Figure 2-4.

This experiment was run using the synthetic workload as a tool, and the measurements were taken for one, two, and three triads for all rate groups. The data is summarized in Figure 3-10.

As seen from the data the behavior is regular, with skewing as the number of executing triads increase. This skewing is probably caused by bus access contention, measured in Section 3.2. The large spread of the data in the R1 task switching could be explained by the difference in the experimental acquisition of the data. To measure the switching times, the tasks were ordered by controlling the task lengths. To have one triad execute two tasks from the same rate

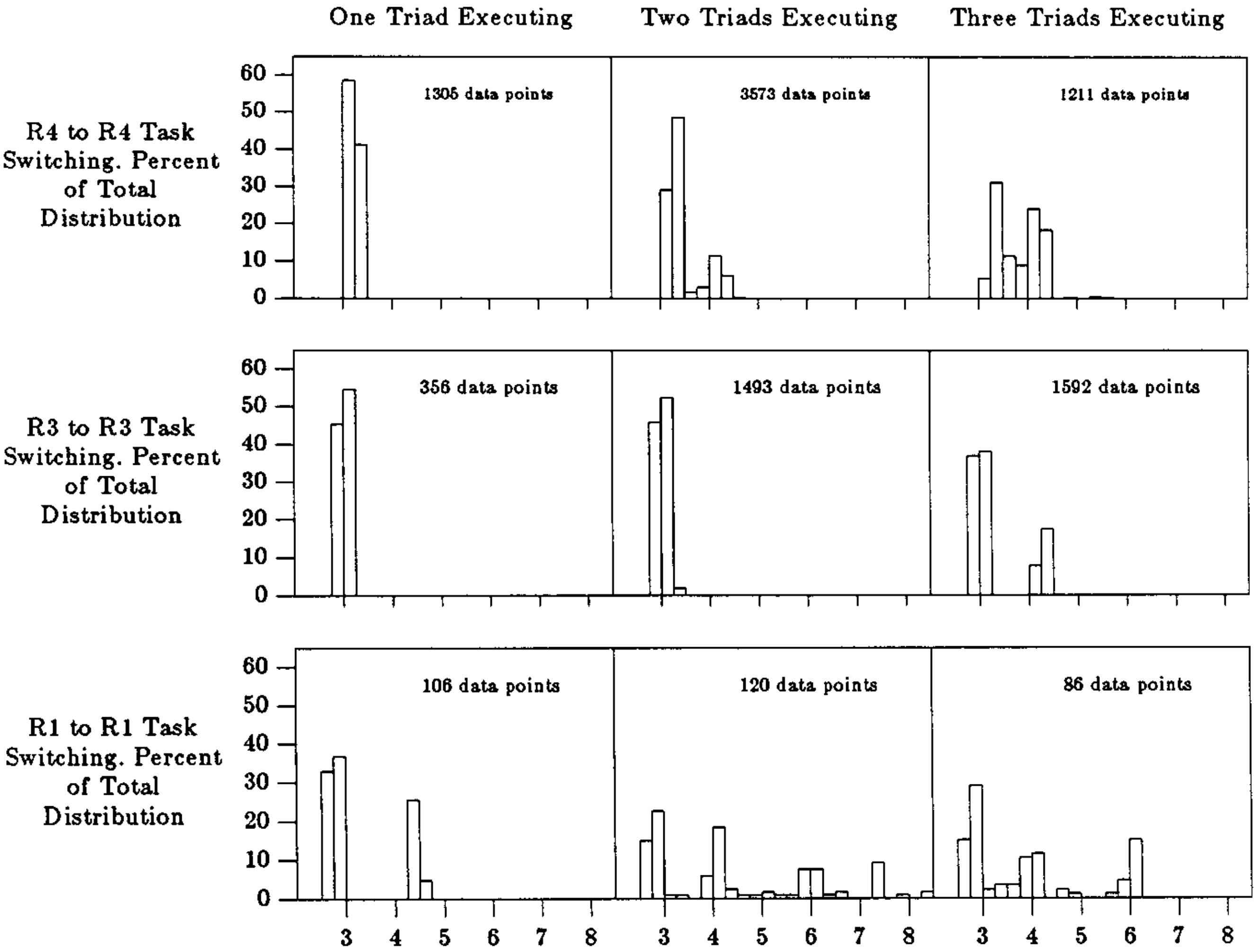

Figure 3-10: Intra-Task Group Switching Times in Milliseconds.

group require the other triad(s) executing another task (usually an R4 task since the R4 frame can be extended by lengthing an R4 task). This method works well for controlling R4 and R3 task orders, but the R1 tasks were interrupted by the start of the next R4 frame. Hence the R4 and R3 switching times were measured under easily repeatable conditions while R1 switching times the conditions were repeatable though to a lesser extent.[16] In general the intratask group switching times are predictable within a certain range. The time for task switching is large in comparison with the desired frame size (40 milliseconds). If a triad needed to execute three R4 tasks in a 40 millisecond frame, 8 milliseconds or 20% would be spent switching between the tasks. This will be summarized in Section 4.

[16] In the R3 and R1 dispatcher there are about 20 bus transactions. Each transaction takes about .2 milliseconds. A delay caused by contention may cause the bus transaction to take two or more times the uncontented time.

3.3.4 Inter-Task Group Switching

As previously discussed, when the dispatcher finds all the tasks in its rate group completed or dispatched, it executes a 'resume' to restart the previously interrupted or pending task. Three cases of this process were studied, see Figure 2-4

1. Time and behavior of an R4 task finish time to the start of the R3 task in the same triad. (R4 to R3.)
2. Time and behavior of an R4 task finish time to the start of the R3 task in the R4 responsible triad.[17] (R4 to R3.)
3. Time and behavior of the R3 task finish time to the start of the R1 task in the same triad. (R3 to R1.)

The behavior of these three processes are summarized in Figure 3-11. As is shown in the figure the behavior of these interactions is predictable and well behaved. The variation in the data is probably the result of bus contention and queue semaphores or the dispatcher executing different sections of code or both. The large spread in the R3 to R1 switching is caused by the same factors as for the spread in the intratask group switching. (See Section 3.3.3.) In summary, the intertask group switching times are predictable over a range of 4.5 to 10.5 milliseconds. The execution times of the dispatcher for the intertask group switching are large for the desired frame iteration rate. The R4 frame is 40 milliseconds with task switching taking 7 milliseconds, a significant percent of the useable time.

3.3.5 Frame Size and Frame Slippage

The last dispatcher experiment involves the actual task iteration rates for the two highest task rate groups, R4 and R3. This experiment measured the difference between consecutive starts of the first task in the rate groups. The results yield the nominal frame size and variation of the frame sizes. The data for these experiments is presented in Figures 3-12 and 3-13.

Figure 3-12 shows the spread of data for the R4 frame size. The frame sizes show a similar grouping for the one and two triad case, and for the three triad case a similar grouping but at different locations. The pattern of the execution times for the R4 frame is 36, 42, 114, 92, ... millisecond for one and two triads and 40, 65, 105, 40, ... milliseconds for three triads. The first number is the start of the major frame. The R4 pattern determines the pattern for the R3

[17] An R4 responsible triad is the last triad to complete its R4 task. This triad is responsible for the start of the next R4 frame by arming its timer interrupt.

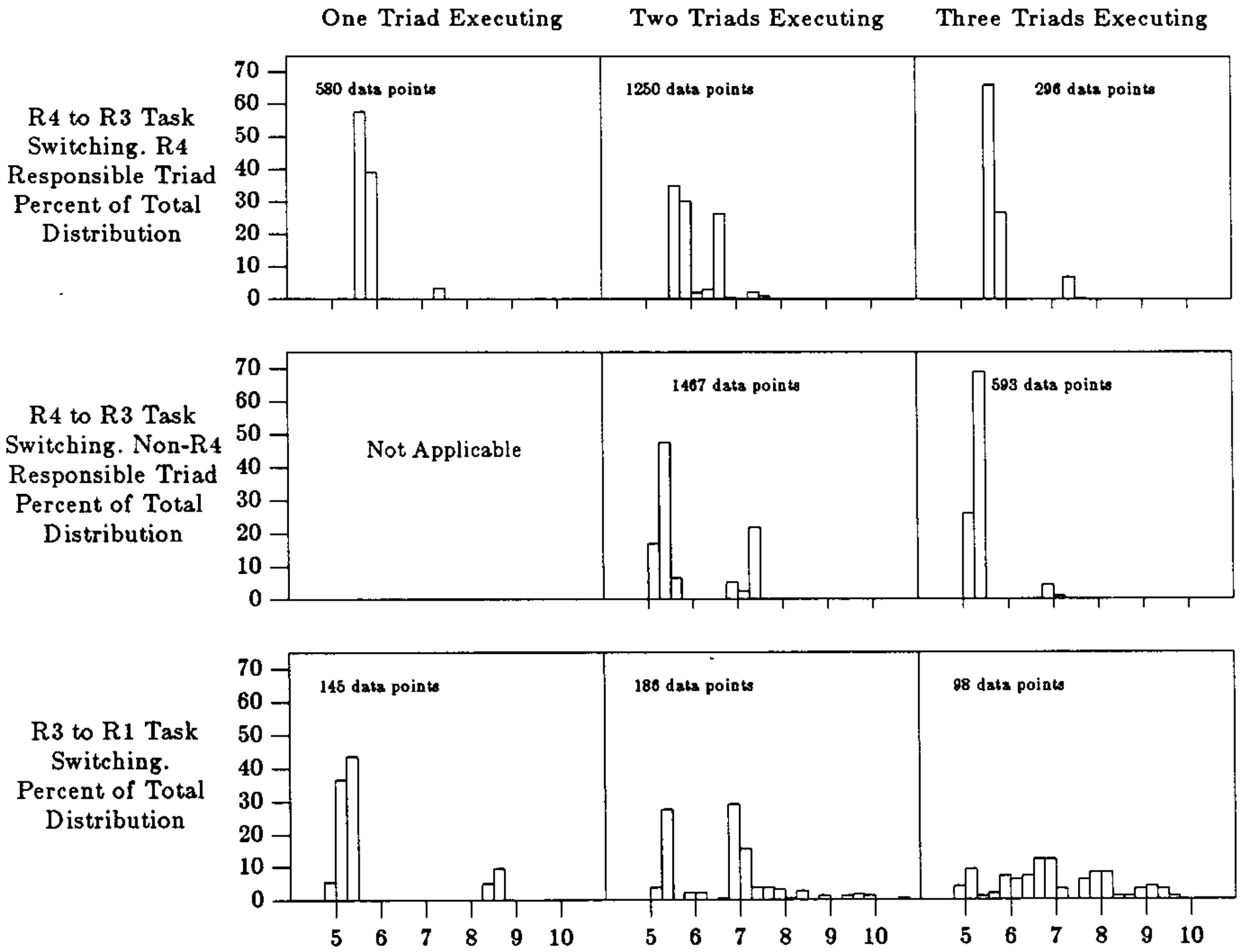

Figure 3-11: Inter-Task Group Switching Times in Milliseconds.

frame. R3 frames are started every second R4 frame, hence the sum of two consecutive R4 frame sizes determine the size of the R3 frame. For the one and two triad case the R4 frame size pattern is 36, 42, 114, 92, ... The first two frame sizes determine the R3 frame size about 78 milliseconds and the second pair determine the next R3 frame size about 205 milliseconds. Similarly for the three triad case the R4 pattern is 40, 65, 105, 40, ... Thus the R3 pattern would be 105, 145,... This pattern differs from the observed pattern, a constant 125 milliseconds, but the difference could be explained by additional overheads occurring in dispatcher. The long frames are in correlation with the long dispatcher times (Section 3.3.1, and Figure 3-7). The dispatcher will not schedule the start of the next R4 frame until the present R4 frame is complete and ten milliseconds is allowed for lower rate group tasks. The two triad case in

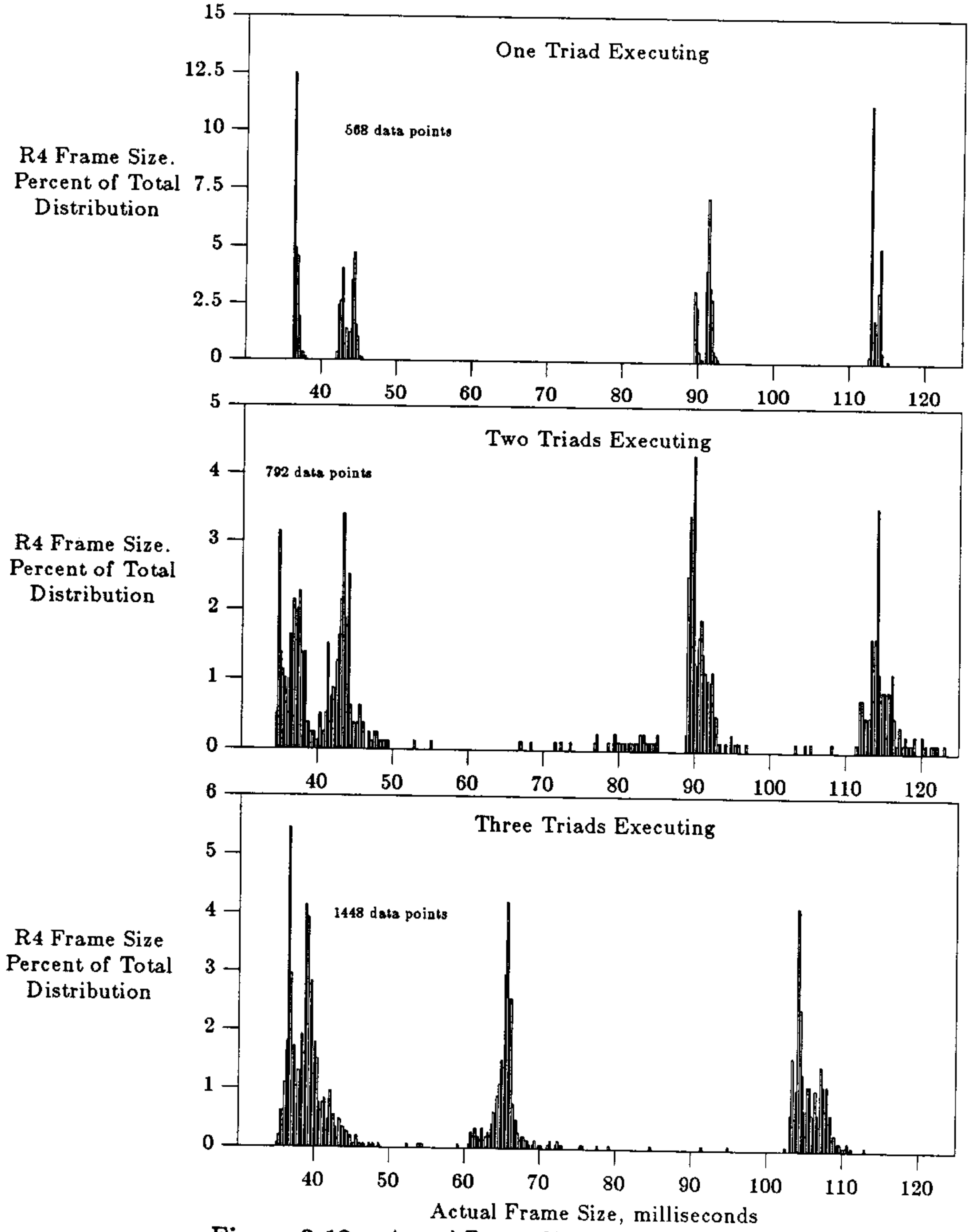

Figure 3-12: Actual Frame Size for R4 Tasks.

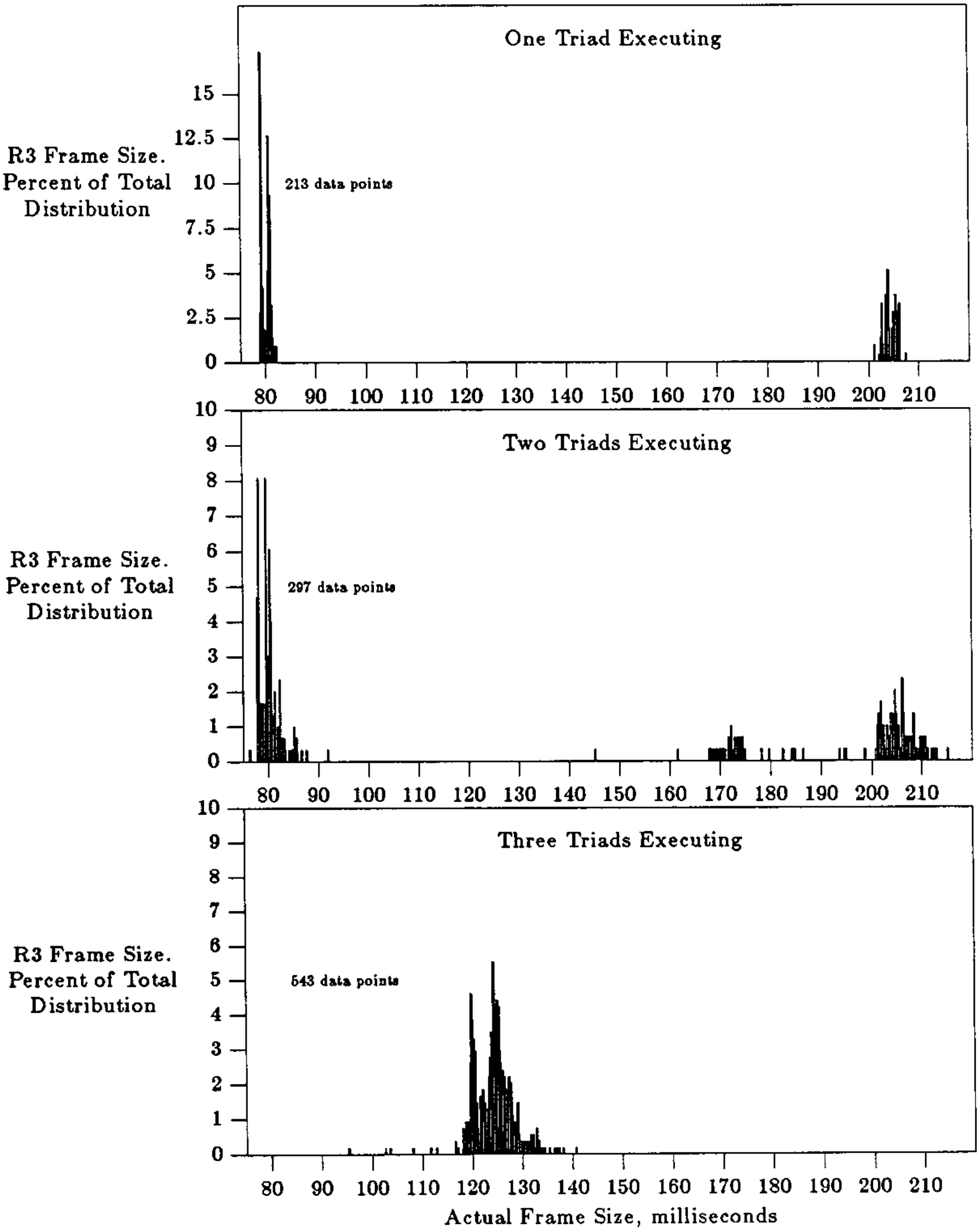

Figure 3-13: Actual Frame Size for R3 Tasks.

Figures 3-7 and 3-12 represent this behavior best:[18] as shown in Figure 3-7 about one-quarter of the execution times lie near 40 milliseconds, and another quarter around 65 milliseconds. Similarly the R4 frame sizes are grouped at 90 and 115 milliseconds or 50 milliseconds greater than the corresponding dispatcher times. From the frame size measurements, the dispatcher time and kick time behavior one can see a problem with the FTMP scheduler meeting its real time constraints.

3.3.6 Dispatcher-Scheduler Software Overhead Summary

This section summarizes the Dispatcher-Scheduler software overhead. Table 3-2 gives a summary of the dispatcher times. The first entry in each block is the average execution time, in milliseconds, followed by the standard deviation of the data. The second entry is the range of the data also in milliseconds, and the third entry is the data sample size.

As seen in the table, the dispatcher software overhead is predictable within a range of about 6 millisecond with a two exceptions. The two exceptions are:

- Frame start interrupt to the application task start, and
- IPC kicks to start other triads working on R4 tasks.

These two problems and the overall problem of long execution times of the dispatcher cause the scheduling problem of late frame starts discussed in Section 3.3.5. An evaluation of possible improvements to the dispatcher will be made in Section 4 specificly dealing with decreased execution times of the system bus primitives, and their possible effect on the dispatcher. The dispatcher should be further characterized.

3.4 Software Overhead for Fault Detection and Isolation

The final experiment presented in this report measures the software overhead for fault detection and isolation. The FTMP software tasks for these include READALL. READALL incrementally reads system memory to assure all copies of the data in the memory elements is the same. The second task is the System Configuration Controller, SCC described in Section 2.2.3. These tasks are specifically dedicated to fault detection and isolation, but they do not include all the software overhead for fault tolerance. Some fault detection commands are done

[18] The exact correlation between the data is not because of different experimental setup, the dispatcher behavior, and the distribution of tasks between triads.

Dispatcher Goal	Number of Triad(s) Running		
	1	2	3
Frame Interupt to Task Start	40.3 ± 25.62 13.75 → 68.5 (6400)	33.7 ± 20.8 13.0 → 67.75 (6400)	33.4 ± 21.1 13.5 → 67.0 (6400)
IPC 'Kick' Time Triad 1 to 2	-	6.79 ± 9.70 1.25 → 27.25 (3569)	2.80 ± 0.49 1.25 → 4.25 (1788)
IPC 'Kick' Time Triad 2 to 3	-	-	5.10 ± 7.55 1.25 → 25.0 (2366)
IPC 'Kick' Time Triad 1 to 3	-	-	7.83 ± 7.45 3.75 → 27.75 (2447)
Intra-Task Group Switching R4	3.10 ± 0.12 3.0 → 3.25 (1305)	3.34 ± 0.38 3.0 → 4.5 (3573)	3.68 ± 0.43 3.0 → 5.5 (1211)
Intra-Task Group Switching R3	2.88 ± 0.12 2.75 → 3.0 (356)	2.89 ± 0.13 2.75 → 3.25 (1493)	3.20 ± .57 2.75 → 4.25 (1592)
Intra-Task Group Switching R1	3.13 ± .77 2.5 → 4.5 (106)	4.26 ± 1.67 2.5 → 8.25 (120)	3.74 ± 1.26 2.5 → 6.0 (86)
Inter-Task Group Switching R4 - R3 (R4-responsible)	5.65 ± 0.32 5.5 → 7.25 (580)	5.93 ± 0.47 5.0 → 7.5 (1250)	5.69 ± 0.45 5.5 → 7.5 (296)
Inter-Task Group Switching R4 - R3 (non-responsible)	-	5.78 ± 0.89 5.0 → 7.25 (1467)	5.28 ± 0.64 5.0 → 7.0 (593)
Inter-Task Group Switching R3 - R1	5.59 ± 1.18 4.75 → 8.5 (145)	6.54 ± 1.13 5.0 → 10.5 (186)	6.81 ± 1.24 4.75 → 9.5 (98)

Table 3-2: Dispatcher Software Overhead Summary.

in the R4 dispatcher, these include, reading of the error latches into system memory, and the reconfiguration and retire commands. This experiment will not take these overhead into account.

The experiment was run using the synthetic workload. The workload allows the starting and ending times of these tasks to be measured, hence determining the execution time. A problem occurs in the measurement of the genuine task time due to interruption of the tasks for higher rate group tasks. By extending the basic R4 frame size from 40 to 250 milliseconds, all tasks including SCC and READALL could finish without interruption. The results for this experiment are presented in Table 3-3.

FTMP Fault Tolerant Software Overhead (All times in milliseconds)		
Task	Execution Time	Range
SCC	43.4 ± 23.3	3.5 → 153.
READALL	3.1 ± 0.9	1.25 → 6.25

Table 3-3: Software Overhead for Fault Tolerance.

The variation in the READALL times are due to bus contention and different sections of code being executed. Whereas the SCC times show the spread of data as SCC executes different states. These states include fault isolation, shadow swapping, transient fault handling routines and self tests. These software overheads are small in comparison to the real time scheduling overheads for FTMP. Further comparisons will be presented in Section 4.

Performance Estimates

4. Performance Estimates

This section estimates the useable throughput of the FTMP system from the dispatcher-scheduler and fault tolerant overheads measured in the previous sections. Table 4-1 gives a break down of the available throughput and overheads observed. In Table 4-1 a major frame is defined as eight R4 frames or one R1 frame.

FTMP Performance Estimates
(Times are millisecond per major frame)

	As Designed		As Running		As Corrected	
FTMP Overhead	Time (msec.)	Percent of Total Time	Time (msec.)	Percent of Total Time	Time (msec.)	Percent of Total Time
Useable Throughput Three Triads.	960.	100.	1983	100	960.	100
R4 Dispatcher Time. 8 per Frame Triad.	384.0	40.0	801.6	40.4	272.6	28.4
Task Switching Times 3-R4, 3-R3 and 6-R1 Tasks are Assumed.	222.9	23.2	222.9	11.2	222.9	23.2
8 R4 responsible.	(45.5)	(4.7)	(45.5)	(2.3)	(45.5)	(4.7)
16 Non responsible.	(84.5)	(8.8)	(84.5)	(4.3)	(84.5)	(8.8)
12 R3 to R1.	(81.7)	(8.5)	(81.7)	(4.1)	(81.7)	(8.5)
3 R1 to R1.	(11.2)	(1.2)	(11.2)	(0.6)	(11.2)	(1.2)
Fault Tolerant Software SCC time. 43.3 READALL time. 3.2	46.5	4.8	46.5	2.3	46.5	4.8
Total Useable Time per Major Frame	306.6	31.9	912.0	45.9	418.0	43.5

Table 4-1: Performance Estimates for FTMP.

In Table 4-1 the second and third columns give the performance estimates assuming a 40 millisecond R4 frame (320 millisecond major frame with three triads executing), and the software overhead times measured in this report. The R4 dispatcher execution time used was 16 milliseconds; the upper limit of acceptable times presented in Figure 3-7. This shows 63% of the available throughput is spent in the dispatcher. Of interest is the software overhead of the fault tolerance tasks, 4.8%. This is a small software price for fault tolerance. (Of course the large cost for the fault tolerance lies in the hardware.) The fourth and fifth columns, of the table use the actual frame sizes as measured, instead of the 40 millisecond R4 frame size assumption and the true dispatcher times measured in this report. The overhead percents are lowered but the frames are extended, showing the present behavior of the system, although the behavior is incorrect for a real time system.

The FTMP Executive Summary, [Draper 82], noted the large bottleneck caused by the long bus access times (Section 3.2). The authors state that one third of the R4 dispatcher time is spent doing bus service routines and by microcoding some of the I/O functions the bus service times could be lower to one-eighth of their current times. Using this estimate, one third of the dispatcher time reduced 88%, will show the R4 dispatcher time lowered to 71% of its current value.[19] The sixth and seventh column of Table 4-1 present these results. The performance increase gained by microcoding some I/O functions was applied only to the R4 dispatcher, if this increase was applied to all functions, a larger performance increase could be expected.

[19] 33% of the dispatcher is reduced 88% for a 29% reduction.

5. Future Work

Although much work has been accomplished in refining the experimental methodology, by applying it to FTMP, the methodology still needs to be further verified by additional experiments on FTMP and by application to other systems, such as the Software Implemented Fault Tolerant (SIFT) computer. SIFT [Wensley 78] was designed with similar design specifications as FTMP and by applying the methodology to SIFT the robustness of the methodology will be shown as well as a comparison the two systems.

In particular the following items are some areas in which further characterization of FTMP may be needed.

- Further characterization of the dispatcher to determine the time consuming sections and, if possible, correct this undesirable behavior. This may be done by characterizing some of the executive primitives the dispatcher uses.
- Determine the overhead required for system reconfiguration. How much overhead is required for the dynamic redundancy of FTMP ?
- Characterize the throughput vs. workload and task distribution. Will the system still meet its deadlines under this increased load ?
- Further characterize the software overhead for both the faulty and fault free situations. This includes the times to isolate faults, and reconfiguration overhead.
- Validate the system configuration controller. Does the controller handle faults correctly? A log of failed units should be kept to determine faults within the units or controller problems.
- Explore the fault coverage in the self test routines. How many faults can the self test routines locate in the bus guardian unit, and system buses ?
- Explore the behavior of multiple faults. [Draper 83b] showed the fault tolerant capabilities of the system by injecting pin level faults. How will the system behave if two faults occur close together?

These experiments move the validation and performance measurements into the application level of the performance evaluation matrix, along with exploring faulty behavior of the system.

6. Conclusions

This report outlined a validation methodology for ultrareliable multiprocessors and applied the methodology to one system, FTMP. The methodology entails a building block approach, starting with simple baseline experiments and building to more complex experiments. Previous work has been done to measure the baseline performance, as well as characterization of most hardware primitives [Clune 84, Feather 85]. This report presents a continuation of the baseline experiments, and presents experiments at the executive, operating systems level. In particular this report presented:

- High level language instruction execution times. The execution times measured were consistent and predictable.
- System memory read and writes times and the variance of the times caused by bus contention. The memory read/write times were a linear function of block size (400 Kbytes/sec.) with an overhead of approximately 150μseconds. Bus contention showed a slight increase in average overhead.
- Dispatcher execution times and overhead. The dispatcher consumed approximately 60% of the system throughput in addition to failing to meet its real time constraints.
- Fault tolerant software overhead. Showed the software overhead involved in the fault tolerant tasks consumed 5% of system throughput.
- Performance estimates of the system. Showed the available processor time and the distribution of the time between the dispatcher, fault tolerant tasks, and application tasks.

From these experiments and their results the following points can be inferred about the FTMP system.

- In the present implementation of FTMP there is a large overhead consumed by the real time dispatcher. About one-third of the dispatcher overhead is caused by the large overhead involved in the system bus access which is an implementation problem and not dependent upon the fault tolerant design of FTMP.
- A relatively small software overhead is involved with the system configuration controller, fault detection and isolation, thus showing the advantage of hardware voting over software voting to obtain system fault tolerance.

The goal of the validation methodology is to thoroughly test and characterize the performance and behavior of an ultra-reliable computer system. The validation methodology presented in Section 2.1.1 and applied throughout this report proved effective in the following areas.

- The methodology uncovered both system implementation dependencies with the instruction executions times and behavioral oddities in the dispatcher-scheduler.
- The methodology showed system validation can occur without using life testing approaches.

- By applying a building block approach in a systematic manner the FTMP system was broken down into manageable levels of experimentation thus concealing system complexity from the experimenter.
- Finally most of the experiments were run at the system level, demonstrating system validation can be independent of the implementation (LSI or VLSI.)

The enumerated items demonstrate the feasibility of the validation methodology by addressing the problems encountered with the validation of ultrareliable systems. Tests upon other systems such as SIFT will demonstrate the robustness of the methodology as well as provide comparisons of the two systems.

I. Instruction Execution Times

This appendix contains the tabulate result of the execution times of all the instructions measured. The predicted execution times are from [CAPS Instruction Set 79].

Instruction Execution Times Summary, 16 Bit Integer Operators
(All times in micro-seconds, Range is 95% Confidence Interval)

HLL Instruction	Description	Execution time per one loop	Instruction Time	Predicted Time	Precent Difference
B = 1	Integer assign 4 bits	20.2 ± .30	4.0 ± .22	2.7	48.1
B = 17	Integer assign 8 bits	22.2 ± .31	6.0 ± .22	3.6	66.7
B = 257	Integer assign 16 bits	22.7 ± .30	6.5 ± .22	4.1	58.5
J = 1	Integer assign extended reference	22.4 ± .30	6.2 ± .22	4.1	51.2
D = B	Integer variable assign	23.2 ± .29	5.5 ± .21	4.1	34.1
B = J	Variable assign extended reference	30.2 ± .31	12.5 ± .22	5.4	131.
D = - B	Integer negate	30.7 ± .31	7.0 ± .22	6.3	11.1
D = B + C	Integer addition	33.7 ± .30	10.0 ± .22	7.7	30.0
D = B * C	Integer multiply	46.4 ± .29	20.2 ± .21	12.8	57.8
D = B / C	Integer division	37.9 ± .30	21.7 ± .22	13.1	65.6
D = .N. B	Bit wise negate	29.7 ± .31	6.0 ± .22	5.3	13.2
D = B .A. C	Bit wise and	31.2 ± .30	15.0 ± .22	7.8	92.3
D = B .V. C	Bit wise or	32.7 ± .31	15.0 ± .22	7.8	92.3
D = B .X. C	Bit wise exclusive or	31.2 ± .30	15.0 ± .22	7.8	92.3
D = B .RS. C	Right shift (1 bit)	33.7 ± .29	16.0 ± .21	-	-
D = B .RS. C	Right shift (2 bits)	32.4 ± .30	16.2 ± .22	-	-
A3 = B EQL C	Integer compare ==	39.4 ± .30	23.2 ± .22	14.2	63.3
A3 = B NEQ C	Integer compare !=	38.7 ± .30	21.0 ± .22	11.9	76.4
A3 = B LES C	Integer compare <	37.4 ± .30	21.2 ± .22	12.1	75.2
A3 = B GEQ C	Integer compare >=	41.2 ± .30	23.5 ± .22	14.4	63.2

Table I-1: Instruction Executions Time: Integer

Instruction Execution Times Summary 16 bit Fixed Point Operators
(All times in micro-seconds, Range is 95% Confidence Interval)

HLL Instruction	Description	Execution time per one loop	Instruction Time	Predicted Time	Precent Difference
B = .1	Real assign	24.4 ± .29	6.7 ± .21	4.1	63.4
B = C	Real variable assign	23.3 ± .29	5.5 ± .21	4.1	34.1
A = - B	Real negate	32.2 ± .31	8.5 ± .22	6.3	34.9
A = B + C	Real addition	33.7 ± .29	10.0 ± .21	7.7	30.0
D = B * C	Real multiply	38.2 ± .30	20.5 ± .22	12.6	62.7
A = B / C	Real division	42.2 ± .29	24.5 ± .21	13.3	84.2
A3 = B EQL C	Real compare ==	40.9 ± .29	23.2 ± .21	14.2	63.3
A3 = B NEQ C	Real compare !=	38.7 ± .30	21.0 ± .22	11.9	76.4
A3 = B LES C	Real compare <	38.9 ± .30	21.2 ± .22	12.1	75.2
A3 = B GEQ C	Real compare >=	41.2 ± .29	23.5 ± .21	14.4	63.2

Table I-2: Instruction Executions Time: Real

Instruction Execution Times Summary Long, 32 bit Integers
(All times in micro-seconds, Range is 95% Confidence Interval)

HLL Instruction	Description	Execution time per one loop	Instruction Time	Predicted Time	Precent Difference
B = 1	Long assign	29.7 ± .28	12.0 ± .21	5.7	110.5
B = 17	Long assign (8 bits)	31.9 ± .29	14.2 ± .21	6.6	115.2
B = 257	Long assign (16 bits)	32.4 ± .30	14.7 ± .21	7.1	107.0
B = 65537	Long assign (4 bits)	29.7 ± .30	12.0 ± .22	5.7	110.5
J = 1	Extended variable reference assign	30.4 ± .30	12.7 ± .22	7.1	78.9
B = C	Long variable assign	32.2 ± .29	14.5 ± .21	7.8	85.9
B = J	Extended variable reference	33.2 ± .30	17.0 ± .22	9.3	82.8
A = - B	Long negate	42.2 ± .30	18.5 ± .22	15.3	20.9
A = B + C	Long addition	48.7 ± .30	32.5 ± .22	17.9	81.5
D = B * C	Long multiply	64.9 ± .29	48.7 ± .22	31.7	53.6
A = B / C	Long division	87.4 ± .31	71.2 ± .22	45.4	56.8
A3 = B EQL C	Long compare ==	56.4 ± .29	38.7 ± .21	17.8	117.4
A3 = B NEQ C	Long compare !=	54.2 ± .30	36.5 ± .22	15.5	135.5
A3 = B LES C	Long compare <	54.2 ± .28	36.5 ± .21	15.7	132.5
A3 = B GEQ C	Long compare >=	56.4 ± .30	38.7 ± .22	18.0	115.0

Table I-3: Instruction Executions Time: Long Integers

Instruction Execution Times Summary Boolean Operators
(All times in micro-seconds, Range is 95% Confidence Interval)

HLL Instruction	Description	Execution time per one loop	Instruction Time	Predicted Time	Precent Difference
A = TRUE	Boolean assign	21.7 ± .31	4.0 ± .22	2.7	48.1
A = B	Boolean variable assign	23.2 ± .26	5.5 ± .20	4.1	34.1
A = NOT B	Boolean negate	34.6 ± .30	10.9 ± .22	7.9	38.0
A = B OR C	Boolean OR = F 2 tests required	39.2 ± .28	21.5 ± .22	13.1	64.1
A = B OR C	Boolean OR = T on 1st condition	36.9 ± .30	20.7 ± .22	10.2	102.9
A = B OR C	Boolean OR = T on 2nd condition	41.4 ± .30	23.7 ± .22	15.4	53.9
A = B AND C	Boolean AND = T 2 tests required	41.4 ± .31	25.2 ± .22	15.4	63.6
A = B AND C	Boolean AND = F on 1st condition	32.7 ± .30	15.0 ± .22	7.9	89.9
A = B AND C	Boolean AND = F on 2nd condition	39.2 ± .30	23.0 ± .22	13.1	75.5

Table I-4: Instruction Executions Time: Boolean

Instruction Execution Times Summary Miscellaneous Operators
(All times in micro-seconds Range is 95% Confidence Interval)

HLL Instruction	Description	Execution time per one loop	Instruction Time	Predicted Time	Precent Difference
NULL	Null loop1 (for)	17.7 ± .29	-	15.7+	-
NULL	Null loop2 (loopf)	23.7 ± .30	-	17.6+	-
Test0()	Procedure call	57.2 ± .31	37.0 ± .22	35.8	54.
Test1(B)	Procedure call	67.9 ± .29	51.7 ± .21	38.0	36.0
Test2(B,C)	Procedure call	73.7 ± .31	57.5 ± .22	40.2	43.0
Test3(B,C,D)	Procedure call	79.4 ± .32	63.2 ± .22	42.4	49.0
Test4(B,C,D,E)	Procedure call	85.2 ± .32	69.0 ± .22	44.6	54.7
If A3 then B = 1	Conditional, True	32.7 ± .31	9.0 ± .22	7.9	13.9
If A3 then B = 1	Conditional, False	29.2 ± .31	5.5 ± .22	5.2	5.8
If A3 then B = 1 Else C = 1	Conditional, True	36.9 ± .31	13.2 ± .22	10.1	30.7
If A3 then B = 1 Else C = 1	Conditional, False	33.2 ± .32	9.5 ± .22	7.9	20.3

Table I-5: Instruction Executions Time: Miscellaneous Operators

II. Block Transfer Execution Times

This appendix contains tabulated results of the block transfer experiment, Section 3.2.

Block Transfer Times: Read from System to Local Memory

(Times given in micro-seconds, Ranges are 95% Confidence Intervals)

Block Size	Block Transfer Time with	
(words = 16 bits)	1 Triad	2 Triads
1	162.7 ± 1.54	170.7 ± 4.02
2	165.7 ± 1.06	172.9 ± 4.60
3	168.6 ± 1.38	176.3 ± 5.94
4	171.6 ± 1.45	176.3 ± 4.33
5	174.7 ± 0.74	180.3 ± 3.86
10	189.6 ± 0.82	193.3 ± 4.31
15	204.7 ± 0.71	208.6 ± 3.71
20	224.7 ± 0.76	230.9 ± 3.99
25	235.7 ± 1.08	239.3 ± 3.86
50	310.8 ± 1.13	316.0 ± 6.12
100	460.7 ± 1.11	465.3 ± 6.23
125	535.6 ± 1.02	541.0 ± 6.63
150	610.7 ± 1.10	621.3 ± 10.3
175	685.6 ± 1.02	692.0 ± 8.48
200	760.8 ± 1.13	765.1 ± 6.56

Table II-1: Block Transfer Times, Read from System to Local Memory

Block Transfer Times: Write from Local to System Memory

(Times given in micro-seconds, Ranges are 95% Confidence Intervals)

Block Size	Block Transfer Time with	
(words = 16 bits)	1 Triad	2 Triads
1	158.0 ± 1.35	170.4 ± 4.03
2	163.7 ± 1.35	170.6 ± 3.77
3	168.6 ± 1.38	178.5 ± 6.22
4	173.7 ± 1.34	178.9 ± 3.81
5	178.7 ± 1.35	186.2 ± 4.45
10	203.6 ± 1.38	207.8 ± 2.76
15	228.6 ± 1.38	233.7 ± 3.90
20	258.7 ± 1.36	263.1 ± 3.56
25	283.7 ± 1.35	290.0 ± 5.81
50	408.6 ± 1.38	413.7 ± 4.23
100	658.6 ± 1.38	663.6 ± 5.00
125	783.7 ± 1.36	790.8 ± 7.63
150	908.8 ± 1.33	919.9 ± 10.1
175	1033.8 ± 1.31	1039.8 ± 6.04
200	1158.7 ± 1.35	1164.2 ± 6.18

Table II-2: Block Transfer Times, Write from Local to System Memory

References

[CAPS Instruction Set 79]
CAPS Instruction Set Description
Rockwell Collins, 1979.

[Clune 84] Ed Clune.
Analysis of the Fault Free Behavior of the FTMP Multiprocessor System.
Master's thesis, Carnegie - Mellon University, 1984.

[Draper 82] *Development and Evaluation of a FTMP Computer, Vol IV, FTMP Executive Summary*
Charles Stark Draper Laboratories, 1982.

[Draper 83a] *Development and Evaluation of a FTMP Computer, Vol II, FTMP Software*
Charles Stark Draper Laboratories, 1983.
CR166072.

[Draper 83b] *Development and Evaluation of a FTMP Computer, Vol III, FTMP Test and Evaluation*
Charles Stark Draper Laboratories, 1983.
CR166073.

[Draper 83c] *Development and Evaluation of a Fault-Tolerant Multiprocessor (FTMP) Computer, Vol I, FTMP Principles of Operations*
Charles Stark Draper Laboratories, 1983.
Contract Report (CR) 166071.

[Feather 85] Frank E. Feather.
Validation of a Fault-Tolerant Multiprocessor: Baseline Experiments and Workload Implementation.
Master's thesis, Carnegie - Mellon University, 1985.

[Ferrari 78] Domenico Ferrari.
Computer Systems Performance Evaluation.
Prentice-Hall, 1978.

[Hopkins 78] Hopkins, A.L., etal.
FTMP - A Highly Reliable Multiprocessor.
IEEE Trans. on Computers :1221-1237, October, 1978.

[Lala, P.K. 85] Lala, Parag K.
Fault Tolerant & Fault Testable Hardware Design.
Prentice Hall International, 1985.

(NASA 79a) NASA-Langley Research Center.
Validation Methods for Fault-Tolerant Avionics and Control Systems - Working Group Meeting I, NASA-Langley Research Center, 1979.
NASA Conference Publication 2114.

(NASA 79b) Research Triangle Institute.
Validation Methods for Fault-Tolerant Avionics and Control Systems - Working Group Meeting II, NASA-Langley Research Center, 1979.
NASA Conference Publication 2130.

[Siewiorek and Swarz 82]
Siewiorek, Daniel P., Swarz, Robert S.
The Theory and Practice of Reliable System Design.
Digital Press, 1982.

[Siewiorek, Bell, and Newell 82]
Siewiorek, Daniel P.,Bell, C. Gordon, and Newell, Allen.
Computer Structures: Principles and Examples.
McGraw-Hill Book Company, 1982.

[Toy 78] W.N. Toy.
Fault-Tolerant Design of Local ESS Processors.
IEEE Trans on Computers :1726-1745, October, 1978.

[Walpole and Myers 82]
Ronald E. Walpole, and Raymond H. Myers.
Probability and Statistics for Engineers and Scientists.
The Macmillan Company, 1982.

[Wensley 78] Wensley, J.H., etal.
SIFT: A Computer for Aircraft Control.
IEEE Trans. on Computers :1240-1255, October, 1978.