NASA Contractor Report 178236

FAULT-FREE PERFORMANCE VALIDATION OF FAULT-TOLERANT MULTIPROCESSORS

(NASA-CR-178236) FAULT-FREE PERFORMANCE VALIDATION OF FAULT-TOLERANT MULTIPROCESSORS (Carnegie-Mellon Univ.) 68 p CSCL 09B

N87-17448

Unclas

G3/62 44011

Edward W. Czeck, Frank E. Feather,
Ann Marie Grizzaffi, Zary Z. Segall,
and Daniel P. Siewiorek

CARNEGIE-MELLON UNIVERSITY
Pittsburgh, Pennsylvania

Grant NAG1-190
January 1987

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

Table of Contents

List of Figures

List of Tables

Abstract

By the 1990's, aircraft will employ complex computer systems to control flight-critical functions. Since computer failure would be life threatening, these systems should be experimentally validated before being given aircraft control.

A validation methodology for testing the performance of fault-tolerant computer systems was developed and applied to the Fault-Tolerant Multiprocessor (FTMP) at NASA-Langley's AIRLAB facility. This methodology was claimed to be general enough to apply to any ultrareliable computer system.

The goal of this research was to extend the validation methodology and to demonstrate the robustness of the validation methodology by its more extensive application to NASA's Fault-Tolerant Multiprocessor System (FTMP) and to the Software Implemented Fault-Tolerance (SIFT) computer System. Furthermore, the performance of these two multiprocessors was compared by conducting similar experiments.

An analysis of the results shows high level language instruction execution times for both SIFT and FTMP were consistent and predictable, with SIFT having greater throughput. At the operating system level, FTMP consumes 60% of the throughput for its real-time dispatcher and 5% on fault-handling tasks. In contrast, SIFT consumes 16% of its throughput for the dispatcher, but consumes 66% in fault-handling software overhead.

1. Introduction

Aircraft today employ computers to perform isolated functions. If a computer fails, its tasks are assumed by the aircrew without loss of life or cargo. Soon 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^{-9} at ten hours. This specified failure rate translates to less than one failure per million years of operations. Two multiprocessor systems designed to these specifications, SIFT [Wensley et al. 78] and FTMP [Hopkins et al. 78], have been delivered to NASA's Avionics Integrated Research Laboratory (AIRLAB).

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 the experimentation described in this paper is to refine the experimental methodology through its application to FTMP and SIFT. This report covers the following:

- A background of the experimental methodology and the hardware and software environments for both SIFT and FTMP.
- The instruction set, hardware level baseline experiments conducted on SIFT and FTMP along with a comparison of the two systems.
- The executive level baseline experiments conducted on the two systems, and a comparison of the result where applicable.
- Conclusions and future work for the computer systems and the validation methodology.

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 are no shortcuts 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, ultrareliable system other problems arise that make the validation task difficult. Some of these problems are: [NASA 79b]

- Life testing is inappropriate, due to large mean-time-to-failure of the system.
- 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 Validation Framework

NASA held several workshops to determine validation procedures. One [NASA 79b] in particular produced a detailed outline of a validation procedure. The procedure is based on a building block approach. Primitive system 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 conducted. This orderly progression insures uniform, thorough coverage and maximizes the ability to locate the cause of unexpected phenomena. The steps in the 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 system.
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. This report presents fault-free baseline performance measurements.

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, as depicted in 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 Primitive 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 up to 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 for use in an ultrareliable avionics environment. The architecture is discussed in [Hopkins et al. 78] and [Draper 83a]. This section gives an overview describing the hardware structure, the software composition, and the experimental environment.

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

[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, working as one processor element of a multiprocessor.

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.).

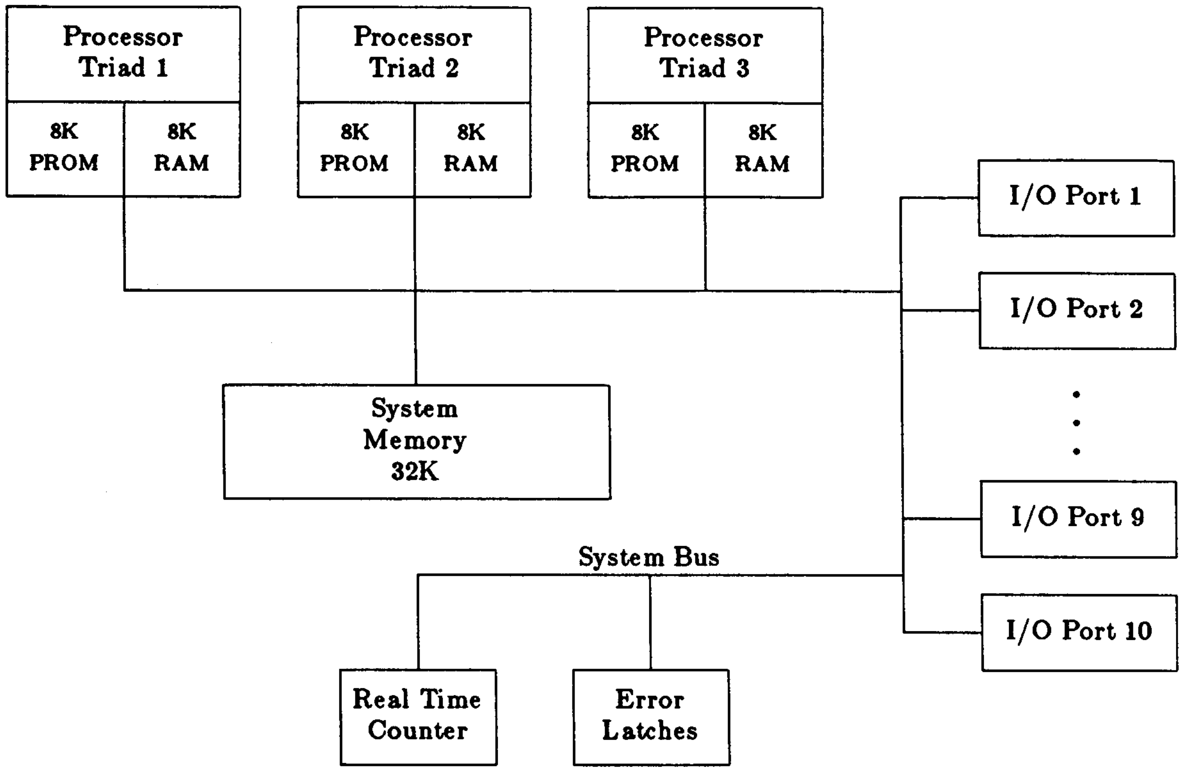


Figure 2-1: FTMP Structure, Programmers Model

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 integrity 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.

2.2.2 FTMP Software

The software support for FTMP includes the real time operating system and the software tasks required to maintain system reliability. The following two sections describe the scheduling strategy for the operating system and the software tasks required to maintain the fault tolerant characteristics of the system.

2.2.2.1 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 two parts, an overview and definition of the task frame structure and an example of scheduling as seen by a single triad. [Draper 83b]

An FTMP task is a single "program" or thread of execution. Tasks in a real time environment run at regular intervals called task iteration rates. All tasks do not need to 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 2-2 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.

The dispatch strategy for FTMP frame structure as seen by a single triad proceeds in the following manner. Figure 2-3 presents a diagram of this process.

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 iteration 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.[2]
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.

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

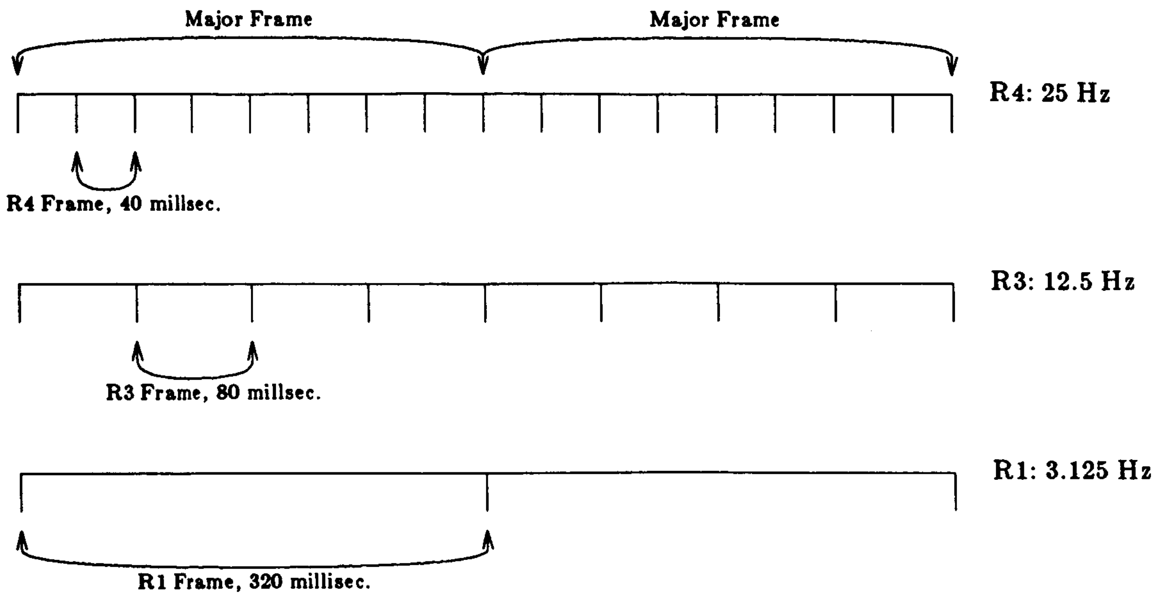


Figure 2-2: FTMP Task Frame Structure

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.
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 2-3a 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[3].
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 2-3a), is suspended until the R4 tasks (and possibly R3 tasks) of this frame complete. The R4 dispatcher begins its execution, does housekeeping and then works on the R4 application tasks.
9. When all the R4 tasks are completed, control is passed to the previously interrupted task or pending task, such as the R3-application task 2 shown in Figure 2-3b.
10. This process continues with each major frame.

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

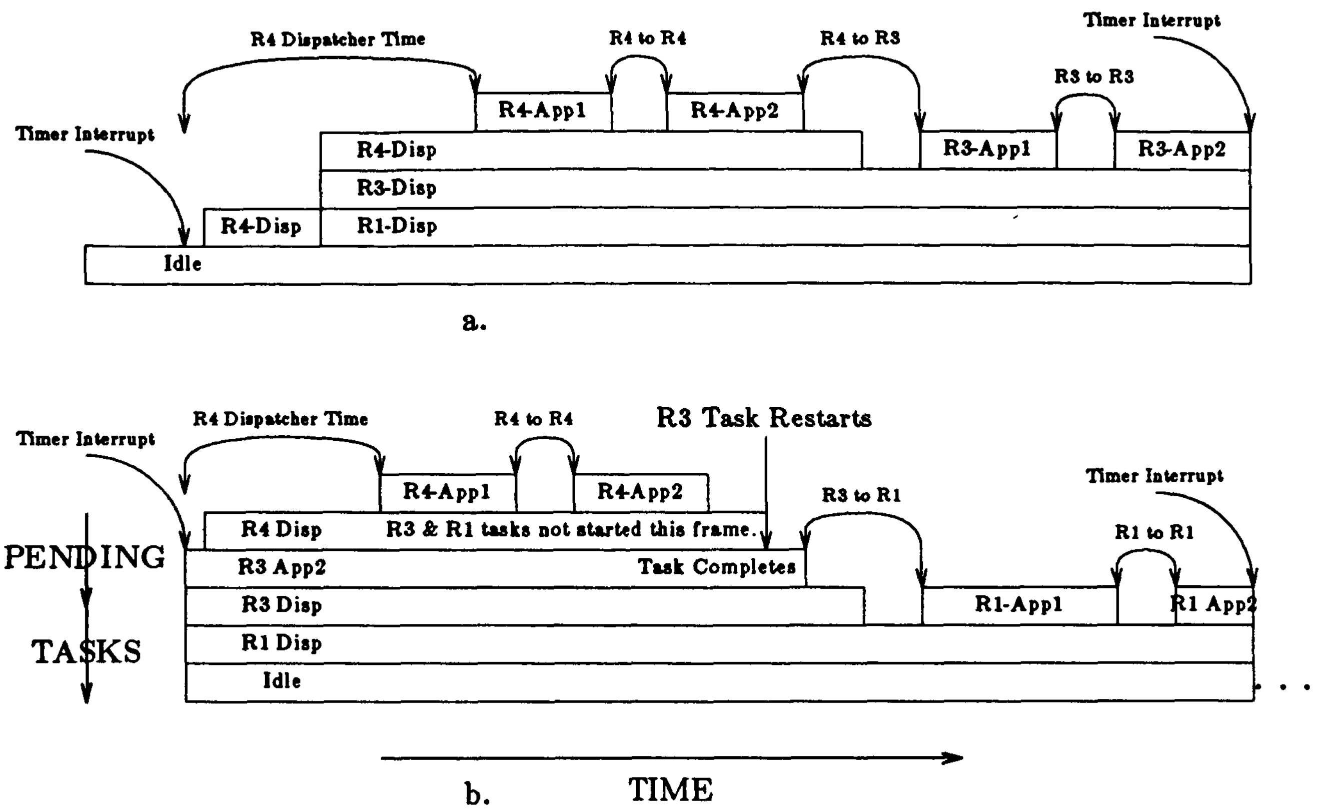


Figure 2-3: FTMP Dispatcher Scheduler Strategy, Showing Two Consecutive Frames

A potential problem which arises in this strategy is the hogging of the CPU cycles by the highest rate group. FTMP handles this situation by delaying or slipping the start of the next frame (R4, R3, or R1.) The stretching of the frame should only occur under abnormal conditions such as a temporary increase in the workload (e.g. fault isolation.)

2.2.2.2 FTMP Fault-Handling Software

Errors[4] are detected with the hardware majority voters of the bus interface units. An error is corrected by voting to maintain system integrity, 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 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[5] , and compacts the error latches into four words, one for each bus quintuple, for further processing.

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

[5] 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.

- If no errors were detected, SCC rotates active and shadowing elements (processors, memories, and buses). The rotating of active and spare 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 crosses a threshold, the unit is retired.

2.2.3 FTMP Experimental Environment

Figure 2-4 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 errors 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, 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 iteration values in the experimental tasks, and to dump data from FTMP's system memory to the VAX for further analysis.

In an effort to shorten the experimental turnaround time, a synthetic workload generator was proposed by [Clune 84] and developed by [Feather et al. 85]. A synthetic workload is a set of programs designed to exercise a computer system to check its performance and behavior under artificial conditions. A natural workload is an environment where the system does useful work. 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.

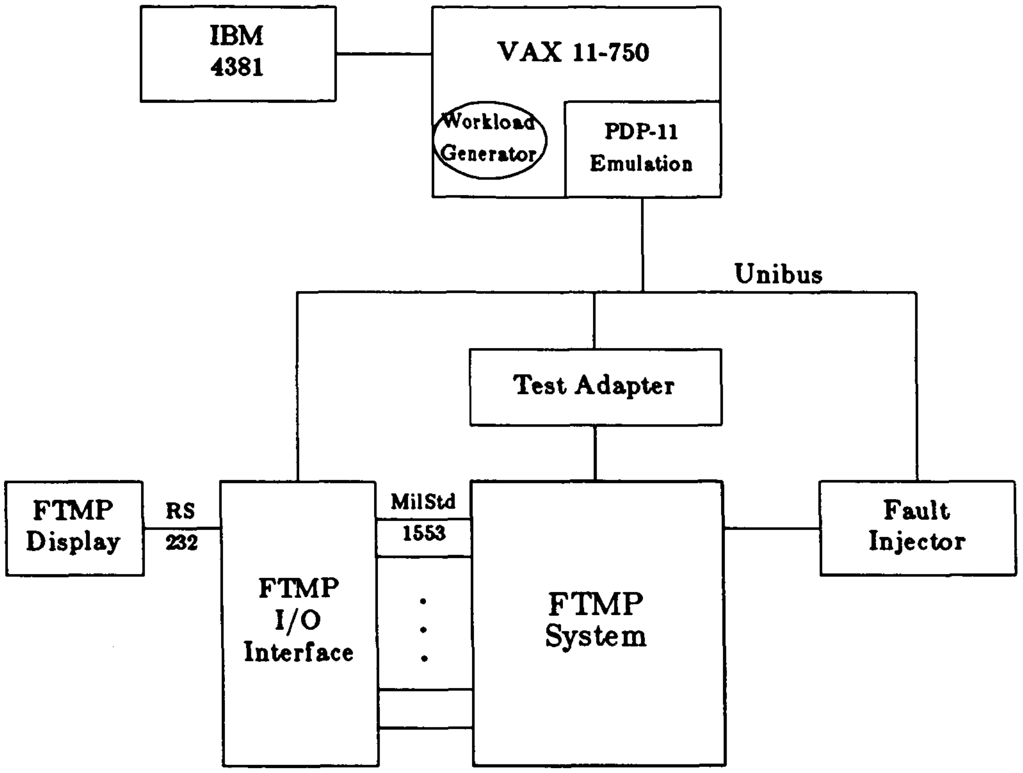


Figure 2-4: FTMP Experimental Environment

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 2-5. 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.

```
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 2-5: Representation of a Synthetic Workload Task

2.3 Software Implemented Fault Tolerance Structure

SIFT was designed and built by Bendix Flight Systems Division, under subcontract to SRI International, and delivered to the AIRLAB in April 1982. This section gives a brief overview of the SIFT's hardware configuration, software, and experimental environment [SRI 84].

2.3.1 Hardware Configuration

The SIFT architecture is made up of a fully distributed configuration of Bendix BDX-930 processors, with point-to-point communication links between every pair of processors as shown in Figure 2-6. Although SIFT was designed and built to accommodate eight processors, there are seven in the current system. Reliability estimations have demonstrated six are needed to meet the required safety margin of less than 10^{-10} probability of failure per hour [Palumbo and Butler 85]. The seventh processor is used by the Data Acquisition System described in the next section.

In a fully distributed system, dependency on shared facilities are kept to a minimum. Therefore, each SIFT processor contains its own main memory, power supply, clock, and I/O channel. A block diagram of a SIFT processor is given in Figure 2-7. Each processor in the system comprises:

- 16-bit bit-sliced CPU.
- 32K words of static random access memory (RAM) which holds the SIFT executive program, the application programs, the transaction and data files, and the control stack.
- A broadcast controller for interprocessor communication.
- A 1553A controller used to support external I/O to terminals, sensors, or avionics modules.
- 1K words datafile memory used as a buffer area for the broadcast and 1553 controller.
- 1K words transaction file memory used to hold the destination address of the values in the datafile to be transmitted.

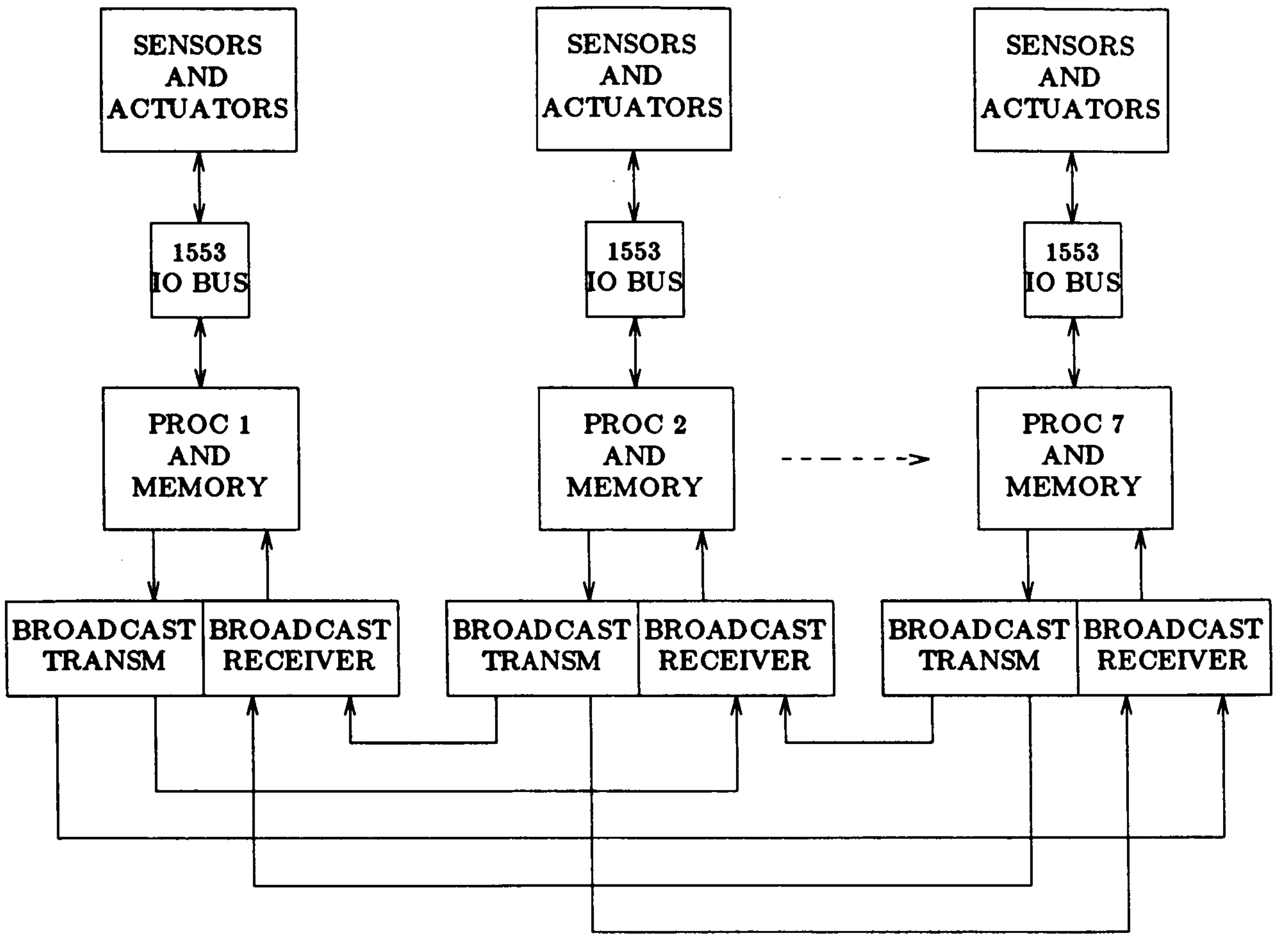


Figure 2-6: Block Diagram of SIFT Distributed System

- A real-time clock driven by a 16 MHz crystal.

2.3.2 SIFT Software

To run an experiment on SIFT, the user writes a task in Pascal on the host computer. Once a task is written, it is compiled, assembled, and linked with the SIFT operating system. This procedure creates an absolute executable image file that can be loaded directly onto the selected SIFT processors. Reliability is achieved by replicating the task on more than one processor. The number of processors chosen is specified by the user, whose decision is based on the importance of the task.

Allocation of a task is done through a user defined Schedule Table. The Schedule Table lists the set of tasks that will be periodically dispatched, along with task specific information. It is the user's job to decide the order tasks are executed, the number of processors used for replication, and the data to be voted. The user must also specify the "duration" of the task in increments of 1.6 millisecond slots. This step insures results are broadcasted in time for voting. It also prevents a non-faulty processor from being configured out of the system because of task time-out.

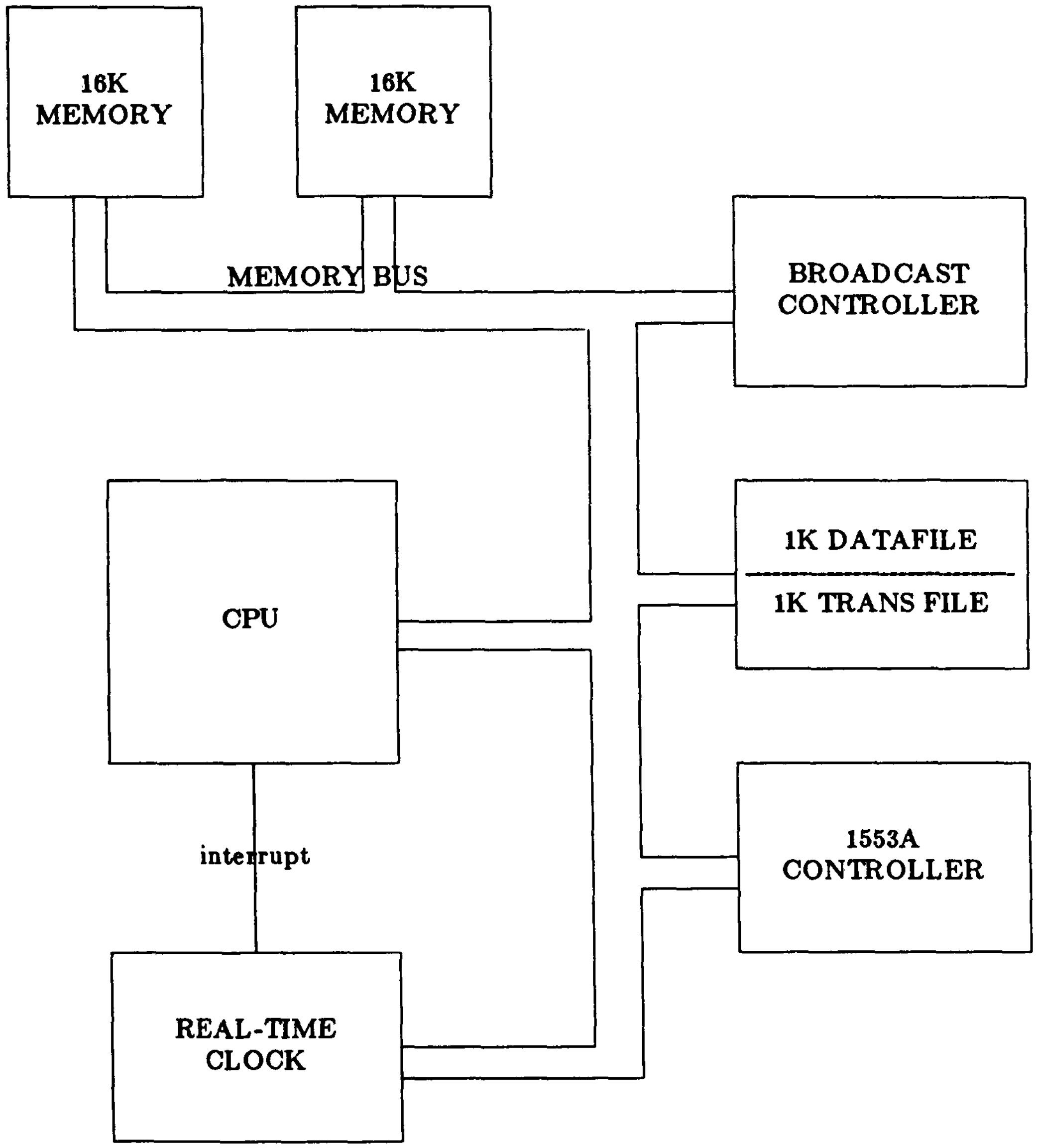


Figure 2-7: A SIFT Processor

After execution of a task, the results from each processor are compared, or "voted" on. If all copies are not the same, an error has occurred. These errors are recorded in the processors' memories to assist the Executive System in determining which processor is faulty. If an error occurred, the Executive System masks the fault by ensuring that only the correct or "majority" value is passed onto the next task. Fault masking prevents a faulty unit from causing problems in the system, such as corrupting a non-faulty processor's memory. If fault masking is not done, a faulty or "malicious" processor could create a life threatening situation, such as transmitting an invalid control signal. Once a processor has been found faulty, the Executive System reassigns the processor's tasks to another processor, thereby configuring it out of the system.

2.3.3 Experimental Environment

SIFT provides the experimenter with a user-friendly test environment, promoting experimentation through interactive facilities designed to help prepare, exercise, and observe the system's behavior. From a terminal linked to the host computer, a researcher can create and run experiments on SIFT, collect data, print out files, and dump data to an on-line printer. Figure 2-8 depicts the test environment as seen by the user. All communication to and from SIFT is through a VAX-11/750. This host computer is solely dedicated to SIFT research. NASA also installed added features to the SIFT environment to enhance experimental conditions: a Data Acquisition System (DAS) for improved data collection and a global clock for improved measurement conditions.

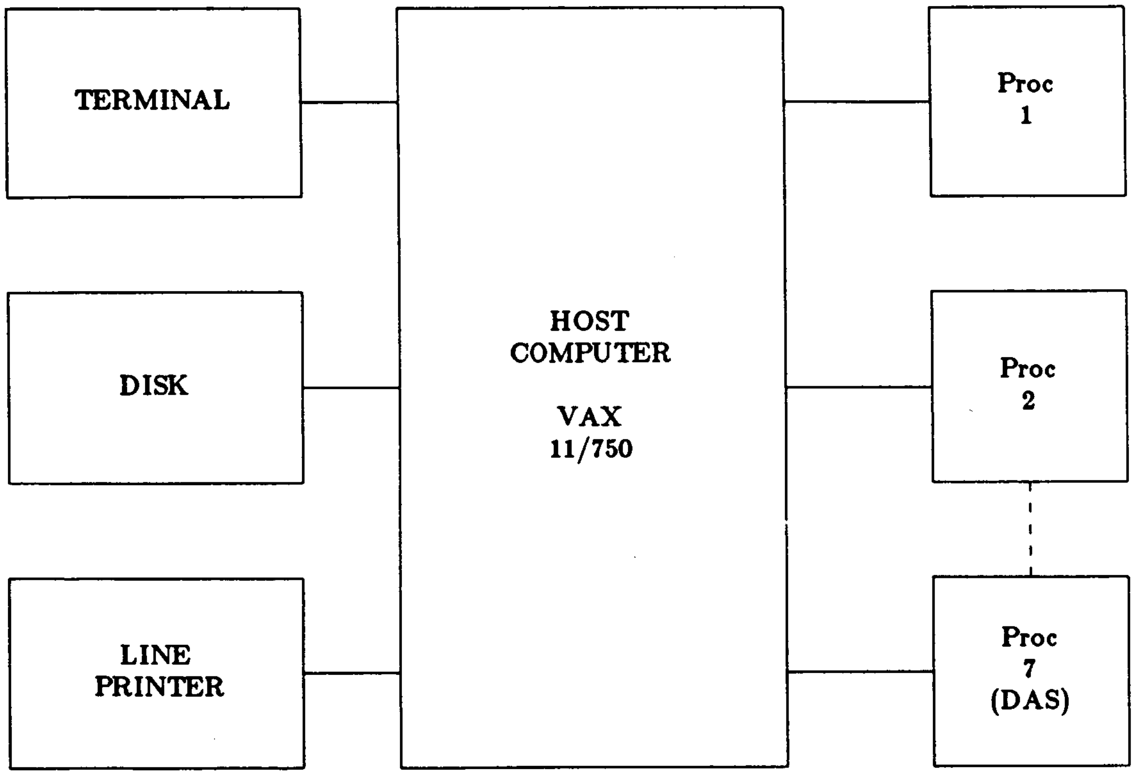


Figure 2-8: The SIFT Test Environment

DAS is made up of many integrated programs that receive and analyze data from the SIFT processors. These programs are downloaded to the seventh SIFT processor, which can then control data collection. Before this system was created, data collection was limited to 4K words of memory. With the Data Acquisition System, information is sent from the SIFT processors to a disk capable of holding 50K blocks, a total of 12.8 Mwords. DAS requires some initial preparation, but it features an interface program that facilitates the task. A preprocessing program is also available which provides the user with the ability to manipulate the data straight from the disk, and the ability to specify what data to save for later processing.

The global clock is a 16-bit counter, like the real-time clocks of the SIFT processors, except that it is an independent measuring device. It features a programmable time-base so the user can specify the resolution of the clock (i.e. 1 microsecond, 1 millisecond, etc.). The processors gain access to the clock via a read bus. The clock values are available for all processors simultaneously, since there is no arbitration for this bus and therefore no contention. The advantage of having a global clock is the assurance of the consistency and reliability of the measurements taken by the processors, since clock times come from a common external reference.

3. Instruction Set, Hardware Level Baseline Experiments

Instruction set and hardware baseline experiments consist of measuring the characteristics of the most basic level of the computer visible to the programmer. This includes measuring execution time of hardware instructions and testing the existence of and time to process interrupts. Interrupt times for FTMP were covered in [Feather et al. 85]. SIFT's interrupts have not been validated yet.

Figure 3-1 illustrates the typical loop for measuring instruction timings. This task reads the global clock and stores the value of the starting time in memory. It then enters the loop where it executes the statement being tested LOOPCOUNT times. After the loop terminates, the global clock is read again and the ending time is stored. The value of LOOPCOUNT, and the number of times the task executes to collect data is controlled by external variables set by other tasks or the experimenter through an interface. This task is set up to test instruction times for all processors. For both SIFT and FTMP, the null loop itself was measured so that the overhead from its execution could be subtracted from the results of the other statements.

```
begin
   data[time] := gclock;
   for i := 1 to LOOPCOUNT do
      begin
        < function to be measured >
      end;
   data[time+1] := gclock;
end
```

Figure 3-1: Basic Task Algorithm

Since the collection loop reads the clock, an experiment to validate clock consistency is first done, followed by measurement of High Level Language (HLL) instructions. HLL instructions were chosen since this is the level visible to the user, thus the experimentation process is simplified. Also, the executive systems are written in HLL. Efficiency of high-level language depends on compiler technology. Instead of statement by statement translations, the compiler may optimize to produce instruction pairs that execute faster than the sum of the single instructions. Thus, instruction pairs were also tested to determine the effects of compiler optimizations. This section will cover the three experiments mentioned above:

- Clock read time.
- High level instruction execution time.
- High level instruction pairs.

3.1 Clock Read Delay

In the clock read experiment, the global clock was tested for consistency. The clock can be used as a measuring tool if reading it produces consistent results, or if variations are predictable. To insure that future experiments using the global clock are valid, repetitive readings are performed. Should the clock be found to be inconsistent, steps should be taken to adjust future experiments [Kong 82].

[Clune 84] measured the time to read FTMP's clock from two triads running standalone and running simultaneously. On FTMP, the processors read from a common global clock located on the memory bus. Thus, simultaneous reads of the clock or system bus accesses can cause contention. The results of these experiments are summarized in Table 3-2. These experiments found FTMP's clock to be a reliable measuring device with little additional delay in the face of bus contention. The resolution of FTMP's clock is 250 microseconds.

In the SIFT experiment, a clock read statement was inserted in the basic task and iterated 100 times. Using the experimental procedure described, 3000 data points were collected. Analysis of the data shows the global clock to be a reliable measuring tool. Clock read results for the three processors used are shown in Table 3-1.

Read Time Clock Delay
Microseconds Per 100 Reads with Overhead

Processor	Microseconds min/max
P1	2855/2856
P2	2855/2856
P3	2857/2860

Table 3-1: SIFT Clock Read Results

As Table 3-1 illustrates, the clock reads differed by only 1 to 3 microseconds within a processor, a negligible amount. As for the variation between processors, the maximum difference was five microseconds. This is an excellent result considering the SIFT processors are only loosely coupled. The difference in clock read time is caused by slightly different processor execution rates.

Summaries of SIFT and FTMP clock read results are shown in Table 3-2. For both machines, the global clock proved to be a reliable measuring device where any delays were predictable and negligible. In comparison to FTMP however, SIFT's clock can measure finer grain of events.

Execution Time for SIFT Clock Read:

100 Iterations of 1 Clock Read = 2856.5 microseconds
With Null Loop Overhead
1 Clock Read = 17.7 microseconds
Without Null Loop Overhead

Execution Time for FTMP Clock Read:[6]

16 Iterations of 5 Clock Reads = 13.99 milliseconds
With Null Loop Overhead
1 Clock Read = 172 microseconds
Without Null Loop Overhead

Table 3-2: Clock Read Results for SIFT and FTMP

3.2 High Level Language Instruction Execution Times

In the second category of baseline experiments, the execution times of various instructions were measured. Since we eventually want to compare two systems, efforts were made to insure that as many of the applicable instructions tested on FTMP were also measured on SIFT. Table 3-3 presents the instructions that were measured on FTMP and SIFT. Since the SIFT architecture does not provide hardware or software support for real or long word (32 bit) data, only integer and boolean data type were tested on this system.

As an overall comparison, the execution speed of SIFT instructions is listed along side FTMP's in Table 3-3 and illustrated in Figure 3-2. Although SIFT requires more time than FTMP in negating variables, it is faster at all other instructions including procedure calls. For example, when executing a boolean "OR" function; SIFT is 219% faster. This disparity is due to the differences in compilers. Whereas SIFT's compiler simply loads the variables into two registers and "OR"s them, FTMP's compiler tests each variable separately and executes code depending on the outcome of the test (i.e. if the first variable is true, it jumps without testing the other). Worst case is when both variables are false: it must test both variables before it can jump. An overall unweighted average (assuming all instructions tested are equally likely) shows that SIFT is 129% faster than FTMP in executing instructions.

Along with simple instructions, execution times for procedure calls were measured for various numbers of parameters. To help visualize the results, Figure 3-3 plots procedure calls against number of parameters. An analysis of Figure 3-3 shows that after some constant overhead, the execution time increases almost linearly with increasing number of parameters. As a comparison, the results of FTMP's

[6]Average of Two reported in [Clune 84]

Instruction Execution Times: SIFT vs. FTMP (In Microseconds Per Instruction, Without Null Loop Overhead)				
Pascal Instruction	Description	SIFT	FTMP	Percent Difference
A := 1	Integer Assign	3.70	4.0	8.1%
A := B	Integer Variable Assign	4.39	5.5	25.3%
A := B + C	Integer Addition	6.45	10.0	55.0%
A := B * C	Integer Multiply	12.57	20.2	60.7%
A := B div C	Integer Division	20.83	21.7	4.2%
A := -B	Integer Negate	9.48	7.0	-26.2%
A := B = C	Integer Compare	8.51	23.2	172.6%
A := B >= C	Integer Compare	9.70	23.5	142.3%
A := B < C	Integer Compare	9.45	21.2	124.3%
A := True	Boolean Assign	3.70	4.0	8.1%
A := B	Boolean Variable Assign	4.39	5.5	25.3%
A := B or C	Boolean Or	6.89	22.0	219.3%
A := B and C	Boolean And	6.89	21.1	206.2%
A := NOT B	Boolean Negate	6.26	10.9	74.12%
NULL	Null Loop	10.86	17.7	63.0%
Procall()	Procedure Call	6.45	37.0	473.6%
Procall(A)	Procedure Call	7.00	51.7	638.6%
Procall(A,B)	Procedure Call	15.88	57.5	262.1%
Procall(A,B,C)	Procedure Call	20.27	63.2	211.8%
Procall(A,B,C,D)	Procedure Call	24.39	69.0	182.9%
If GO then A:=1	Conditional, True	6.95	9.0	29.5%
If GO then A:=1	Conditional, False	3.70	5.5	35.1%
If GO then A:=1 Else B:=1	Conditional, True	8.32	13.2	58.7%
If GO then A:=1 Else B:=1	Conditional, False	7.14	9.5	33.0%

Table 3-3: Instruction Times: SIFT vs. FTMP

experiment are plotted on the same graph. Although FTMP's execution time also increases linearly, it has 474% more initial overhead than SIFT's. Since FTMP is a stack machine, it executes extra instructions that SIFT does not. It must push the number of parameters on the stack before executing a return statement. The return statement must then pop this number of parameters so it can adjust the stack pointer before returning control to the calling program, thereby removing parameters no longer needed.

3.3 High Level Language Instruction Pair Execution Times

In the third category of baseline experiments, the execution times of instruction combinations were measured to determine if the results exceeded the worst case time, the sum for executing each instruction alone. This was an important experiment since in the SIFT operating system the user is responsible for defining the duration of a task: if instruction combinations take longer than expected, the allocated time may prove insufficient and the task will time out. It was also of interest to determine if each system's compiler takes advantage of optimizations. In these experiments, each set of instructions was executed in the basic task. For the SIFT experiments, using the standard procedure, 3000 data points were collected.

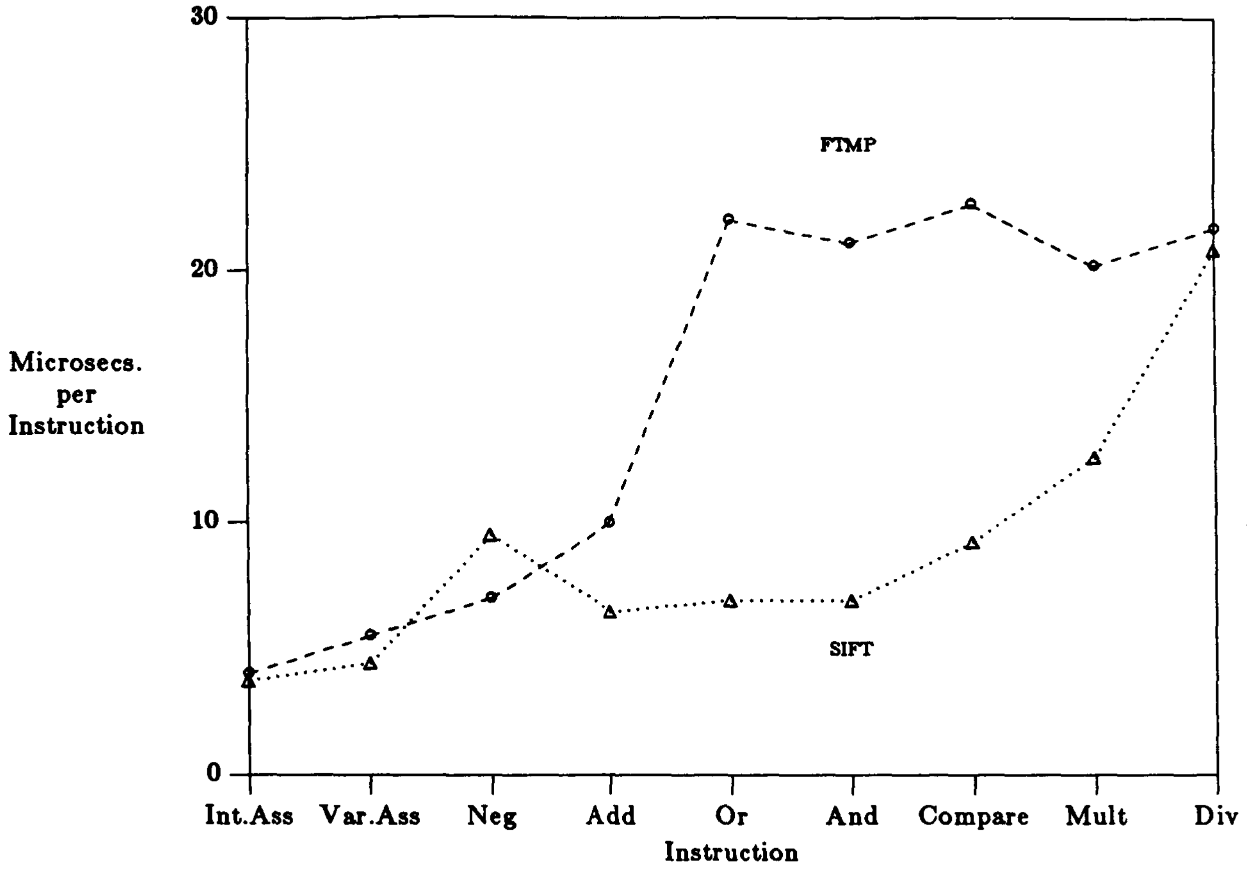


Figure 3-2: Graph of Instruction Times: SIFT vs. FTMP

Two approaches were taken for this experiment. One experiment tested the effect on execution times when the consecutive iteration of a single instruction was increased. For this case, the integer assign statement A=1 was iterated between 1 and 20 times, inside the basic task loop. Figure 3-4 illustrates the results for SIFT and FTMP.

Inspection of Figure 3-4 shows that for SIFT, although execution time increases linearly with the number of iterations, the slope reflects signs of compiler optimization. An analysis of the SIFT assembly code shows that savings occurs because the compiler loads a register with the value 1 the first time and uses stores to assign 1 to A thereafter. For example, a representation of the assemble code for the case where A=1 was executed five times in a row is illustrated in Table 3-4 for both FTMP and SIFT.

In comparison, the results of FTMP's experiment show that although FTMP's graph is also linear, there is no compiler optimization. This result occurs because FTMP is a stack machine and consecutive stores are not done. Consequently, although SIFT and FTMP start off with a similar execution time, by the 20th iteration SIFT is done 94% sooner.

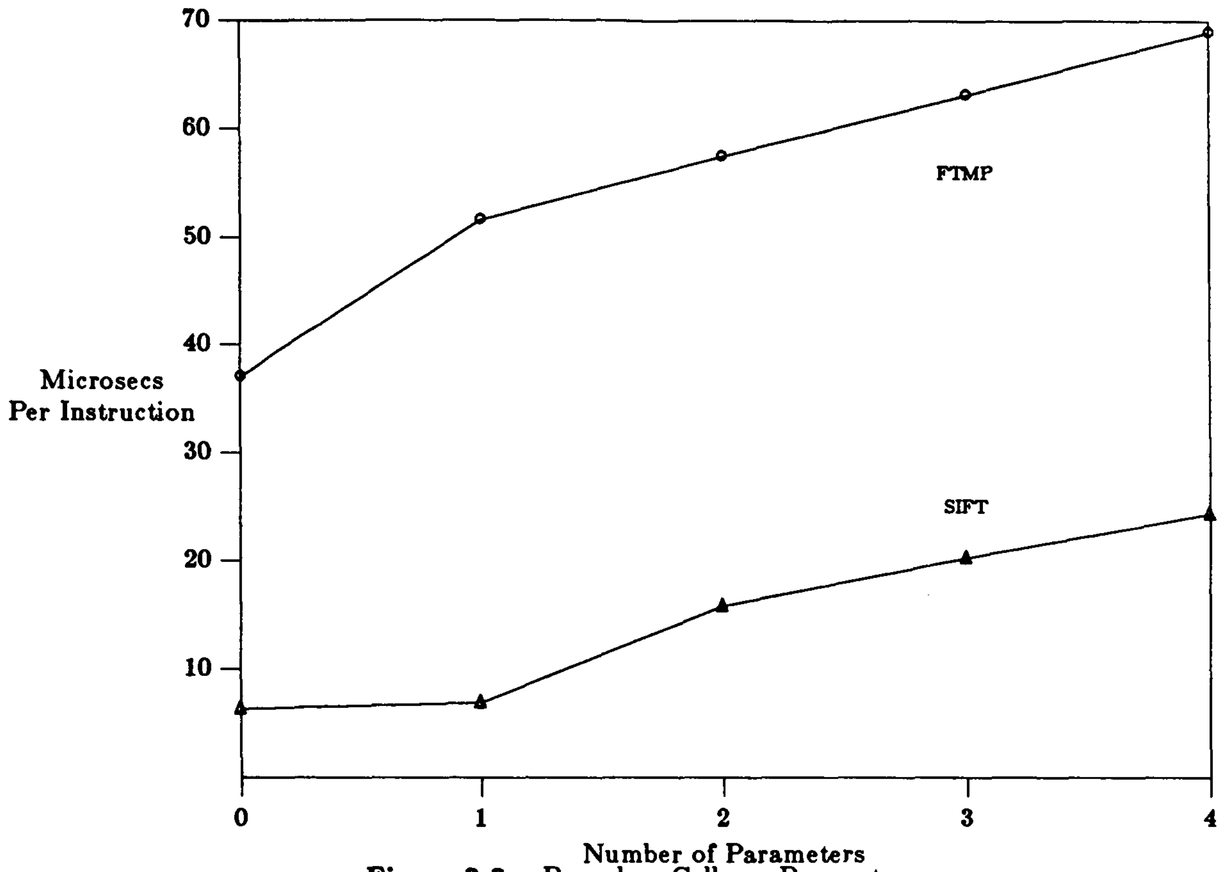


Figure 3-3: Procedure Calls vs. Parameters

In the second set of experiments the execution times of instruction pair and triple combinations were measured. The architectural difference between register and stack machines was witnessed when the instruction combinations performed on FTMP were applied to SIFT. Table 3-6 and Table 3-7 shows the results of these combinations. Table 3-5 is an example illustrating how each machine handles a representative instruction combination.

Comparing the two systems, SIFT's compiler is able to optimize for cases where FTMP's compiler can not, such as register allocation to avoid unnecessary loads and stores. In general, the only optimization FTMP features is a duplicate store: in some cases, if a variable is going to be used twice it is duplicated instead of stored and reloaded.

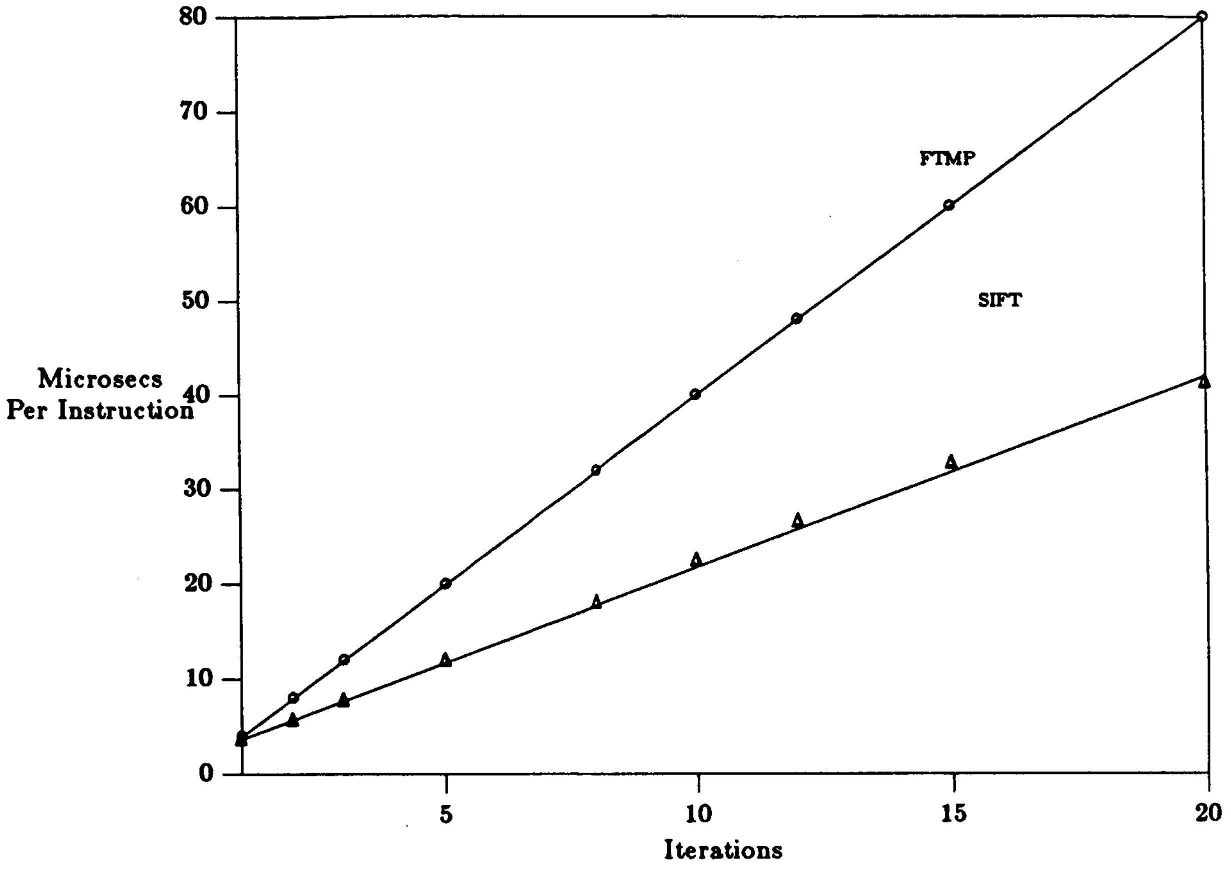


Figure 3-4: A=1 vs. Consecutive Executions

Instruction	SIFT	FTMP
A = 1	Load 1,R1	Push 1
	Store R1,A	Pop A
	Store R1,A	Push 1
	Store R1,A	Pop A
	Store R1,A	Push 1
	Store R1,A	Pop A
		Push 1
		Pop A
		Push 1
		Pop A

Table 3-4: SIFT vs. FTMP for Integer Assign A=1

Instruction	SIFT	FTMP
B = C + D	Load C,R1	Push C
E = C + D	Add D,R1	Push D
	Store R1,B	Add
	Store R1,E	Pop B
		Push C
		Push D
		Add
		Pop E

Table 3-5: SIFT vs. FTMP in Addition Combination

SIFT Combination Comparisons (in microseconds per instruction without overhead)				
Pascal Instruction	Description	Time If Done Separately	Time For Combination	Percent Difference (Separate vs. Combo)

B := 2
C := 2
	Assign Combination	7.40	5.76	28.5%

B := C + D
E := C + D
	Addition Combination	12.90	8.95	44.1%

B := C + D
E := F + A
	Addition Combination	12.90	12.63	2.1%

B := C + D
E := A + B
	Addition Combination	12.90	12.63	2.1%

B := 2
C := B
	Assign Combination	8.09	7.82	3.4%

B := 2
C := D
	Assign Combination	8.09	7.82	3.4%

Table 3-6: SIFT: Comparison Between Single Instructions and Combinations

FTMP Combination Comparisons (in microseconds per instruction without overhead)				
HLL Instruction	Description	Time If Done Separately	Time For Combination	Percent Difference (Separate vs. Combo)

B = 2
C = 2
	Assign Combination	8.0	8.0	0.0%

B = C + D
E = C + D
	Addition Combination	20.0	20.0	0.0%

B = C + D
E = F + A
	Addition Combination	20.0	21.0	-5.0%

B = C + D
E = A + B
	Addition Combination	20.0	21.0	-5.0%

B = C + D
E = B + A
	Addition Combination	20.0	19.5	2.5%

B = 2
C = B
	Assign Combination	9.5	6.5	31.5%

B = 2
C = D
	Assign Combination	9.5	9.5	0.0%

Table 3-7: FTMP: Comparison Between Single Instructions and Combinations

4. Executive Level Baseline Experiments

This section discusses the executive level baseline experiments conducted on FTMP with a comparison of similar experiments conducted on SIFT. Executive level baseline experiments include the behavior of the executive level software (operating system), along with the throughput, utilization, and delay of the operating system software. The experiments in this section include:

- The time to transfer blocks of data to and from local memory for FTMP.
- The actions of the systems when the task is allowed to overrun its allotted time.
- The task iteration rate for both FTMP and SIFT.
- The software overhead for FTMP's and SIFT's real time dispatcher.[7]
- The overhead for FTMP's and SIFT's fault-handling tasks.

4.1 Data Block Transfers

On a multiprocessor system where processors access a common resource, there can be contention for that resource. Experiments to measure the time to access the resource along with the delay and variance caused by contention for the resource should be performed. On FTMP the shared resources are the global memory and the I/O ports. During execution, blocks of data or code are transferred to and from the shared resources by the triads. The remainder of this section describes experiments to measure data block transfer times to and from global memory on FTMP. SIFT is a distributed computer system with no shared or global memory. For this reason a comparable experiment cannot be done on SIFT.

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 83b]

- 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 bus, 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.

[7] SIFT's software overhead was presented in [Palumbo and Butler 85]; their results are summarized here for comparison.

Of these executive primitives two system bus routines were examined. The remaining executive routines could be validated in a similar fashion as the bus service routines measured.

4.1.1 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 by placing the function inside a loop, similar to the high level language instructions. The size of the block transfers was varied from 1 to 200 words[8] (1 word = 16 bits). The number of processor triads competing for the system bus was also varied: one triad running to show execution time with no contention for the bus, and two triads running to show the effects of bus contention.

4.1.2 Results of Block Transfers

Figures 4-1, 4-2, and 4-3 show the execution times of reads and writes to be linear with respect to the size of the data block being transferred. On these figures, the horizontal axis indicates the size of the block transfer in 16 bit words. The vertical axis is the average time to transfer the block of data. 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.

Of interest is the variation within each group of data and between the two groups of data. With one triad, no bus contention, the measured times were all within two clock ticks.[9] With two triads competing for the bus, most of the measured times were within eight clock ticks, with a few outliers up to 20 clock ticks (5 milliseconds) from the mean. (The number of clock ticks to complete a transfer task, 50 transfers per task, varied from 31, for a one word write, to 248, for a 200 word write.) This bus contention is illustrated in Figures 4-2 and 4-3, showing the increase in the 95% confidence intervals.

A final word regarding system bus service routines comes from the FTMP Executive Summary [Draper 84]. In the summary, the authors noted the bottleneck caused by system bus access (greater than 150 μseconds overhead per access, as shown on the graphs, which equals about 15 high level language instructions) lowered the expected throughput of the system. [Draper 84] 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.

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

[9] A clock tick is one period of the system clock, or the observable resolution of the clock. FTMP's clock period was 250 micro-seconds.

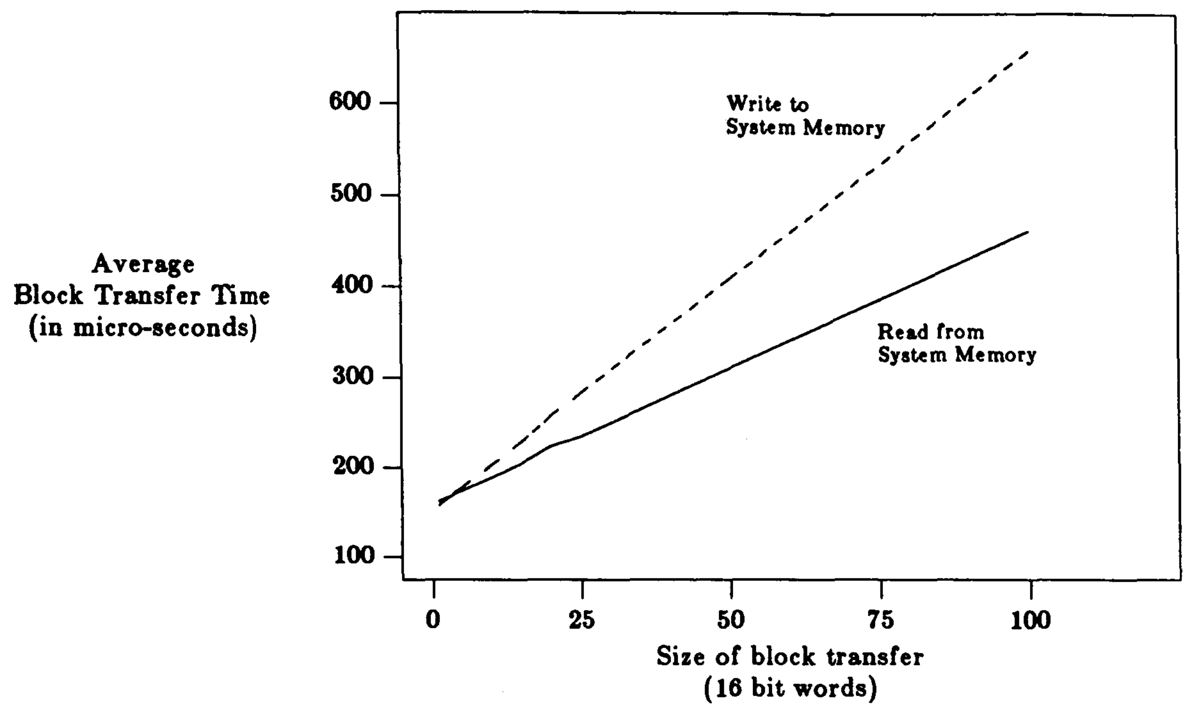


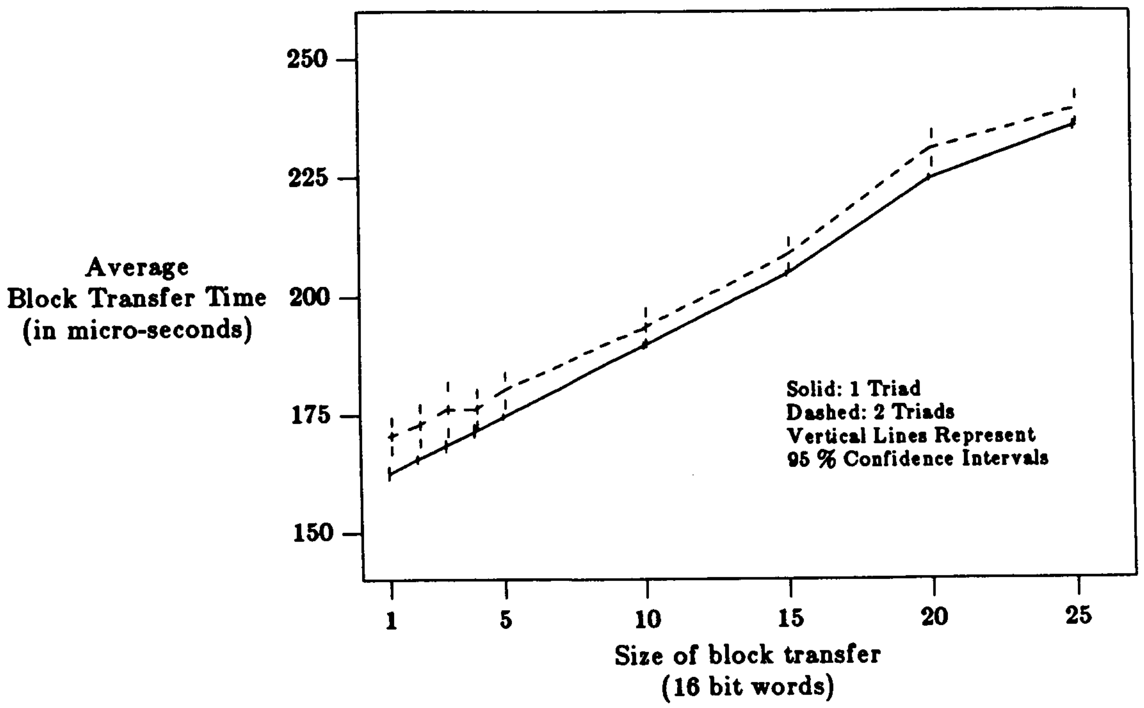
Figure 4-1: Read and Write Execution Times as a Function of Block Size, 1 Triad

4.2 Real Time Task Behavior

In real time systems, application tasks are executed at a fixed iteration rate or frequency to meet hard deadlines. Since periodic task execution is an integral part of the system, experiments must be devised to validate the basic properties of task iteration rates. First, the basic task iteration rate should be validated. Next, the behavior of these iteration rates in the presence of errors or "malicious" tasks should be validated. The type of errors that can occur depend in a large part on the scheduling mechanism of the particular real time system. On SIFT, tasks are dispatched by a fixed schedule, as discussed in Section 2.3. Tasks are divided into 1.6 millisecond slots, assigned at compile time. Thus, the only malicious properties are a task broadcasting bad data or a task exceeding its slot allocation.

In contrast, FTMP uses a dynamic scheduling mechanism as discussed in Section 2.2. Tasks are grouped by execution rates. All of the tasks in a rate group must execute within a frame. FTMP has three frame sizes which define the task grouping. Processors select tasks from the appropriate global task queues in each frame group. When all higher frame tasks are exhausted, processors select tasks from the next lower rate group. Thus, the effect a malicious task has on individual tasks and frames should be tested.

The remainder of this section discusses the experiments used to measure the basic properties of the task iteration rate on FTMP and SIFT.



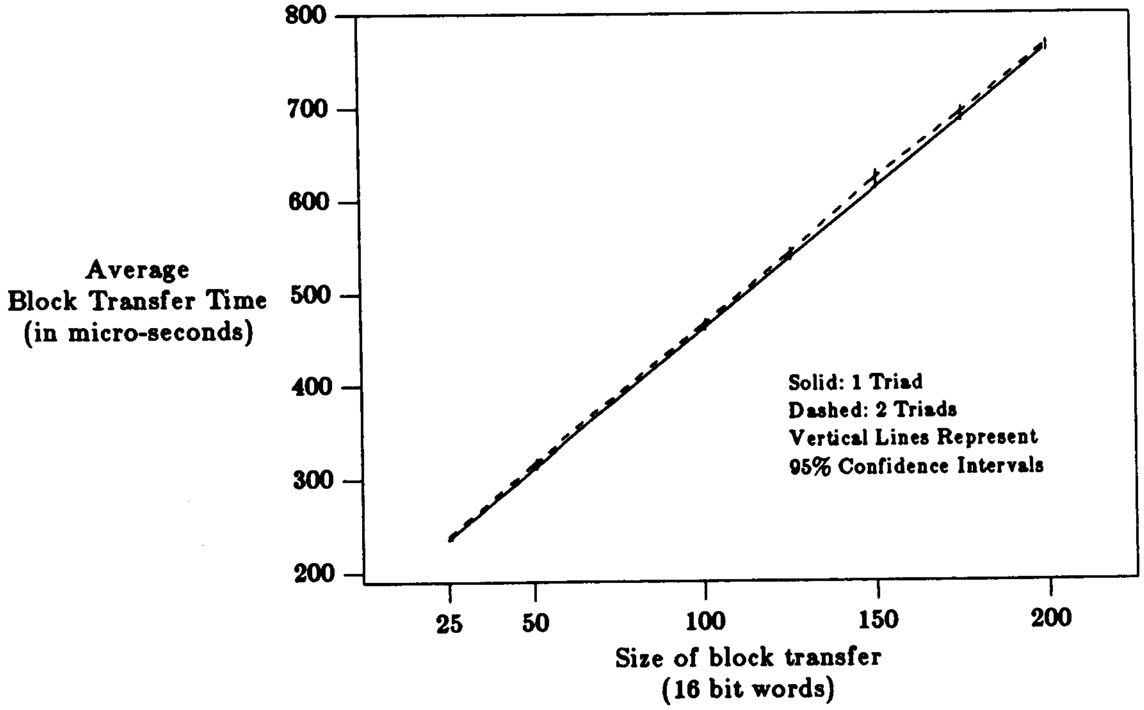
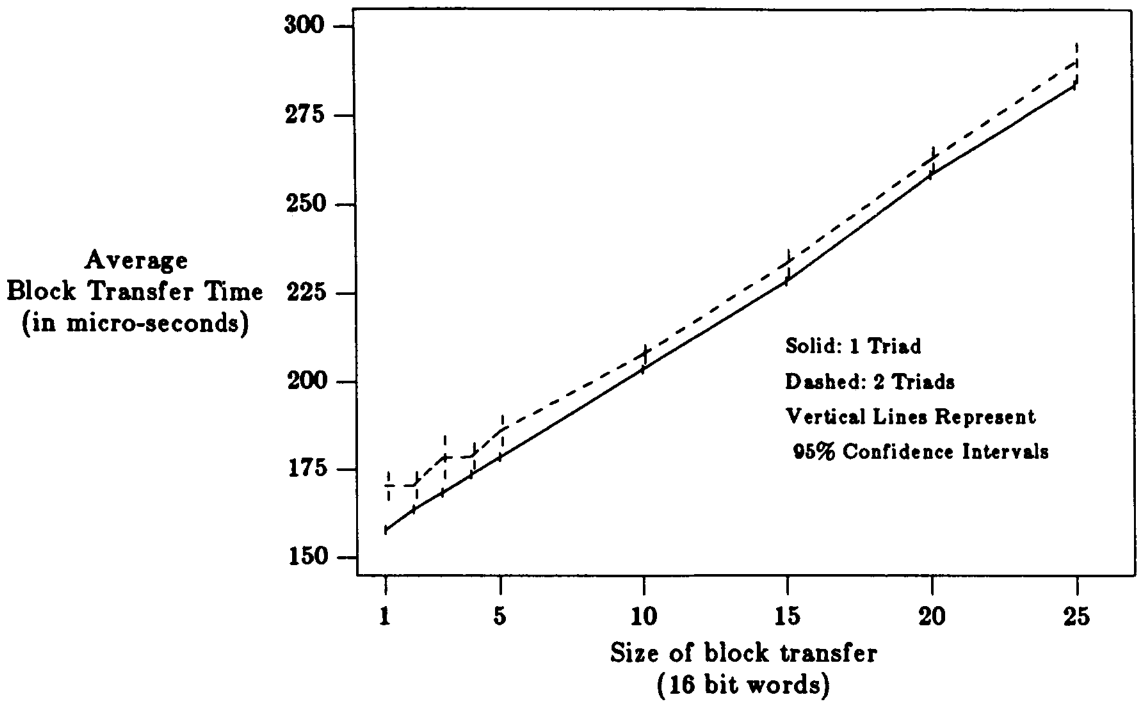


Figure 4-2: Read Execution Times as a Function of Block Size, 1 Triad and 2 Triads



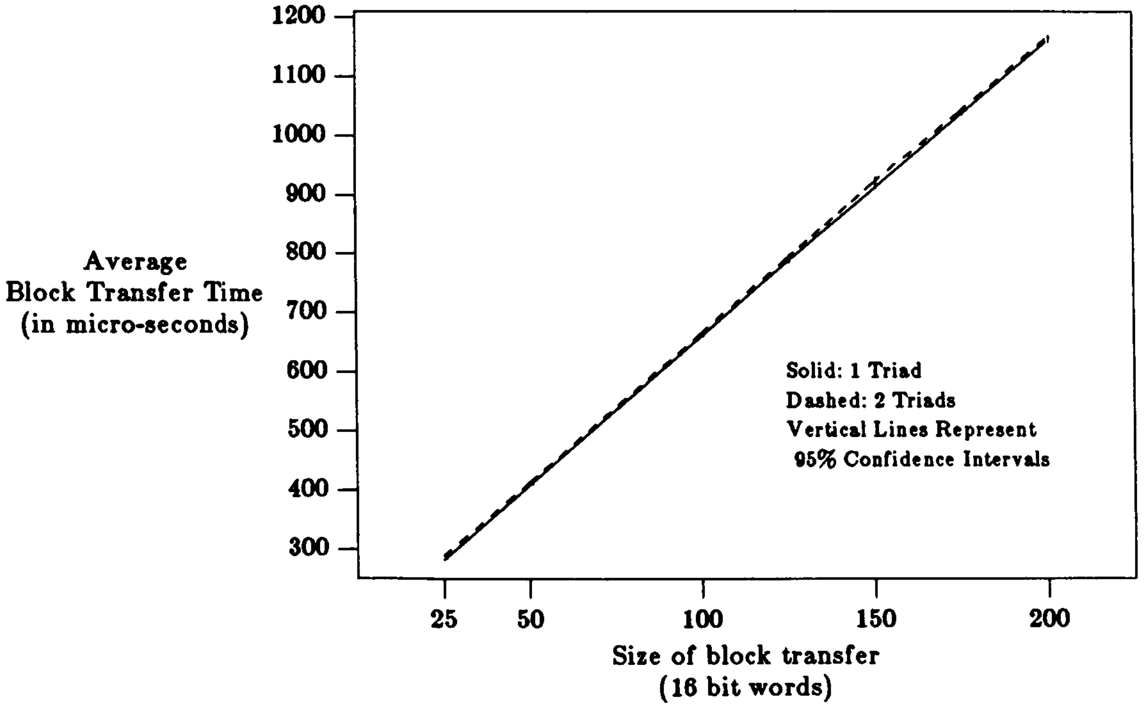


Figure 4-3: Write Execution Times as a Function of Block Size, 1 Triad and 2 Triads

4.2.1 FTMP Frame Sizes

This experiment involves measuring the true 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 are presented in Figures 4-4 and 4-5.

Figure 4-4 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 R4 pattern determines the pattern for the R3 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 observed 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. These patterns were observed in the unprocessed data dumps. Similarly for the three triad case the observed R4 pattern is 40, 65, 105, 40, ... Thus the R3 pattern would be 105, 145,... This pattern differs from the observed pattern, a single 125 milliseconds group, but the difference could be explained by additional overheads occurring in the scheduler.

These patterns are probably caused by the variation in scheduler workload between tasks. (i.e. scheduling and I/O for R3 and R1 tasks, etc.) Further study of the dispatcher will be necessary to fully characterize its behavior.

The long frames are in correlation with the long scheduler times (Section 4.3.1.1, Figure 4-7). The scheduler 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 Figures 4-7 and 4-4 represent this behavior best:[10] as shown in Figure 4-7 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 scheduler times. From the frame size measurements, a problem with the FTMP scheduler meeting its real time constraints can be observed.

4.2.2 SIFT Frame Size

In SIFT, the schedule table dispatches tasks periodically. This period is called a "major frame" and is divided into 1.6 millisecond time slots [Palumbo and Butler 85]. The purpose of this experiment was to validate the true size of these slots which consequently validates the frame size. To obtain these results, three experiments were conducted.

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

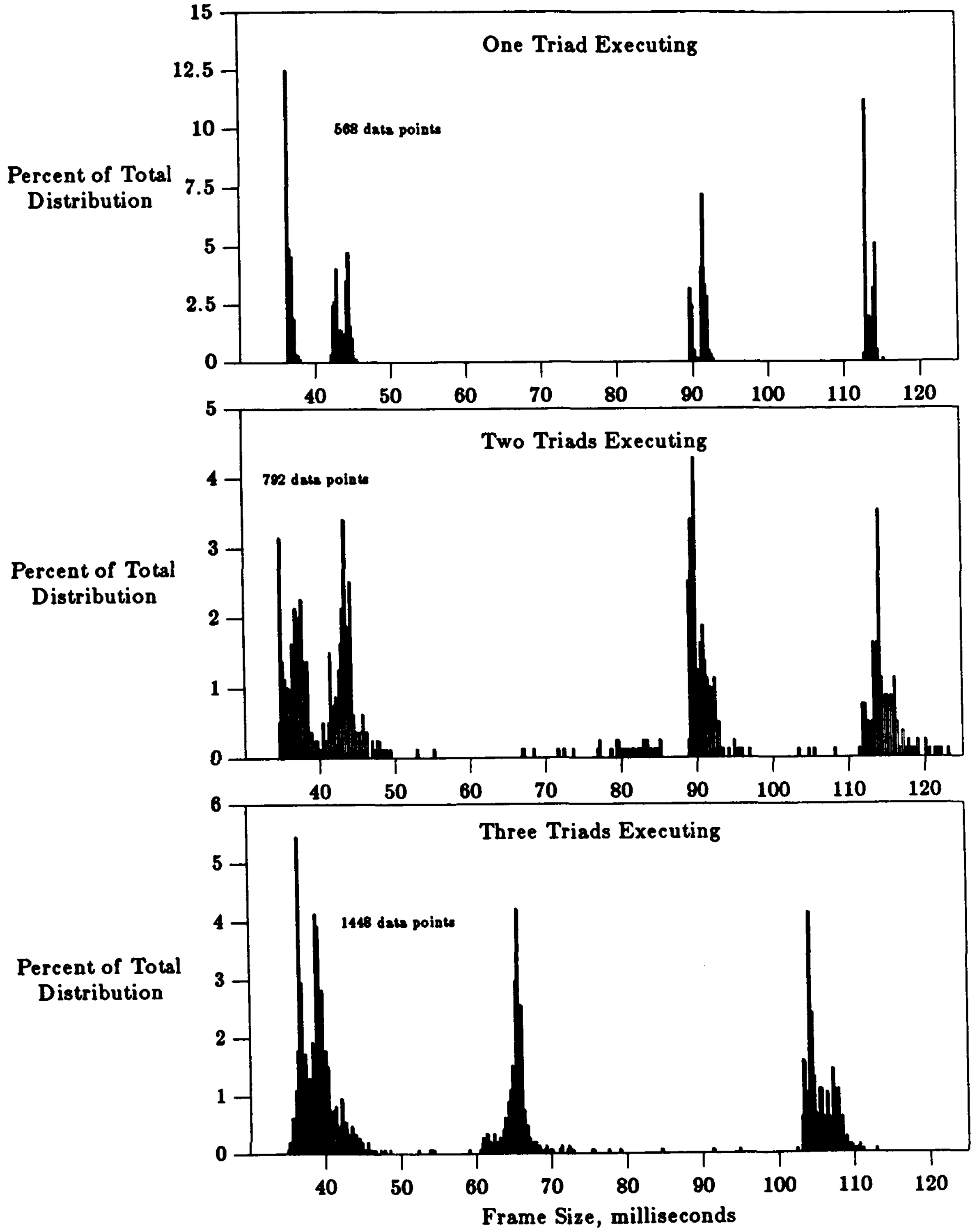


Figure 4-4: Actual Frame Size for R4 Tasks

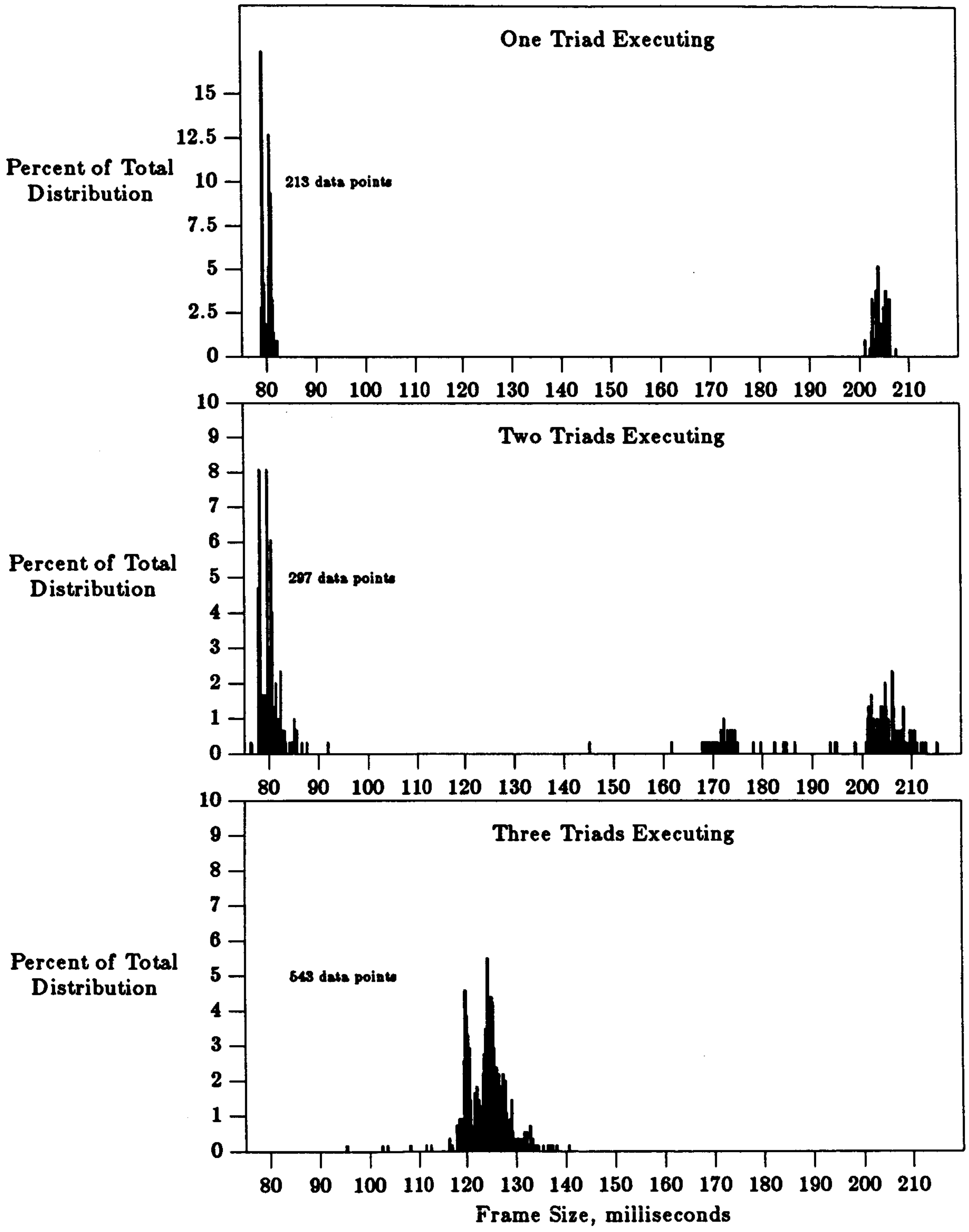


Figure 4-5: Actual Frame Size for R3 Tasks

In the first experiment, the time between consecutive tasks was measured by reading the clock upon entering each task. This delta-time validated the true time for a slot, 1.6 milliseconds. In the second experiment, the same two tasks were consecutively run. However, this time the tasks were allocated a duration time of two slots per task. In the last experiment, three tasks were run back to back and the time was taken upon entering the first task and entering the second task. These experiments showed that the slots sizes were consistently 1.6 milliseconds.

4.2.3 Comparison of FTMP's and SIFT's Frame Sizes

In comparing the frame sizes for both systems, SIFT uses a timer interrupt, occurring at a fixed rate, to signal the start of the slot. FTMP also uses a timer interrupt, but the interrupt is software controlled and armed by the scheduler to allow for frame slippage or software extension of the frame. Hence SIFT's frames are constant in size, whereas FTMP's vary because of slipping caused by large and varying overhead in the scheduler (Section 4.3.1.1.) From these results the advantage of using a fixed timer interrupt, rather than a software maskable interrupt, for real time control is demonstrated.

The next section discusses performance of FTMP and SIFT's task stretching mechanisms.

4.2.4 FTMP Frame Stretching

As discussed in Section 2.2, FTMP has three task rate groups or frame sizes: R4 (25 Hz), R3 (12.5 Hz), and R1 (3.125 Hz). Individual tasks in the frame are given time limits and any task that exceeds its limit is aborted. All tasks in a rate group should finish before the next frame. FTMP has a mechanism for handling frame slippage, i.e. tasks in a frame taking longer than scheduled. If all tasks in a frame are not completed before the end of the frame, the frame is extended, or *stretched*, to give tasks more time. [Clune 84] ran two experiments: the first experiment determined the effects of an individual task taking longer than the frame size. [Clune 84] discovered the start of the next frame will be delayed without limit, until all tasks in the present frame are completed. However, the task will abort if it exceeds its time limit. The second experiment determined the effect of a frame trying to execute more tasks than could be completed in the frame's time limit. In this experiment, [Clune 84] created an infinite task queue[11] and showed the frame will slip forever.

Next, the effect of a processor broadcasting bad data was investigated. On FTMP, processors run the same program in lockstep and voting is accomplished in hardware. Therefore, having one of these processors run incorrectly to test correct voting and reconfiguration is not feasible. However, [Draper 83a] reported the results of fault injection experiments which verify that a faulty processor will be configured out.

[11] An infinite task queue was created by dynamicly changing pointers with the test adapter.

4.2.5 SIFT Frame Stretching

In the SIFT system, as long as a task does not exceed its time allocation and a processor does not broadcast bad data, a processor is considered healthy. If either condition is violated, the processor will be tagged "faulty" and be configured out. The purpose of task stretching was to determine whether a faulty processor will be reconfigured out.

To answer this, two experiments were done. In the first experiment, conditions for task timeout, a task not meeting its time allocation, were explored. Processor 1 and 3 were allowed to complete their task before the deadline, while processor 2 stretched its task beyond the allocated time. For this experiment, the system behaved as predicted--the straying processor was halted when its task took longer than the scheduler allowed.

In the second experiment, the system's reaction to the broadcast of bad data was explored. For this experiment, two methods were used to make a processor appear faulty: a task was stretched, preventing the broadcast of correct data and, correct data was allowed to be broadcast, but the task was stretched beyond its allocation. The first method was used to create the broadcasting of bad data. The second method explored the reaction of the system when a processor became faulty but managed to pass valid data. Figure 4-6 shows the code used for this experiment.

```
begin
   data[time] := gclock;
   for i := 1 to LOOPCOUNT do
      begin
         if i = WHEN then
            stobroadcast(passit,16#ABCD);
         if pid = 2 then
            for j := 1 to STRETCH do
               extraloop := j;
      end;
   data[time+1] := gclock;
end
```

Figure 4-6: Program Used For Task Stretching - Voting Case

To fully control the two conditions which could trigger a configuration, two variables STRETCH and WHEN were introduced. STRETCH controls the amount the task running on processor 2 is lengthened, or stretched. WHEN signals processors 1, 2, and 3 to broadcast hex value ABCD, arbitrarily chosen data. To test the full effects of task manipulation, STRETCH and WHEN were varied from 1 to 100.

The results of this experiment were as predicted. As Table 4-1 illustrates, when STRETCH was increased beyond 7, the process running on P2 was stretched beyond its time constraints. Because its task did not finish, it was forced to broadcast bad data and was therefore configured out. It was also configured out when STRETCH was increased beyond 7 but WHEN was small. This proved that although P2 had the opportunity to pass correct data, it was configured out because its task did not meet its time allocation.

SIFT Task Stretching Results (Task Execution Time in Milliseconds)					
WHEN	STRETCH	Task Execution Time (max)			P2 Reconfigured
		P1	P2	P3	Out
1	5	2.36	9.77	2.31	No
10	5	2.36	9.78	2.31	No
50	5	2.38	9.80	2.35	No
100	5	2.36	9.76	2.31	No
1	7	2.36	12.32	2.31	No
10	7	2.36	12.33	2.31	No
50	7	2.38	12.35	2.35	No
100	7	2.36	12.31	2.31	No
1	8	2.36	timed out	2.31	Yes
10	8	2.36	timed out	2.31	Yes
50	8	2.38	timed out	2.35	Yes
100	8	2.36	timed out	2.31	Yes
1	10	2.36	timed out	2.31	Yes
10	10	2.36	timed out	2.31	Yes
50	10	2.38	timed out	2.35	Yes
100	10	2.36	timed out	2.31	Yes
1	100	3.77	timed out	3.70	Yes
10	100	3.78	timed out	3.71	Yes
50	100	3.82	timed out	3.76	Yes
100	100	3.76	timed out	3.70	Yes

Table 4-1: SIFT: Task Stretching Results

The results of task stretching experiments proved that SIFT handles "malicious" processors exactly as predicted: if a task does not complete before its allocation of 1.6 millisecond time frames, or if it broadcasts bad data, it will be configured out of the system.

4.3 Real Time Scheduler Overhead

The behavior and delay of the executive software are two more elements of the Evaluation Matrix of Table 2-1. This section covers the Dispatcher-Scheduler software overhead. The dispatcher-scheduler strategy was described in Section 2.2 for FTMP and in Section 2.3 for SIFT.

4.3.1 FTMP's Real Time Scheduler Behavior

The following experiments measured the software overhead of the dispatcher-scheduler tasks for the FTMP operating system:

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

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

4.3.1.1 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[12] , kicks the other triads to start R4 tasks, and does necessary I/O. At some point the dispatcher activates[13] the application task. The behavior and execution time of the dispatcher from the interrupt to the start of the first application task was measured. This time is spent scheduling lower priority events, starting other processors in the frame, executing reconfiguration commands and doing necessary task I/O.

The synthetic workload generator provides a means to measure execution times and intertask 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 both the real time clock at the start of its execution and the absolute time of the frame start which is 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 are given in Figure 4-7. As seen from these graphs, the data is grouped into three regions. The dispatch start up 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 variation in scheduler workload between tasks. (i.e. scheduling and I/O for R3 and R1 tasks, etc.) 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 cause missed or late data, allowing the real time system to miss deadlines. (Data on the variation between frame starts and variation between tasks starts was presented in Section 4.2.1.) One-half of the time the dispatcher takes a full R4 frame or longer to execute, clearly an implementation problem.

[12] 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.

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

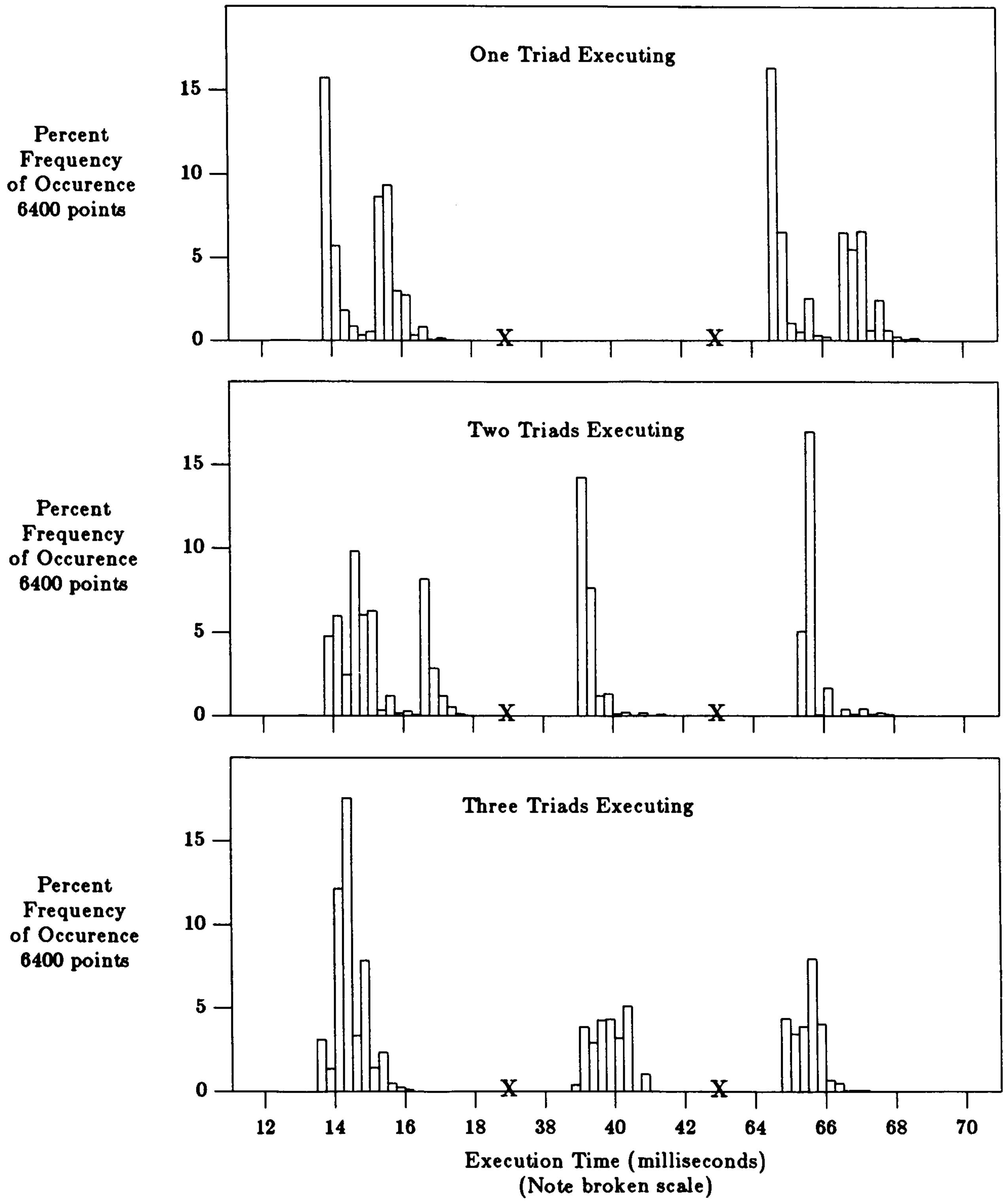


Figure 4-7: R4 Dispatcher Execution Times, 1, 2, and 3 Triads

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 similarly. It was also noted that the frame starts occurred at 250 msec. with no variation[14] (i.e. the interrupt mechanism works correctly).

4.3.1.2 IPC 'Kick' Times

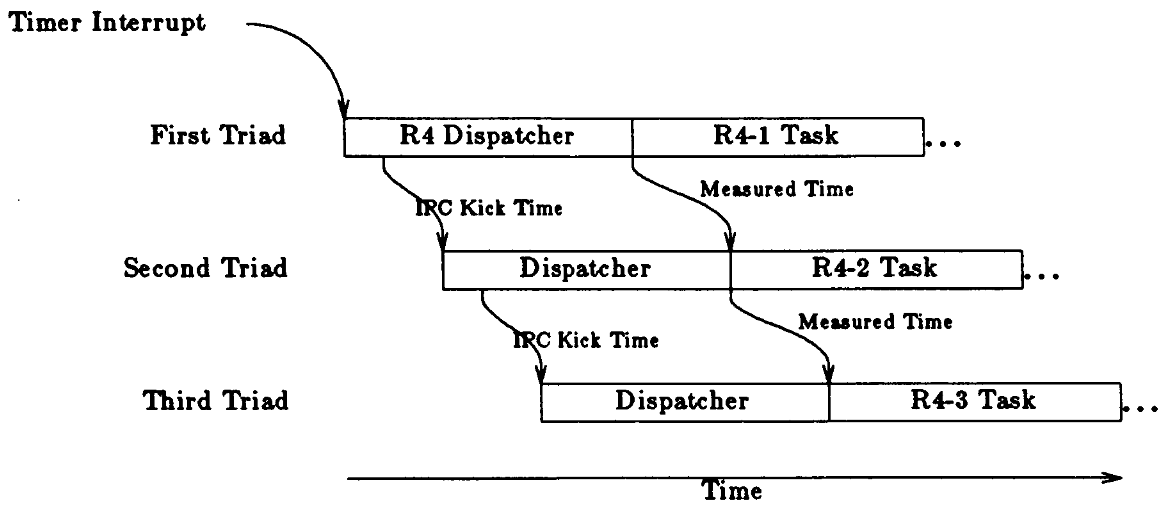


Figure 4-8: IPC 'Kicks' Timing Diagram

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 4-8. 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 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 4-8

[14]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.

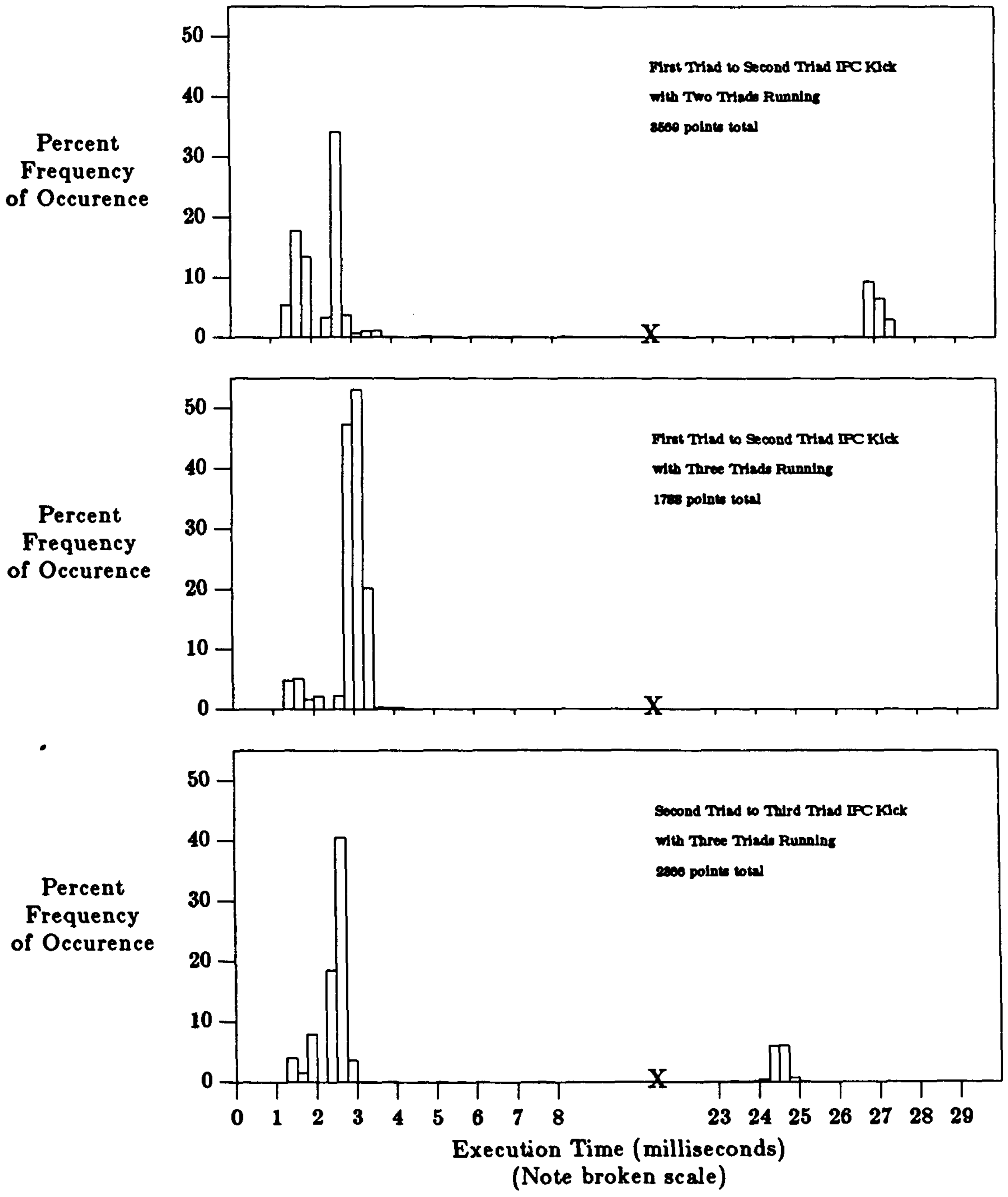


Figure 4-9: Time and Variation Between the Starts of the Application Tasks on Different Processor Triads. Emulating IPC Kick Times

Histograms for the IPC 'Kicks' are given in Figure 4-9. 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 3.0 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 locked 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.

4.3.1.3 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 as shown in Figure 2-3.

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 4-10.

As seen from the data the behavior is regular, with skewing as the number of executing triads increases. This skewing is probably caused by bus access contention, measured in Section 4.1. 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 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.[15] 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.

[15]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.

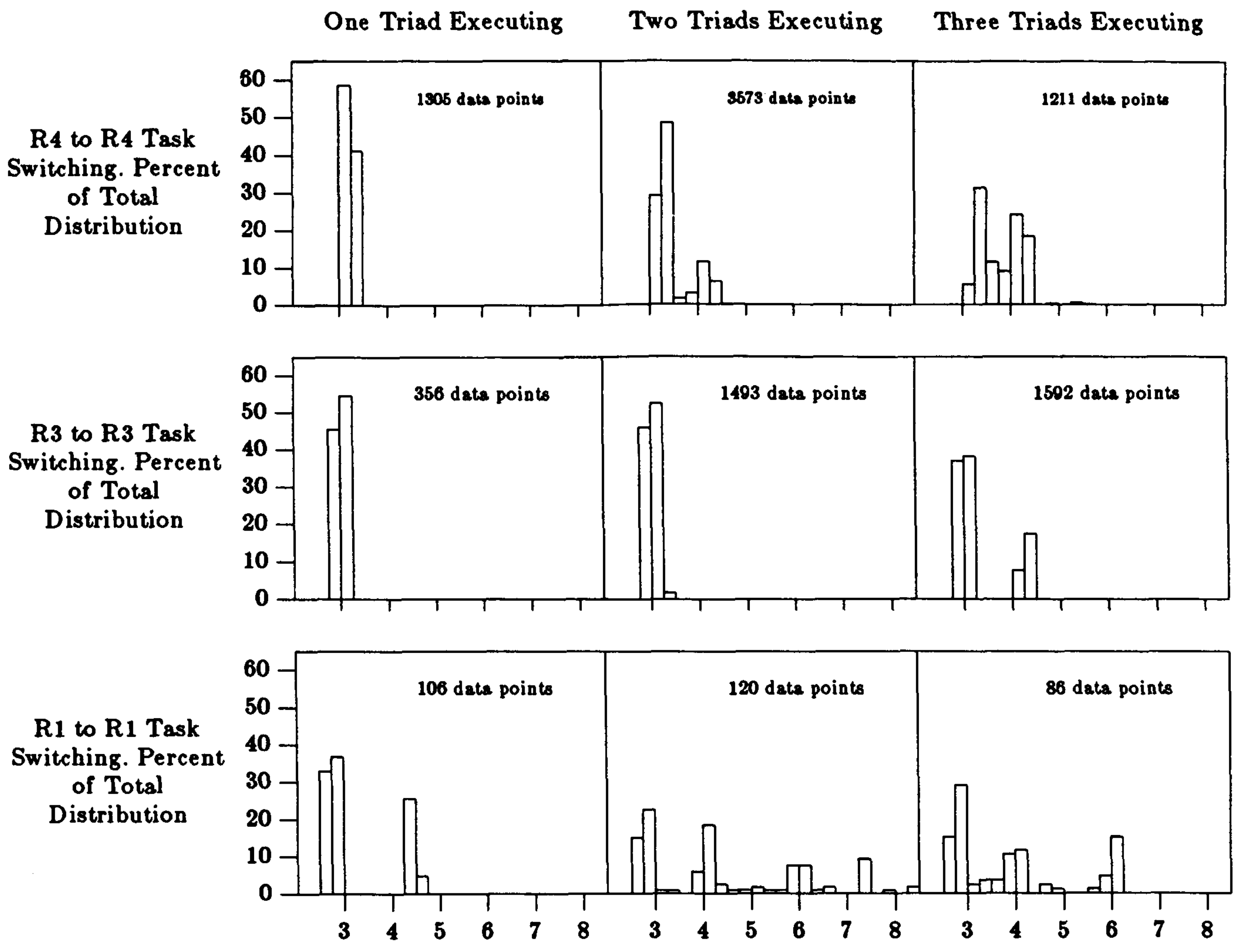


Figure 4-10: Intra-Task Group Switching Times in Milliseconds

4.3.1.4 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 2-3

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.[16] (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.)

[16]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.

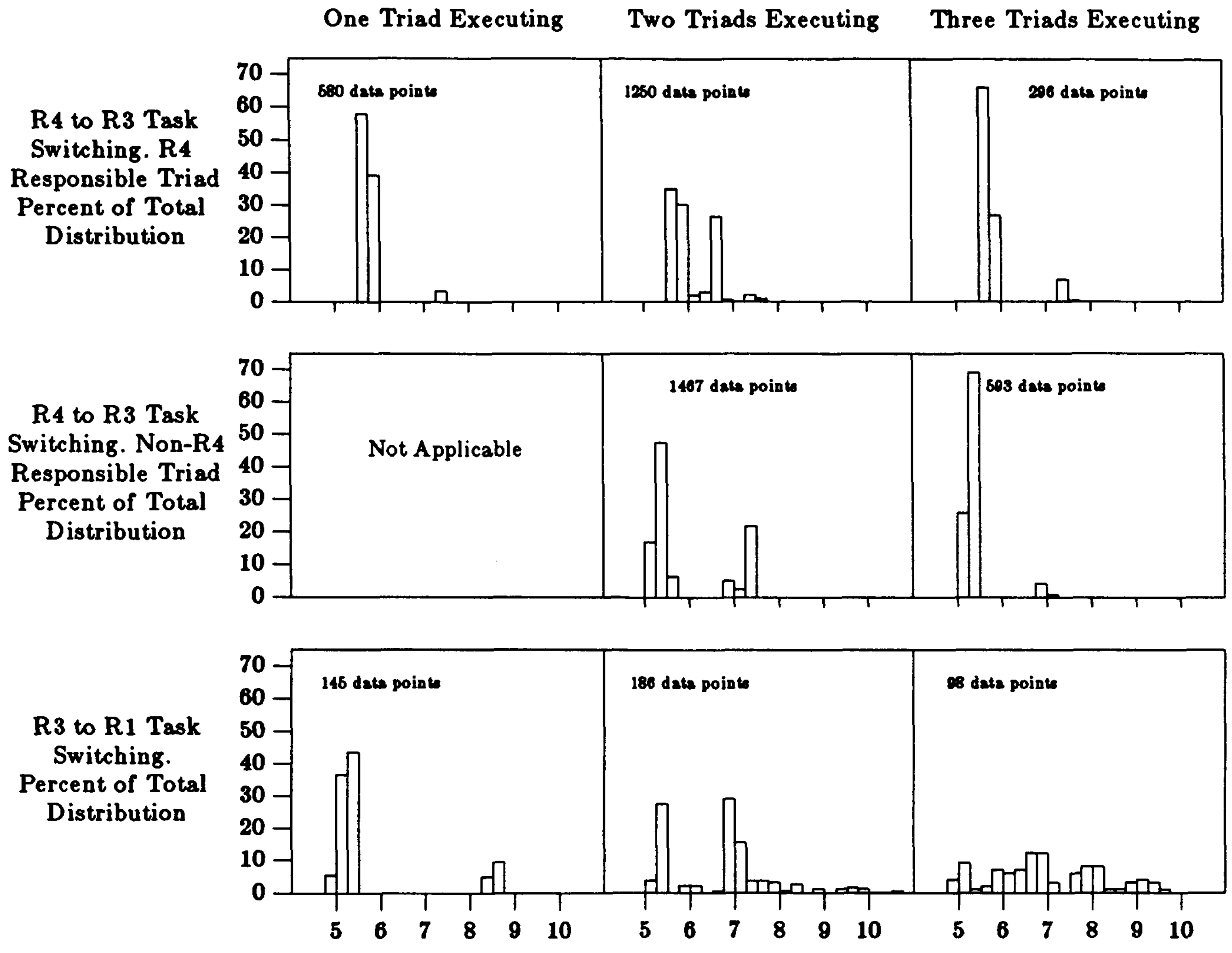


Figure 4-11: Inter-Task Group Switching Times in Milliseconds

The behavior of these three processes are summarized in Figure 4-11. 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. 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.

4.3.1.5 FTMP Dispatcher-Scheduler Software Overhead Summary

This section estimates the useable throughput of the FTMP system from the dispatcher-scheduler overheads measured in the previous sections. Table 4-2 gives a breakdown of the available throughput and overheads observed. In Table 4-2, 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 4-2: Performance Estimates for FTMP

In Table 4-2, the "As Designed" 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 4-7. This shows 63% of the available throughput is spent in the dispatcher. The "As Running" columns of the table use the actual frame sizes as measured, instead of the 40 millisecond R4 frame size assumption and use the true dispatcher times measured in this report, Figure 4-7. The overhead percents are lowered but the frames are extended, showing the present behavior of the system. This behavior is inappropriate for a real time system with a 25 Hz iteration rate.

The FTMP Executive Summary, [Draper 84], noted the large bottleneck caused by the long bus access times (Section 4.1). 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 lowered 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.[17] The "As Corrected" column of Table

[17] 33% of the dispatcher is reduced 88% for a 29% total reduction.

4-2 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.

4.3.2 SIFT's Real Time Scheduler Behavior

The overhead for SIFT's Operating System was presented in [Palumbo and Butler 85]. This section will consider the overhead involved for the real time dispatcher. The results from [Palumbo and Butler 85][18] will be generalized to provide a parallelism between FTMP and SIFT.

SIFT's real time dispatcher is fully distributed with no master controller. Application tasks or fault-handling tasks are dispatched by the local executive in each processor. The local executive is responsible for the timely dispatching of tasks and the voting of data according to the task and vote schedule. The dispatching of the tasks is the only part of SIFT's operating system considered under the real time dispatcher. [Palumbo and Butler 85] determined the dispatching of tasks to be nominally 270 μseconds per subframe per processor or 16.9% of a 1.6 millisecond subframe.

4.3.3 Comparison of FTMP and SIFT Scheduler

In comparing SIFT's and FTMP's real time dispatcher, SIFT's dispatcher is more efficient. When comparing the two real time dispatchers, the reason for the large difference in overhead include:

- SIFT's dispatcher uses fixed task tables and application code stored in each processor's local memory, whereas each FTMP triad needs to process the task queue and application code stored in global memory which require multiple bus accesses.
- The flexibility of FTMP's dispatcher (e.g. same dispatcher and task queues regardless of configuration) requires more processing than the fixed tables used in SIFT's dispatcher.
- FTMP's dispatcher disables the timer interrupts hence allowing the dispatcher to fail to return control to application tasks. Furthermore, the disabling of the timer interrupt allows the frame to stretch possibly forever. [Clune 84]
- FTMP's dispatcher could be improved by using fixed schedule tables and decreasing the amount of traffic on the system bus.
- SIFT's dispatch overhead while small is a significant percentage of the subframe. If the subframe size were increased, the dispatch overhead would be reduced but the distribution of tasks may not efficiently fit into the larger frame size.

[18] [Palumbo and Butler 85] presented overhead times for three versions of SIFT's operating system, this report will only refer to the latest, optimized, version of the operating system, version V.

4.4 Fault-Handling Software Overhead

For both fault-tolerant systems, software tasks must be executed to maintain system integrity in the presence of faults. The following sections measure the software overhead required to attain fault tolerance for FTMP and SIFT.

4.4.1 FTMP Software Overhead for Fault Detection and Isolation

The final FTMP experiment presented in this report measured the software overhead for fault detection and isolation. The FTMP software tasks for these include READALL and the System Configuration Controller, SCC. READALL incrementally reads system memory to assure all copies of the data in the memory elements are the same. SCC is described in Section 2.2. 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 in the R4 dispatcher. These include the execution of reconfiguration and retire commands. This experiment will not take these overheads into account. The results for this experiment are presented in Table 4-3.

FTMP's Fault Handling Software Overhead (All times in milliseconds)			
Task	Execution Time	Range	Percent of Sysetm Throughput
SCC	43.4 ± 23.3	3.5 → 153.	4.5%
READALL	3.1 ± 0.9	1.25 → 6.25	0.3%

Table 4-3: FTMP Software Overhead for Fault-Handling

The variation in the READALL times is due to bus contention and different sections of code being executed. 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 a small percentage of total system throughput, 4.8%, in comparison to the real time scheduling overheads for FTMP.

4.4.2 SIFT Software Overhead for Fault Detection and Isolation

The functions required to maintain system integrity in SIFT include:

1. Clock synchronization to limit the clock skews between processors.
2. Error task to broadcast error information to all processors.
3. Fault isolation task to determine the faulty device.
4. Reconfiguration task to reconfigure the system upon detection of a fault.

5. Interactive consistency task to replicate external data to all processors.
6. Voting of processed data to maintain integrity.

The first five tasks are executed at the global executive or major frame level (equivalent to an application task.) The last task, voting, occurs at the local executive level inside a subframe. The analysis of SIFT's fault-handling software overhead from [Palumbo and Butler 85] will be divided along this break: major frame vs. subframe.

At the global executive or major frame level, the fault-handling tasks are executed once per major frame with the exception of the interactive consistency task. The interactive consistency task is executed once per iteration of the application tasks.[19] All of these tasks take a constant number of subframes to execute and their overhead is tallied in Table 4-4.

SIFT's Fault Handling Software Overhead				
Task	Execution Time (millisec.)	Execution Time (1.6ms subframe)	Overhead in 44.8ms single frame	Overhead in 108.8ms triple frame
Clock Synchronization	2.7	2	7.1%	2.9%
Error Task	1.0	2	7.1%	2.9%
Fault Isolation	3.1	2	7.1%	2.9%
Reconfiguration	3.1	2	7.1%	2.9%
Interactive Consistency	13.1	10	35.7%	44.1%
Total	23.0	18	64.3%	55.9%

Table 4-4: SIFT's Fault-Handling Software Overhead [Palumbo and Butler 85]

At the local executive or subframe level the fault-handling task is voting. The data voted are determined from a vote schedule constructed at the same time as the task schedule is created. All data are voted by each processor to assure consistent data on every processor and to avoid transferring of data during system reconfiguration.

The overhead involved for voting is a function of the number of words voted, the type of vote (three or five way), and the number of working processors in the system. The time to vote can be approximated by a linear function of the number of words voted with a constant bias or overhead determined by the other parameters. These times are summed in Table 4-5. In order to estimate the overhead involved with voting at a major frame level, four words are assumed an average vote (five way) in every second frame with the resulting overhead of 23.6% per subframe.

[19] When application tasks are executed once per major frame, the major frame is called a single frame. Similarly, when the application tasks are executed three times in a major frame, the major frame is called a triple frame.

SIFT Vote Times		
Type of Vote	Initial Overhead ms.	Vote Time per Buffer ms.
3 way	0.245	0.079
5 way	0.328	0.107

Table 4-5: SIFT's Vote Times [Palumbo and Butler 85]

Combining the result of the global executive and local executive overhead, the total overhead involved per major frame is summarized in Table 4-6.

SIFT's Fault Handling Overhead Summary		
Operating System Level	Overhead in 44.8ms Single Frame	Overhead in 108.8 Triple Frame
Global Executive	64.3%	55.9%
Local Executive	23.6%	23.6%
Total Software Overhead	72.7%	66.3%

Table 4-6: SIFT Software Overhead for Fault-Handling Summary

4.4.3 Comparison of FTMP and SIFT for Fault-Handling Software Overhead

In comparing FTMP and SIFT's software overhead for fault-handling, the main difference lies in the architecture. SIFT does most of the fault-handling functions in software, hence a large software overhead, whereas FTMP accomplishes most of its fault-handling in hardware. Differences between the two systems in achieving their fault tolerance and the large difference in overhead stems from the following sources:

- Clock synchronization for SIFT is achieved in software whereas FTMP maintains synchronization with hardware voting of the clock signals.
- Voting for SIFT is completed in software according to a fixed schedule table; hence a limited quantity of words are voted. FTMP votes during a system bus access; words communicated between system memory or system bus devices (I/O ports) and local memory are voted upon. The number of words voted in FTMP is larger but does not result in additional software overhead.
- Interactive Consistency is the largest overhead in SIFT. Interactive consistency involves the replication of external data to all processors of the system. The overhead is proportional to the number of application tasks and the number of words required for the tasks. Hence a faster processor will reduce the time to achieve data consistency, but the data requirements would also increase. This overhead cannot be reduced significantly without hardware

support, [Palumbo and Butler 85]. Although interactive consistency was not considered part of FTMP overhead, the function is performed in software with hardware support (voters). FTMP accomplishes interactive consistency by reading the external data in a simplex mode to local memory. The data is written to system memory via a voted write. Hence the overhead is two system bus transfers.

- The error task, fault isolation, and reconfiguration functions in SIFT are similar to the FTMP system configuration controller.
- [Palumbo and Butler 85] described the overhead for three versions of the operating system, each an improvement over the previous. Although improvements in performance were obtained, further reductions in overhead, especially with interactive consistency, does not appear likely without hardware support.

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

- Further characterize 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 versus workload and task distribution. Will the system still meet its deadlines under increased load ?
- Further characterize the software overhead for both the faulty and fault-free behavior. 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 83c] showed the fault-handling 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 for FTMP into the application level of the performance evaluation matrix, along with exploring faulty behavior of the system.

Although application of the validation methodology on SIFT has thus far proven successful, it is by no means complete. The following discuss a few thoughts in these areas:

- Implement a synthetic workload to provide a tool for measuring the performance and interaction of the SIFT system.
- Determine the action if all processors executing a task exceed the allotted time.
- Characterize the interrupts on SIFT. Are the interrupts implemented on SIFT, and how will interrupts affect the real time and fault tolerant performance of the system ?
- Explore communication bandwidth versus the number of tasks. High performance implies minimizing overhead, hence a single task structure. Maximizing reliability requires subdividing a task to increase the frequency of voting.
- Investigate communication versus I/O, as both intertask communication and I/O contend for the same resources.
- Study broadcast bandwidth on SIFT; load the broadcast system to capacity.

- Characterize the effects of malicious liars, especially on clock skews.
- Explore the minimal time between faults such that reconfiguration is successful.

These experiments for SIFT move the validation and performance measurements into the executive level of the 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 FTMP and SIFT. 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 characterize most hardware primitives on FTMP [Clune 84, Feather et al. 85]. This report presents a continuation of the baseline experiments on FTMP and the start of validation procedure on SIFT. In particular this report presented:

- Clock read delay for SIFT. The global clock proved to be a reliable measuring device with a resolution of 28.3 microseconds, five times finer than FTMP's clock.
- High level language instruction execution times for both SIFT and FTMP. The execution times measured, for both FTMP and SIFT, were consistent and predictable with SIFT having a greater throughput.
- System memory read and writes times and the variance of the times caused by bus contention on FTMP. 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. SIFT is a fully distributed system with no global or system memory.
- Dispatcher execution times and overhead. In addition to failing to meet its real time constraints, the FTMP real time dispatcher consumed approximately 60% of the system throughput; whereas the SIFT dispatcher consumed 16% of a single subframe.
- Fault-handling software overhead. The software overhead involved in the fault-handling tasks consumed 5% of system throughput for FTMP. The overhead required for SIFT's fault tolerance required 66% of the system throughput.

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

- 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 on the fault tolerant design of FTMP.
- With SIFT a large overhead is involved with the fault-handling tasks, especially voting and interactive consistency. For 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 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 validation methodology is not machine specific. Although specific experiments were

designed for each machine, the methodology is general enough to apply to both systems without change.

- The methodology showed system validation can occur without using life testing approaches.
- By applying a building block approach in a systematic manner, the FTMP and SIFT systems were 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.

Appendix A. Instruction Execution Times

This appendix contains the tabulated results of the execution times of all the instructions measured. The predicted execution times are from [Rockwell Collins 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 A-1: FTMP Instruction Execution Times: 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 A-2: FTMP Instruction Execution Times: 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 A-3: FTMP Instruction Execution Times: 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 A-4: FTMP Instruction Execution Times: 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 A-5: FTMP Instruction Execution Times: Miscellaneous Operators

Raw Data: Read Time Clock Delay (Microseconds Per 100 Clock Reads, Including Null Loop Overhead)			
Processor	Actual Readings (Hex)		Microseconds
	Starting Time	Ending Time	(Decimal)
P1	53F6	5F1D	2855
P2	53F3	5F1B	2856
P3	53F0	5F1A	2858
P1	B4BC	BFE4	2856
P2	B4B7	BFDF	2856
P3	B4B9	BFE5	2860
P1	C519	D041	2856
P2	C51A	D042	2856
P3	C517	D040	2857
P1	F694	01BC	2856
P2	F694	01BB	2855
P3	F691	01BC	2859

Table A-6: Raw Data: SIFT Clock Read Experiment

Instruction Execution Times: Integer and Boolean (Range for 95% Confidence Interval)				
Pascal Instruction	Description	microsecs/100 inst. w/overhead	microsecs per instruction	
			w/overhead	w/o overhead
A := 1	Integer Assign	1456.28 ±.0072	14.56	3.70
A := B	Integer Variable Assign	1525.00 ±.0047	15.25	4.39
A := B + C	Integer Addition	1731.19 ±.0056	17.31	6.45
A := B * C	Integer Multiply	2343.46 ±.0089	23.43	12.57
A := B div C	Integer Division	3169.47 ±.0089	31.69	20.83
A := -B	Integer Negate	2031.08 ±.0036	20.31	9.48
A := B = C	Integer Compare	1937.37 ±.0083	19.37	8.51
A := B >= C	Integer Compare	2056.07 ±.0033	20.56	9.70
A := B < C	Integer Compare	2031.08 ±.0036	20.31	9.45
A := True	Boolean Assign	1456.26 ±.0069	14.56	3.70
A := B	Boolean Variable Assign	1526.26 ±.0036	15.26	4.39
A := B or C	Boolean Or	1774.96 ±.0035	17.75	6.89
A := B and C	Boolean And	1774.94 ±.0037	17.75	6.89
A := NOT B	Boolean Negate	1712.43 ±.0088	17.12	6.26

Table A-7: SIFT Instruction Execution Times: Integer and Boolean Data Types

Instruction Execution Times: Miscellanous (Range for 95% Confidence Interval)				
Pascal Instruction	Description	microsecs/100 instr. w/overhead	avg. microsecs per instruction w/overhead	w/o overhead
NULL	Null Loop	1086.38 ±.0085	-	10.86
Procall()	Procedure Call	1731.19 ±.0054	17.31	6.45
Procall(A)	Procedure Call	1785.92 ±.0064	17.86	7.00
Procall(A,B)	Procedure Call	2674.57 ±.0088	26.75	15.88
Procall(A,B,C)	Procedure Call	3113.23 ±.0065	31.13	20.27
Procall(A,B,C,D)	Procedure Call	3525.58 ±.0087	35.26	24.39
If GO then A:=1	Conditional, True	1781.18 ±.0052	17.81	6.95
If GO then A:=1	Conditional, False	1456.29 ±.0074	14.56	3.70
If GO then A:=1 Else B:=1	Conditional, True	1918.62 ±.0084	19.19	8.32
If GO then A:=1 Else B:=1	Conditional, False	1799.90 ±.0042	18.00	7.14

Table A-8: SIFT Instruction Execution Times: Miscellaneous Instructions

Instruction Execution Times: Combinations (Range for 95% Confidence Interval)				
Pascal Instruction	Description	microsecs/100 instr. w/overhead	microseconds per instruction w/overhead	w/o overhead
A := 1	1 Iteration	1456.28 ±.0072	14.56	3.70
	2 Iterations	1662.48 ±.0089	16.62	5.76
	3 Iterations	1868.63 ±.0083	18.69	7.82
	5 Iterations	2280.98 ±.0034	22.81	11.95
	8 Iterations	2899.51 ±.0089	29.00	18.13
	10 Iterations	3338.17 ±.0065	33.38	22.52
	12 Iterations	3750.52 ±.0089	37.51	26.64
	15 Iterations	4369.00 ±.0358	43.69	32.83
	20 Iterations	5219.97 ±.0216	52.20	41.34

A := 1
A := B + C
	Assign & Add Combination	2074.83 ±.0049	20.75	9.88

A := 1
A := B * C
	Assign & Mult Combination	2687.12 ±.0043	26.87	16.01

A := 1
A := B div C
	Assign & Div Combination	3513.13 ±.0077	35.13	24.27

A := 1
A := B + C
A := B * C
	Assign, Add, Mult Combination	3331.93 ±.0067	33.32	22.46

A := 1
A := B + C
A := B / C
	Assign, Add, Div Combination	4131.63 ±.0020	41.32	30.45

A := 1
A := B * C
A := B / C
	Assign, Mult, Div Combination	4564.03 ±.0170	45.64	34.78

Table A-9: SIFT Instruction Execution Times: Instruction Combinations

Comparison of Instruction Combinations Not Tested on FTMP (in microseconds per instruction without overhead)				
Pascal Instruction	Description	Time If Done Separately	Time For Combination	Percent Difference Separate vs. Combo

A := 1
A := B + C
	Assign & Add Combination	10.15	9.88	2.7%

A := 1
A := B * C
	Assign & Mult Combination	16.27	16.01	1.6%

A := 1
A := B div C
	Assign & Div Combination	24.53	24.27	1.1%

A := 1
A := B + C
A := B * C
	Assign, Add, Mult Combination	22.72	22.46	1.2%

A := 1
A := B + C
A := B / C
	Assign, Add, Div Combination	30.98	30.45	1.7%

A := 1
A := B * C
A := B / C
	Assign, Mult, Div Combination	37.1	34.78	6.7%

Table A-10: SIFT: Comparison Instruction Combinations Not Done on FTMP

Appendix B. Block Transfer Execution Times

This appendix contains tabulated results of the block transfer experiment:

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 B-1: FTMP 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 B-2: FTMP Block Transfer Times, Write from Local to System Memory

References

[Clune 84] Ed Clune.
Analysis of the Fault Free Behavior of the FTMP Multiprocessor System.
Technical Report CMU-CS-84-130, Carnegie Mellon University, 1984.

[Czeck et al. 85] Edward W. Czeck, Daniel P. Siewiorek, Zary Z. Segall.
Fault Free Performance Validation of a Fault-Tolerant Multiprocessor: Baseline and Synthetic Workload Measurements.
Technical Report CMU-CS-85-177, Carnegie Mellon University, 1985.

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

[Draper 83b] *Development and Evaluation of a Fault-Tolerant Multiprocessor Computer, Vol. II, FTMP Software*
Charles Stark Draper Laboratories, 1983.
NASA CR 166072.

[Draper 83c] *Development and Evaluation of a Fault-Tolerant Multiprocessor Computer, Vol. III, FTMP Test and Evaluation*
Charles Stark Draper Laboratories, 1983.
NASA CR 166073.

[Draper 84] *Development and Evaluation of a Fault-Tolerant Multiprocessor Computer, Vol. IV, FTMP Executive Summary*
Charles Stark Draper Laboratories, 1984.
NASA CR 172286.

[Feather et al. 85]
Frank Feather, Daniel Siewiorek, and Zary Segall.
Validation of a Fault-Tolerant Multiprocessor: Baseline Experiments and Workload Implementation.
Technical Report CMU-CS-85-145, Carnegie Mellon University, July, 1985.

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

[Green et al. 84] David F. Green, Jr., Daniel L. Palumbo, and Daniel W.Baltrus.
Software Implemented Fault-Tolerant (SIFT) User's Guide
NASA-Langley Research Center, 1984.
NASA TM 86289.

[Holt et al. 84] H.M. Holt, A.O. Lupton, and D.G. Holden.
Flight Critical System Design Guidelines and Validation Methods.
AIAA/AHS/ASEE Aircraft Design Systems and Operating Meeting (AIAA-84-2461), 1984.

[Hopkins et al. 78]
A.L. Hopkins, T.B. Smith, and J.H. Lala.
FTMP - A Highly Reliable Multiprocessor.
In *Proceeding of the IEEE*, pages 1221-1237. October, 1978.

[Kong 82] Thomas H. Kong.
Measuring Time for Performance Evaluation of Multiprocessors Systems.
Master's thesis, Carnegie-Mellon University, 1982.

[Lala 85] Parag K. Lala
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.

[Palumbo 85] Daniel L. Palumbo.
The SIFT Hardware/Software Systems
NASA-Langley Research Center, 1985.
NASA TM 87574.

[Palumbo and Butler 85]
Daniel L. Palumbo and Ricky W. Butler.
Measurement of SIFT Operating System Overhead
NASA-Langley Research Center, 1985.
NASA TM 86322.

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

[Shin and Krishna 84]
Kang G. Shin, C.M. Krishna.
Characterization of Real-Time Computers
NASA-Langley Research Center, 1984.
NASA CR-3807

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

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

[SRI 83] *Investigation, Development, and Evaluation of Performance Proving for Fault-Tolerant Computers*
SRI International, 1983.
NASA CR-166008.

[SRI 84] *Development and Analysis of the Software Implemented Fault-Tolerance (SIFT) Computer*
SRI International, 1984.
NASA CR 172146.

[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 et al. 78]
J.H. Wensley, L. Lamport, J. Goldberg. M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shostak, and C.B. Weinstock.
SIFT: A Computer for Aircraft Control.
In *Proceeding of the IEEE*, pages 1240-1255. October, 1978.

Standard Bibliographic Page

1. Report No. NASA CR-178236	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Fault-Free Performance Validation of Fault-Tolerant Multiprocessors		5. Report Date January 1987
		6. Performing Organization Code
7. Author(s) Edward W. Czeck, Frank E. Feather, Ann Marie Grizzaffi, Zary Z. Seagall, and Daniel P. Siewiorek		8. Performing Organization Report No.
9. Performing Organization Name and Address Carnegie-Mellon University Department of Electrical and Computer Engineering Pittsburgh, PA 15213		10. Work Unit No.
		11. Contract or Grant No. NAG-1-190
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546		13. Type of Report and Period Covered Contractor Report
		14. Sponsoring Agency Code 505-66-21-01
15. Supplementary Notes Langley Technical Monitor: George B. Finelli		

16. Abstract

A validation methodology for testing the performance of fault-tolerant computer systems was developed and applied to the Fault-Tolerant Multiprocessor (FTMP) at NASA-Langley's AIRLAB facility. This methodology was claimed to be general enough to apply to any ultrareliable computer system.

The goal of this research was to extend the validation methodology and to demonstrate the robustness of the validation methodology by its more extensive application to NASA's Fault-Tolerant Multiprocessor System (FTMP) and to the Software Implemented Fault-Tolerance (SIFT) computer System. Furthermore, the performance of these two multiprocessors was compared by conducting similar experiments.

An analysis of the results shows high level language instruction execution times for both SIFT and FTMP were consistent and predictable, with SIFT having greater throughput. At the operating system level, FTMP consumes 60% of the throughput for its real-time dispatcher and 5% on fault-handling tasks. In contrast, SIFT consumes 16% of its throughput for the dispatcher, but consumes 66% in fault-handling software overhead.

17. Key Words (Suggested by Authors(s)) Fault-Tolerant, Multiprocessors, Validation, Performance Measurement, SIFT, FTMP		18. Distribution Statement Unclassified - Unlimited Subject Category - 62	
19. Security Classif.(of this report) Unclassified	20. Security Classif.(of this page) Unclassified	21. No. of Pages 67	22. Price A04

For sale by the National Technical Information Service, Springfield, Virginia 22161

NASA Langley Form 63 (June 1985)