Validation of a Fault-Tolerant Multiprocessor: Baseline Experiments and Workload Implementation

Frank Feather, Daniel Siewiorek, Zary Segall

22 July 1985

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

CMU-CS-85-145

Validation of a Fault-Tolerant Multiprocessor: Baseline Experiments and Workload Implementation

Frank Feather, Daniel Siewiorek, Zary Segall

22 July 1985

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

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

Table of Contents

List of Figures

Abstract

In the future, aircraft must employ highly reliable multiprocessors in order to achieve flight safety. Such computers must be experimentally validated before they are deployed. This project outlines a methodology for validating reliable multiprocessors. The methodology begins with baseline experiments, which test single phenomenon. As experiments progress, tools for performance testing are developed. This methodology is used, in part, on the Fault-Tolerant Multiprocessor (FTMP) at NASA-Langley's AIRLAB facility. Experiments were designed to evaluate the *fault-free* performance of the system.

This report presents the results of interrupt baseline experiments performed on FTMP. Interrupt causing exception conditions were tested, and several were found to have unimplemented interrupt handling software while one had an unimplemented interrupt vector. A synthetic workload model for realtime multiprocessors is then developed as an application level performance analysis tool. Details of the workload implementation and calibration are presented.

Both the experimental methodology and the synthetic workload model are general enough to be applicable to reliable multiprocessors beside FTMP.

1. Introduction

In the 1990's aircraft will employ computers that must run correctly and continuously for the aircraft to fly. NASA, in its Aircraft Energy Efficiency (ACEE) program requires that the probability of failure in these computers be less than 10^{-10} per hour. Meeting such requirements can not be achieved with standard realtime computers; instead fault-tolerant computers have been developed to meet these requirements. Two such systems are SIFT (Software Implemented Fault-Tolerance) [Wensley 78] conceived by SRI International and fabricated by Bendix Corporation; and FTMP (Fault-Tolerant Multiprocessor) [Hopkins 78], conceived by MIT's Charles Stark Draper Laboratory, Inc. and fabricated by Collins.

These complex systems, which must meet stringent performance requirements, have to be validated (i.e. proven functionally correct). However, since a probability of failure of 10^{-10} per hour translates to one failure per million years of operation, a validation method must be developed to discover flaws in the design and implementation before such a system is placed into service. Proving a system correct can take place at many stages from mathematical models and theorem proving, also called *verification*, to experimental testing, called *validation*. Mathematical models of the system are based on simplifying assumptions and can be used in conjunction with, but not as a substitute for, actual experimentation. Indeed, many of the errors in a system surface during the experimentation and use of the system. Bell Telephone [Toy 78] divided the causes of system outages for their fault tolerant electronic switching systems into several categories. The percentages given for each category represents fraction of total down time measured in the field attributed to each cause:

- Hardware Reliability: Actual component failures -- 20%
- Software Deficiencies: Software design errors -- 15%
- Recovery Deficiencies: Inability to detect, isolate, and correctly recover from faults -- 35%
- Procedural Errors: Human error on the part of maintenance personnel or office administrators -- 30%

Fault-Tolerant techniques directly impact the first category. The later three categories are all forms of design errors. These errors can be reduced by effective system design and validation.

The goal of this research is to develop a methodology for the validation of the fault free performance of fault-tolerant avionic multiprocessors. Initially this methodology will be applied to FTMP, although the approach should be general enough to migrate to other fault-tolerant systems like SIFT.

2. Background

2.1. Guidelines to Experiments

Over the last decade, Carnegie-Mellon University has devoted over 100 man-years to the design, construction, and validation of multiprocessor systems. Some of the guidelines developed over the last decade include:

- The experimental validation methodology is successively refined as experiments uncover new information and/or the methodology is applied to new multiprocessor systems.
- Design experiments to validated behavior that is documented as well as uncover behavior that is not documented.
- Perform experiments in a systematic manner. Since the search is for the unexpected there is no shortcut to thorough testing.
- Experiments should be repeatable.
- The feasibility of performing various experiments is tempered by what is available in the experimental environment. More sophisticated experiments may have to be postponed until the experimental environment is provided with more tools.
- A building block approach should be used wherein one variable is changed at a time so that causes of unexpected behavior are easy to isolate.
- Testing should take advantage of the abstract levels used in the design of the system.

Using these guidelines, we will develop a generalized methodology for testing multiprocessor systems.

2.2. Proposed Methodology

Showing that a computing system, as designed, will meet its dependability goals is called validation [NASA 79a]. In 1979, NASA held several workshops to determine system validation procedures. One in particular [NASA 79b], produced a detailed list of validation categories to evaluate the system in an orderly manner. A building block approach was chosen so that confidence in the system would be built up in an incremental manner starting with the understanding and measurement of primitive hardware and operating system activities. After primitive activities are characterized, more complex experiments are devised to define interactions between primitive activities. This orderly progression insures uniform coverage and makes it easier to locate the cause of an unexpected phenomenon. Steps in the proposed methodology included:

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 Validation and Performance Baseline

7. Simulation of Inaccessible Physical Failures
8. Single Processor Fault Insertion
9. Multiprocessor Fault Insertion
10. Single Processor Executive Failure Response Characterization
11. Multiprocessor System Executive Fault Handling Capabilities
12. Application Program Validation on Multiprocessor
13. Multiple Application Program Validation on Multiprocessor

The first six tasks validate the fault free functionality of the system while the next seven validate fault handling capabilities. Step 1, initial checkout and diagnostics, is usually done before system delivery, while Step 2, manual validation, is ongoing throughout the testing process. Part of this project involved updating and clarifying information in FTMP's manuals [Draper 83a, Draper 83b] with a user's guide [Feather 84]. Of the other fault free validation steps, Step 4 is considered hardware validation, Steps 3 and 5 are operating system level validation, and Step 6 is application level validation. This project deals with fault free performance (Steps 2 through 5), and develops an application level tool called the *synthetic workload* to address Step 6.

Ideally, hardware and operating system validation should take place in the development stage of the respective levels. For example, as the operating system is written, a set of validation tests is produced.

Each step of the methodology, like the whole methodology, follows a building block approach. First, *baseline* experiments are conducted. Baseline experiments measure a single phenomenon while all other interactions are held constant. These experiments are designed to validate the basic assumptions used in the mathematical models as well as validate the assumptions made by the application programmers. Once individual phenomenon have been characterized, more advanced experiments can be conducted which explore the interaction between basic phenomena.

As stated in the experiment guidelines, the validation procedure is tempered by the available experimental environment. This implies that at any one step, more sophisticated experiments may have to be postponed while the experimenter moves on to the next step to conduct baseline experiments until the advent of more sophisticated experimental tools. Experiments can proceed in parallel if tools are available at a higher yet disjoint step. For example, at AIRLAB, fault insertion experiments occur in parallel with fault-free validation and performance experiments.

2.3. Definition of Performance

Validation experiments test system behavior and establish whether the system works correctly. That is, validation experiments test functional correctness. In addition to establishing behavior, performance can also be measured. Performance refers to *how well* a system, assumed to be functionally correct, works. Validation and performance are not always dichotomous; in some systems, if performance criteria are not met the system is considered to be incorrect. Therefore, validation experiments are usually accompanied

by performance analysis. For example, testing basic instruction times, besides testing functional correctness of hardware instructions, also can be used to estimate total system throughput in terms of operations per second.

Performance measurements can be conducted at many levels, starting with the instruction set, working up to the operating system and then the application level. Three parameters which can be measured at each level are Throughput, Utilization, and Delay. Initially, the baseline experiments took measurements from the instruction set and operating system level. However, these experiments quickly progressed to the application level with construction of the synthetic workload. There are several advantages to validation at the application level:

1. This is the level that real programs (i.e. natural workloads) run. Any meaningful statements about computer performance to the application programmer must be based on measurements made at this level.

2. Experiments are much easier to design at the application level. The person validating the system at this level does not need hardware and/or operating system expertise.

	Behavior	Throughput	Utilization	Delay
Application	Display, Flight Control	Subtask Execution Times	Idle Time	Write, Read Delay & Variation
Executive Software	Scheduler, Message System	OS Primitives Times	OS Primitives Freq. of Use	Primitive Variation, Contention
Instruction Set	Instruction, Exceptions	Instr. & Resource Times	Resource Freq. of Use	Resource Variation, Contention

Figure 2-1: Performance Evaluation Matrix

Figure 2-1 illustrates the system levels and the types of performance experiments that can take place at each level. In more detail, the performance measurements are:

- Throughput:
    - Instruction Set: Measure the time to access limited resources (e.g. memory, clock) and execute instructions
    - Operating System: Measure the execution times of the operating system primitives and tasks
    - Application Software: Measure the execution times of the different subsections of each application task

  Utilization:
    - Instruction Set: Frequency and percentage of hardware resource used

- Operating System: Frequency of OS primitives use
- Application Software: Measure idle time between tasks

Delay (and Variation):

- Instruction Set: Variation in the access time of resources; amount of contention for resources
- Operating System: Variation in execution of primitives due to resource contention
- Application Software: Delay (and variation) between a data write and a data read of common data

In general, baseline experiments are conducted at the instruction set and operating system levels while more complex measurements occur at the application level.

Initially, this project deals with instruction set/executive level baseline experiments (interrupts). However, realizing that the most meaningful performance statements come from the application level, an application level performance tool called the synthetic workload was developed. Baseline experiments and workload implementation were done on the Fault-Tolerant Multiprocessor (FTMP). The next section discusses that computer.

2.4. The FTMP and Experimentation Environment

The Fault-Tolerant Multiprocessor (FTMP) has been discussed in several papers and manuals [Draper 83b, Hopkins 78]. This section is a software overview of FTMP from the application programmer's perspective. The reader is referred to the references mentioned above for more details.

Figure 2-2 illustrates the FTMP system. Each processor is this figure actually consists of three processors in a fault-tolerant configuration executing in lockstep. This trio of processors is sometimes referred to as a *processor triad* or a *virtual processor* because the application programmer sees it as a single processor. Likewise, memory is in a triad configuration. The FTMP can consist of one, two or three processor triads. Each triad has a local memory which is divided into PROM and RAM. The PROM contains frequently used executive code and is identical in all processors. Each processor's RAM holds local variables and stack, plus application software paged in from global memory. A bus connects the triads to global memory, I/O devices, a real-time clock and several latches needed for fault handling. The triads execute independently of each other when accessing global memory. If a program running on a processor triad uses a global variable, the program must first move the variable from global to local memory with a bus service routine. Similarly, the variable is written back to global memory with another bus service routine.

Work on FTMP is performed by tasks. A task is a single thread of execution that runs by itself. Each task has a time limit associated with it. If a task does not complete by its allotted time it is aborted and

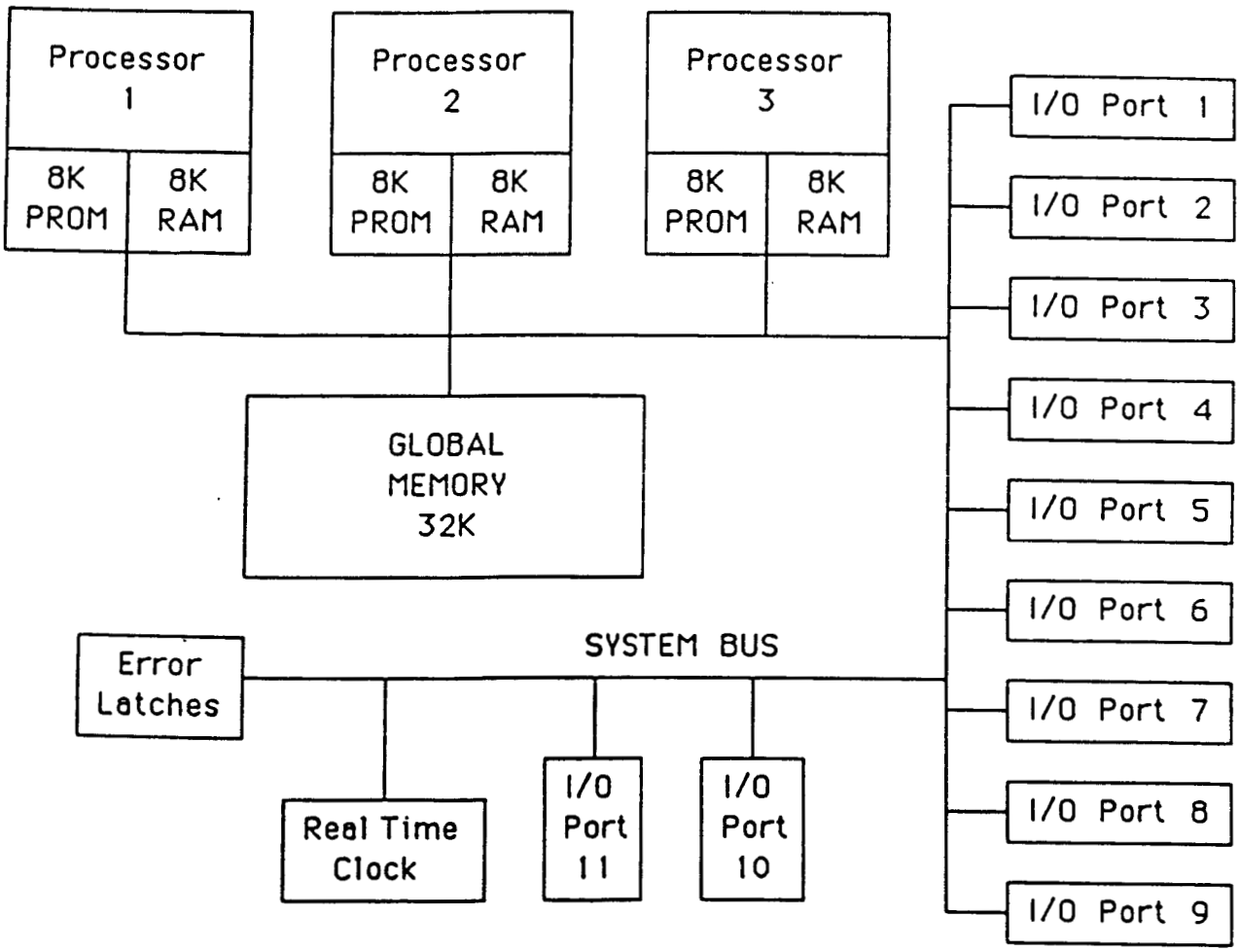

Figure 2-2: Software Appearance of FTMP (virtual machine)

another task is started. A task can execute on any processor triad[1].

In a realtime system a task is run at regular interval which defines the task's *iteration rate*. Not all tasks need to run at the same iteration rate. For example, the task that updates the display terminal does not have to be executed nearly as often as the task that monitors and adjusts the plane's airspeed. Tasks are grouped by common iteration rate, called rate groups, and are run within *frames*. A frame defines the execution interval length and is essentially one over the iteration rate. In the time allotted by the frame, the working triads must execute all the tasks defined for the frame's iteration rate. Task control blocks, which contain all the information necessary to run a task, are in a linked list resident in global memory. Individual triads access this global list to select a task to run. When FTMP is in a multiple triad configuration, some tasks will execute in parallel. When there are no more tasks left in a particular iteration rate group to execute, a triad will either become idle or start executing tasks from a lower iteration rate group. Figure 2-3 is an example of a task control block structure arranged by rate groups (defined below). The control blocks in this figure are those of the synthetic workload (Section 4).

The FTMP has three iteration rates which define three different frame sizes. There are separate task control block lists -- one for each rate group. The frame sizes are:

- R4, the basic frame size
- R3, equivalent to 2 R4 frames
- R1, equivalent to 4 R3 frames; also called the *major* frame

Figure 2-4 illustrates the different frames and their execution frequencies.

FTMP handles the multiple rate groups as follows. At the beginning of an R4 frame, one of the triads, called the *responsible triad*, starts the R4 frame for that triad and signals another triad to start its frame. This second triad in turn signals the third triad, if it exists, to start its R4 frame. Each R4 frame does not necessarily have the same responsible triad. Every second R4 frame signals the start of an R3 frame and every eight R4 frames starts an R1 frame. Once a triad runs out of R4 tasks to execute, the triad will begin taking tasks from the R3 task list to execute. Likewise, when a triad runs out of R3 tasks it takes tasks from the R1 task list. Execution of a lower task frame group can be suspended in a triad by the start of a higher numbered frame group. Suspended tasks are continued once the the triad runs out of tasks from the higher iteration rate. For example, the beginning of an R4 frame suspends execution of R3 and R1 tasks until all tasks in the R4 frame finish. The processor triad that finishes the last R4 task in the R4 frame becomes the responsible triad that starts the next R4 frame.

Several computer systems are involved in creating and running experiments on FTMP as illustrated in

[1]The only exception is a rate 1 task called "SCC", the system configuration control task; this task is systematically run on different processor triads so it can execute self-tests on each triad. There is a bit in SCC's Task Control Block, set by SCC, that specifies on which triad the dispatcher should run SCC.

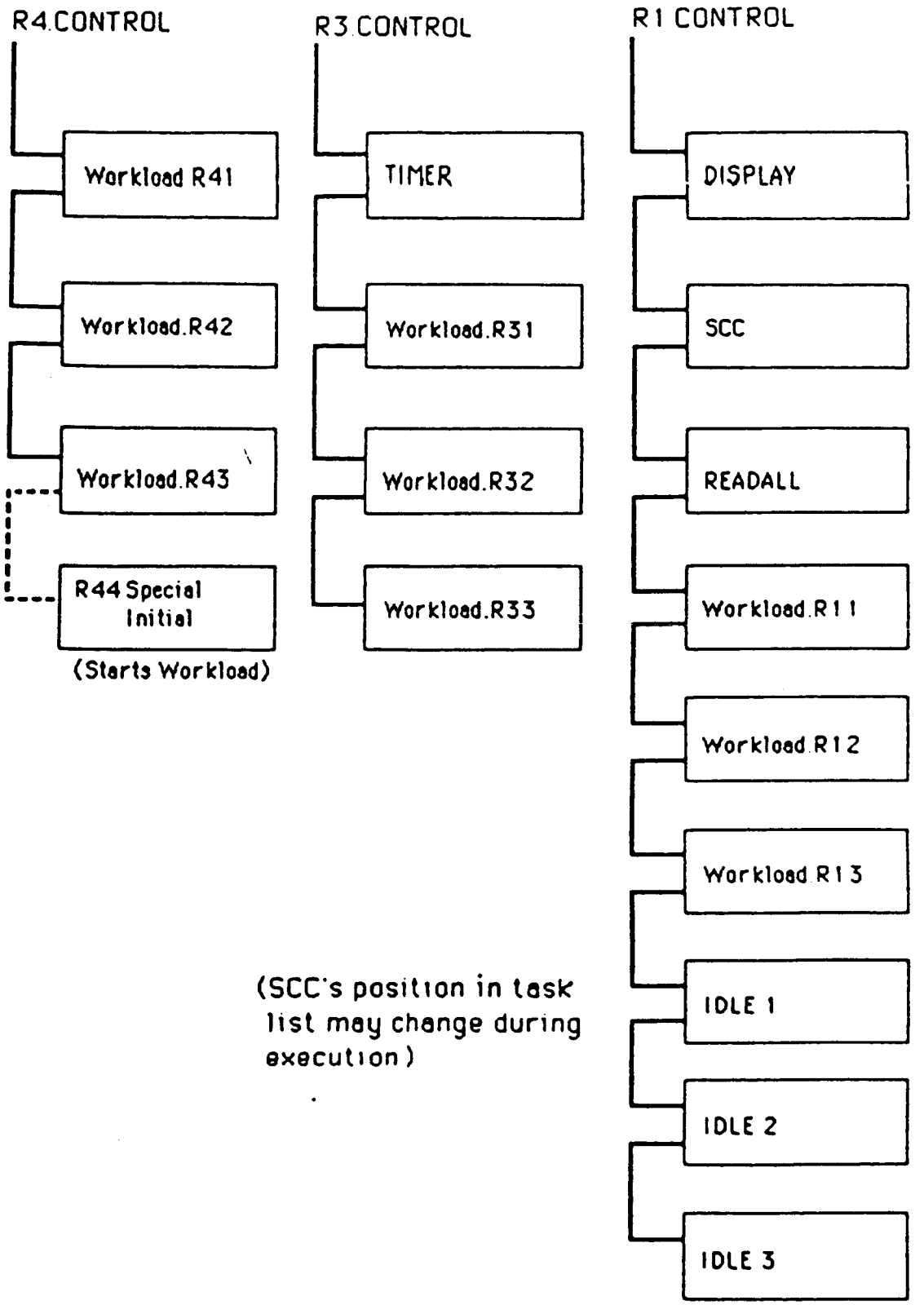

Figure 2-3: Task Control Block Structure

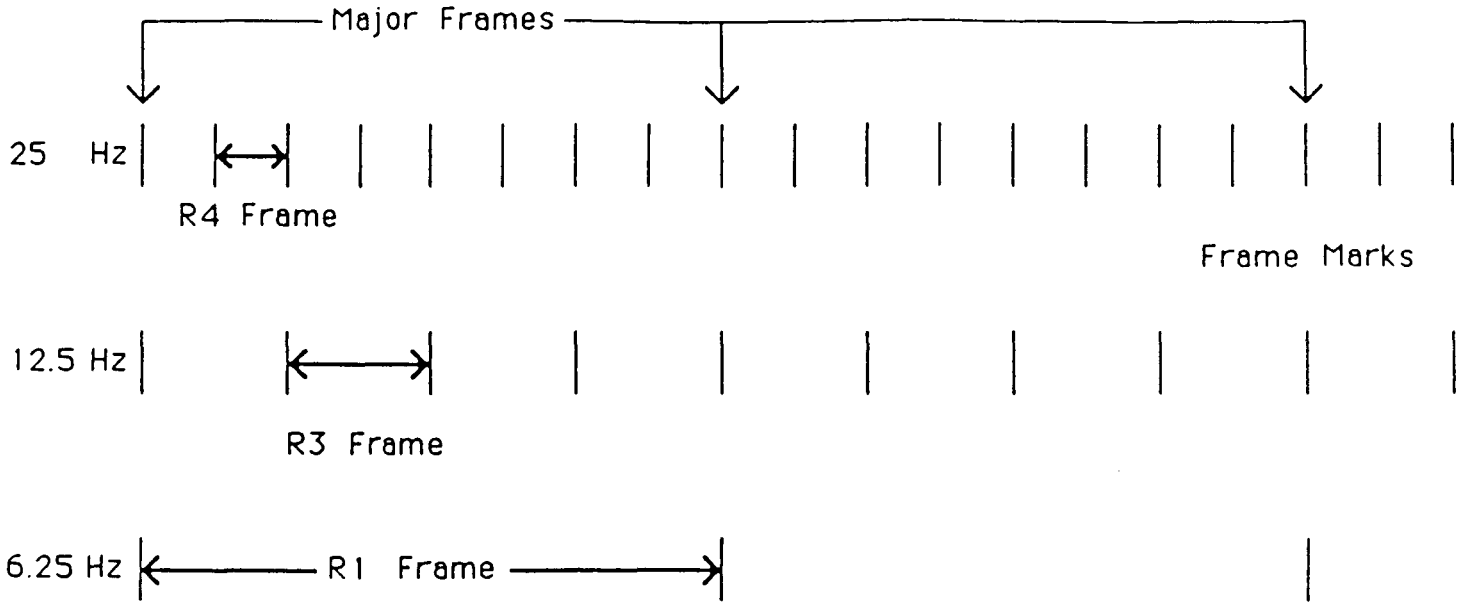

Figure 2-4: Frame Structure

Figure 2-5. The steps to creating an experiment and the systems involve include:

- Create and compile a program task written in a language called Automated Engineering Design (AED) system which runs on an IBM 4341.
- The user must map out where the code goes in memory along with the location of stack, local, and system variables.
- The user then modifies OS task tables to include the task in FTMP's task structure, reassembling task tables when finished.
- The experimental task is linked with the rest of the operating system code to create an absolute load module.
- The load module is downline loaded from the IBM 4341 to a VAX-11/750.
- The load module is downline loaded from the VAX-11/750 to FTMP.
- The FTMP test adapter (CTA) is used to debug the experimental program.
- Once the experimental program is correct, the test adapter is used to dump a memory image into a file for later analysis.

Figure 2-6 illustrates the process of creating a program. The experimental loop may take up to two hours from the time of compiling a program on the IBM 4341 until it is executed on FTMP. The experimenter must have knowledge of several systems including the IBM 4341, the VAX-11/750, and FTMP. The experimenter also must be intimately familiar with FTMP's hardware, operating system, and task structure.

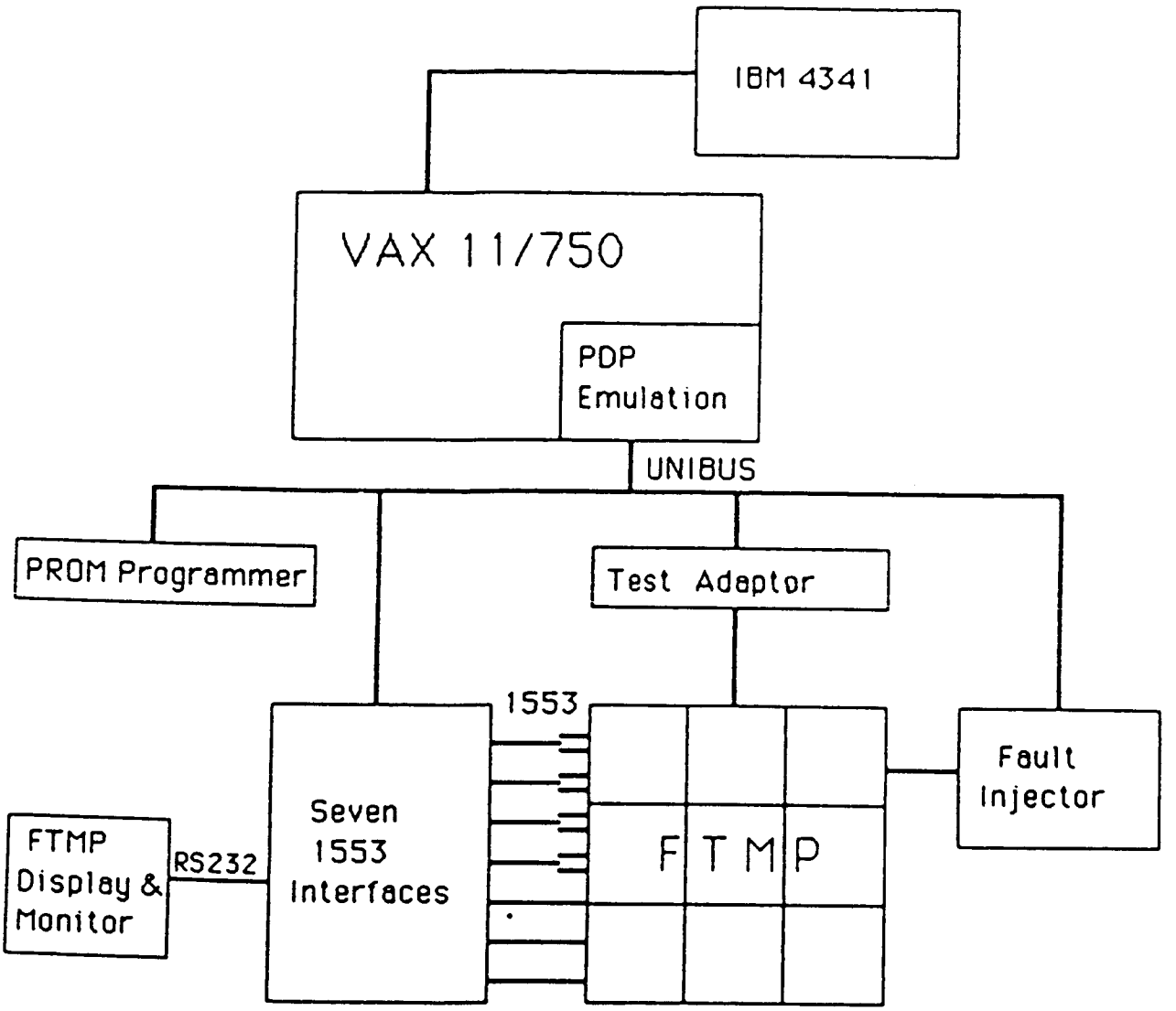

Figure.2-5: FTMP Support Environment

In order to shorten this experimental loop and improve experimental efficiency, a synthetic workload model for real time avionic systems was proposed [Clune 84]. With an easy to envision model, an experimenter can be working with the workload after merely a few hours of reading over the model, getting an overview of FTMP and learning VAX/VMS commands; the IBM 4341 is eliminated from the experimental loop.

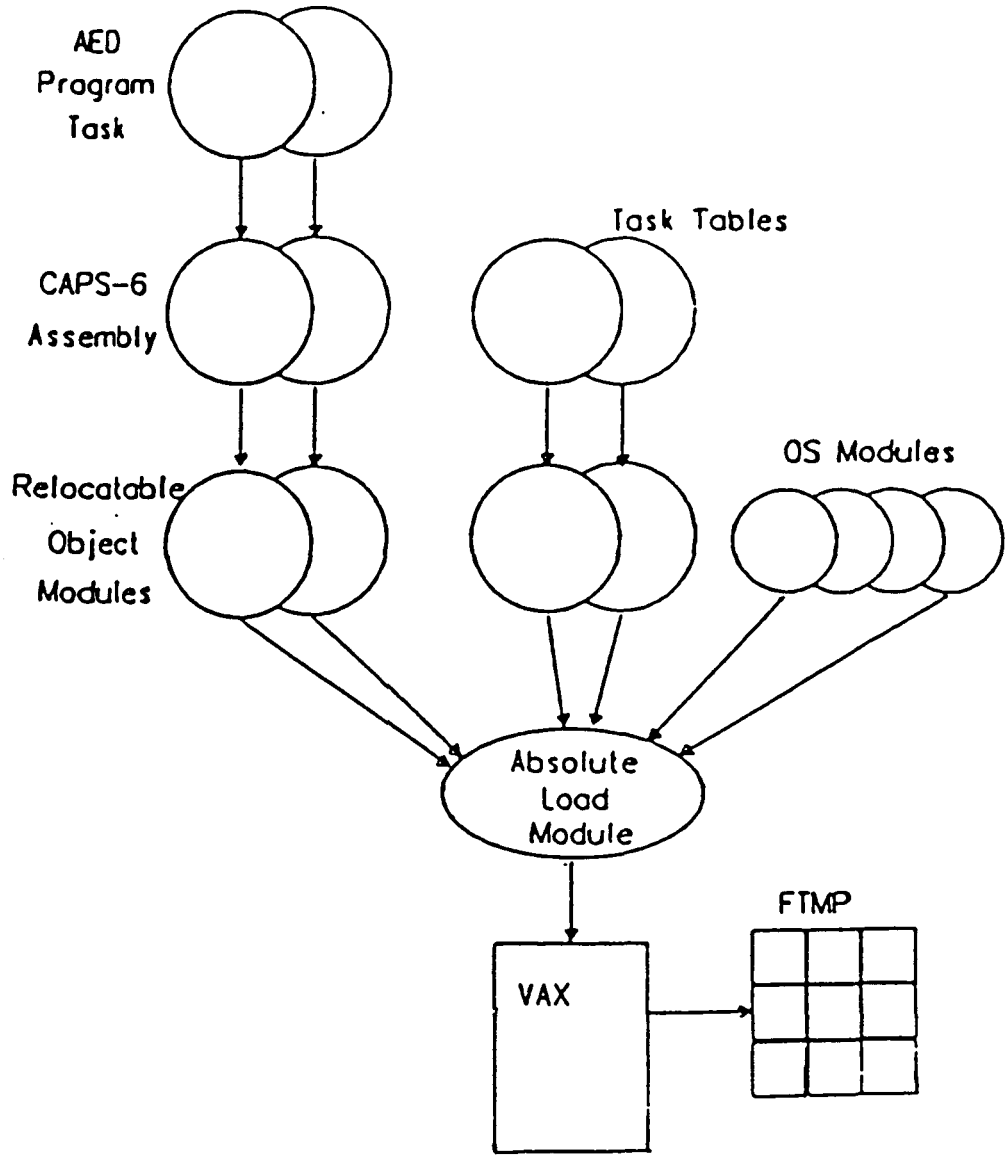

Figure 2-6: Steps to Creating a Program

2.5. Previews of Experiments

To date, baseline experiments up to the application level have been performed. Various experiments, classified by the level of abstraction presented in Figure 2-1, are shown below. Experiments marked by an asterisk (*) have already been performed [Clune 84].

1. Instruction Set Level:

   - Verify the clock as an accurate fundamental measuring device. With the clock calibrated, future performance experiments can be performed with confidence. (*)
   - Timings of Assembly and High-level language instructions. (*)

- Observe and document the existence and the direct effects of interrupts.

2. Executive Software Level:

   - Executive primitive and overhead times (*)
   - Interrupt procedure times
   - Memory Access time
   - Bus access and contention delays
   - Fault-tolerant overheads

3. System and Application Level:

   - Frame utilization characteristics (*)
   - Length of the frame of all task iteration rate groups
   - Fault-tolerant overhead to the application programmer

4. Development of an application level tool for measuring performance.

This report covers two experiments on FTMP. First, an experiment was run to test the existence and document effects of interrupts on FTMP. The second part of this report discusses the development and implementation of the application level tool called the synthetic workload. An experiment to calibrate the synthetic workload is also discussed. Once installed, the synthetic workload can be used to run application level experiments as well as certain executive level baseline experiments.

3. Interrupts

Interrupts can be viewed as a signal of unusual events in a processor. These signals can be of simple events like arithmetic overflow or of more complex events like a device is ready for input. Interrupts can be used for communication between a user process and the supervisor, in which case they are called *traps*. A user process invokes a trap to request service (I/O, resource request, etc.) that the user process could not fulfill directly. Interrupts are also a mechanism for enforcing virtual memory and protection schemes. Interrupts notify the processor that a memory reference was to a page not in memory (page fault) and the page needs to be brought in, or can halt a program that tries to access memory outside its memory space. Finally, interrupts are a mechanism for software reliability. Whereas, fault-tolerant systems, through redundancy, can catch hardware errors and mask or record them for later reconfiguration, interrupts are the mechanism for detecting and recovering from software faults. There are four categories of interrupts:

Intraprocessor — asynchronous events that happen within the processor during the execution of a machine instruction. Examples of these events include: zero divide, arithmetic overflow, memory access violation, privileged instruction execution, and page fault.

Intrasystem — interrupts caused by a peripheral such as a disk, timer or terminal. Examples of these interrupts include timer reached zero, input received, and output device ready.

Executive — caused by the current executing program. Executive interrupts are used to make requests of the executive (operating system) program. Examples of such requests are starting new tasks, allocating hardware resources, communication to other tasks, etc. These interrupts are sometimes referred to as traps, supervisor calls (SVC), or privileged mode calls.

Interprocessor — interrupts between two intelligent processors. This type of interrupt can be used to implement an interprocess communication (IPC) mechanism between processors.

This section describes mechanisms used in implementing interrupts, followed by a discussion of interrupts on FTMP CAPS-6 processor. Finally, results of experiments to test interrupts mechanism on FTMP are presented.

3.1. Mechanisms

Generally, interrupts are *vectored*, that is, the address of the interrupt handling routine is in a special memory location. When an interrupt occurs, control is transferred to a routine pointed to by this vector. Several devices can be associated with a single interrupt vector, in which case the processor must *poll* the devices to see which caused the interrupt.

When there are several interrupt vectors, a system will sometimes have interrupt *priority* nesting. Nesting allows higher priority interrupts (e.g. power failure) to interrupt the processing of low priority interrupt routine (e.g. overflow).

To provide operating system support for protection mechanisms, most computers have, at the very minimum, user and supervisor states. Which protection violations are reported are a function of machine state. Obviously, interrupts like privileged instruction violation should not occur in supervisor state, hence there is an architectural decision of which interrupts are ignored in supervisor state.

Finally, there is the issue of *disabling* and *masking* interrupts. Disabling an interrupt prevents a device from sending an interrupt. Thus the interrupt signal is actually turned off. In contrast, masking does not prevent the interrupt from occurring, but instead ignores the interrupt until the mask is changed. Using this definition, in a priority interrupt scheme, low priority interrupts are masked by a higher priority interrupt. Processors generally have a hardware mask field which tells which interrupts to ignore. In general, most interrupts (overflow, I/O, etc.) are supervisor maskable, but only intrasystem and interprocessor interrupts can be disabled.

Some system responses to interrupt include:

- Do nothing. The results are equivalent to masking the interrupt except that the interrupt is cleared since it was acknowledged. For example, some applications might wish to be notified of an overflow condition yet continue execution.
- Abort the current job (e.g. divide by 0, memory access violation, etc.).
- Restart the job or start a job with new software (e.g. N-version programming). This is a consideration in a software reliable system.
- Performs service (e.g. privilege mode call, page fault).
- React to an event (e.g. timer interrupt, i/o interrupt, IPC interrupt).

3.2. Interrupts, System Validation, and Performance

The steps to evaluating interrupts are similar to the steps taken when evaluating any part of the system. First, the existence of the interrupt is tested, thus validating the programmer's manual. Baseline experiments follow which test functional correctness of the interrupt mechanisms (i.e. do interrupt masking mechanisms work correctly, are supervisor/user effects of interrupts correct, etc.). Interrupt evaluation encompasses both the hardware and the operating system. Interrupts are invoked in hardware, but the interrupt handlers are in the operating system.

Interrupts do effect performance. An add instruction that overflows (thus invoking an interrupt) is slower than the equivalent instruction that does not overflow. Likewise, page faults impact performance. Therefore, the performance matrix of Figure 2-1 was used:

- Throughput -- How long does it take to process the interrupt? This delay is a function of the length of the interrupt handler, the system load, whether the handler is in memory (i.e. does it need to be paged in), etc.

- Utilization -- How often are interrupts invoked. Although utilization of processor exception interrupts (overflow, privileged mode violation, etc) is of less interest due to rarity, utilization of IPC and page fault interrupts are more frequent.

- Delay -- Variation of interrupt delay between processors. Also, does the effect of interrupts cross processor boundaries?

The following is an example of experimental steps for evaluating interrupts:

1. Test the existence of interrupts (manual validation).

2. Test interrupt masking mechanisms. Also test which interrupts occur in user versus supervisor mode.

3. Test how long it takes to process each intraprocessor interrupts (overflow, page fault, etc.). Compare this to interrupt-free execution.

4. What is the overhead of processing intrasystem interrupts (timer, terminal, etc.). How often to these interrupts occur?

5. For executive interrupts (traps), evaluate how long it takes to service the trap. Likewise, how long does a processor take to respond to an IPC interrupt?

6. What is the interrupt rate of page fault and IPC interrupts? For typical instruction execution, how often to page faults occur?

7. Perform the above tests in both uniprocessor and multiprocessor configurations.

3.3. Interrupts on FTMP

The processor elements in FTMP are Collins Avionics CAPS-6 processors modified for fault tolerance. The CAPS-6 processor has 18 interrupt vectors, stored in the first 18 words of PROM. Vectors 0-7 are unavailable in the FTMP implementation of the CAPS-6 processor. According to documentation [Draper 83a], interrupts can only occur in user mode; interrupts in supervisor mode are automaticly masked. Actual implementation reveals that interprocess communication (IPC), interval timer, and page fault interrupts can occur in supervisor mode. Otherwise, for example, the processor would not be able to page executive code. Interrupts 8-F_{16} are maskable. The CAPS-6 has a bit mapped interrupt mask which is stored in the Process Status Descriptor (PSD) of each task. This mask is loaded into the hardware interrupt mask when the task is started. There are no interrupt priority levels in the CAPS-6 processor.

Figure 3-1 summarizes FTMP's interrupts. This table also presents the results of experiments to test the effect and existence of these interrupts.

3.4. Experimental Results

Many of the interrupts do not have an interrupt handler. These are:

- Arithmetic Overflow
- Write Protection Violation
- Illegal Opcode
- Stack Overflow
- Non-local Search Fault
- Privileged Instruction Violation
- Privileged Mode Call Fault

Instead, a generic routine called "NO.INT.HANDLER" handles all the above interrupts. "NO.INT.HANDLER" is an infinite **while** loop that will, of course, hang the system when entered. An alternative implementation of "NO.INT.HANDLER" is to ignore the interrupt, immediately returning control to the executing task. The reason for looping forever is for debugging; when the system entered this routine you could examine the system state to find where the error occurred. Since there is this potential of hanging the system if one of the above exceptions occurs, all tasks, including application tasks, run in privileged mode where exceptions are ignored.

In addition, there is no interrupt vector for divide exception. A divide by zero in user or privileged mode will stall the system. Admittedly, the above hazards are a characteristic of the present, experimental system. The original design called for USER/PRIVILEGED mode implementation and interrupt handlers.

Running tasks in privileged mode, while preventing system failure from an unimplemented interrupt, does compromise software reliability. In particular, write protection is ignored in privileged mode, so a software error can be potentially disastrous (i.e. a R4 task writing into a R3 task's stack area). Likewise, an overflow or illegal instruction signals software error and the need to stop the task (for task restart or n-version programming). These signals are missed in privileged mode execution.

Even if interrupts were implemented as the original design called for, one may be reluctant to execute tasks in USER mode because its of limited power. In particular:

1. A user task cannot use system bus service routines, that is, the user cannot access system memory. User tasks attempting to access system memory stall the system (the original design calls for a write protection violation interrupt). Hence, all variables must be in local memory. Since a task might run on any processor triad from one task execution to another, local memory variables are not guaranteed to retain values between task iterations.

2. A user task can save values through use of a *task data block*. Variables in a task data block are copied from system memory into local memory by the dispatcher before the task starts, and moved back to system memory when the task ends. Thus, these variables retain their value between task iterations. However, changes to data block variables are not reflected in system memory until the task finishes, which limits the potential for inter-task communication to task completion boundaries.

Interrupt Number	Maskable	Assignment/ Function	Mode/ Effect
8	yes	unassigned	
9	yes	unassigned	
A	yes	Arithmetic Overflow[1]	USER/Stalls system PRIV/No effect
B	yes	IPC interrupt	
C	yes	Interval timer	
D	yes	Write Protection Violation[1]	USER/Stalls system PRIV/Write protection ignore
E	yes	Page Fault[4]	
F	yes	Test Adapter[4]	
10	no	Halt Instruction Execution[1]	
11	no	Illegal Opcode[1]	USER/stalls system PRIV/ignored
12	no	Stack Overflow[1]	USER/stalls system PRIV/ignored
13	no	Non-local Search Fault[1,2,4]	
14	no	Privileged instr Fault[1,2]	USER/stall system
15	no	pmcall fault[1,3]	USER/No Privileged mode routines
16	no	unassigned	
17	no	unassigned	
*	no	Divide exception[5]	USER or PRIV/Stalls system.

[1] -- No interrupt handler written. If this interrupt occurs, a routine called "NO.INT.HANDLER" is entered which executes a DO-FOREVER loop.
[2] -- Non-local Search Fault occurs when a routine attempts to access a variable in its caller's local environment that does not exist. None of FTMP's software demands non-local searches; instead, the software uses static local variables to communicate to nested procedures.
[3] -- Pmcall, Privileged mode call, is an instruction that a user process can use to call supervisor routines. There are no privileged mode routines on the current version of FTMP.
[4] -- Not tested.
[5] -- There is no interrupt vector for Divide Exception.

Figure 3-1: Summary of FTMP's Interrupts

3. Synchronization between user tasks is very limited (if not impossible) since user tasks cannot access system bus routines. The original design of FTMP does provide constraint bits in the task tables for task ordering (i.e. do not start a task until specified tasks are finished), but these bits are not implemented on the current version of FTMP.

The reliability/system capability trade-offs of running a task in USER or PRIVILEGED mode is a dilemma to the FTMP programmer. However, with minor modifications to the original design, some of the power only available in the privilege mode can be made available to a user application task. As an example, making some of the system bus routines available as traps (see interrupt number hex[15] -- pmcall fault) would give the user controlled access to system memory without compromising the software reliability of user mode execution.

Since many interrupts are not implemented on FTMP, no performance analysis was performed. The rest of the report instead concentrates on a tool for application level experiments: the synthetic workload.

4. Workload

4.1. Definition

The workload of a computer is defined as the set of all inputs (programs, data, commands) the system receives from its environment. A workload can be classify as *natural* or *synthetic*. Natural workloads accomplish useful work while a synthetic workload models a natural workload.

There are many types of natural workloads. If the computer is a timesharing system the workload would be a user typing commands to the terminal. The workload would also include overhead of loading user programs, inputing data, and executing user programs. For control computers the workload is of a different flavor; the input is in the form of sensor readings that must be processed before they are overwritten. The program task that processes the sensor data is also considered part of the control computer workload. These tasks are executed at regular intervals.

The above two situations are examples of natural system workloads. Evaluating the performance of a natural workload involves putting measurement code into an existing system and collecting workload performance data over a period of time. With the second example, a control system, evaluation would involve taking measurements on existing control software to evaluate its performance. Sensor input to the control program could be real input from the actual environment (i.e. the computer would be flying an airplane) or simulated sensor input. In either case, we assume the system and application software already exists and the major effort is in setting up the system for evaluation.

A synthetic workload, like a natural workload, exercises a computer system. But unlike a natural workload which at least must have simulated input to "real" application programs, a synthetic workload is essentially a "fake" set of application programs (or tasks) that are modeling a natural workload. A synthetic workload can test a computer without having to develop or install application software. Characteristicly, synthetic workloads are controllable by the experimenter and can be used to analyze performance by varying parameters in the synthetic workload model.

4.2. Advantages of A Synthetic Workload

As inferred from the above discussion, although a synthetic workload does not represent an application as well as a natural workload, there are several advantages to synthetic workloads:

1. Easy to create and debug. A natural workload must be written as well as have a natural or simulated external environment. If analyzing performance (perhaps for a performance improvement study), a natural workload would already exist and thus would be preferred. However, if we are performing a feasibility study where external input, let alone application software, might not exist for the system, a synthetic workload is an excellent device for measuring performance. With little effort to create and debug the synthetic workload, we could answer some feasibility questions such as "Is the computer fast enough for our target

applications?" or "Does the computer have enough capacity for the natural workload we are modeling?"

2. Easily repeatable. In an earlier section we listed several guidelines for experiments. One of those guidelines included experimental repeatability. With natural workloads repeating an experiment would involve recording all the environmental inputs over a measurement period, as well as output which might have an effect on the input. This is particularly difficult if output from the system effects the input. The natural workload approach tends to be cumbersome in terms of storage requirements. A synthetic workload not only simplifies the environment through a model but also simplifies the interface. The only data that needs to be recorded for repeat experiments is the workload parameters and the measurement period. These parameters can set the system to the exact state of the original experiment.

3. Easily controlled by parameters. The workload model is designed to make variation of parameters easy. With a parametric model, sensitivity to parameter changes can be systematically explored and bottlenecks discovered.

4. Model many natural workloads. With new computer systems we usually want to study the feasibility of using the system for many types of applications or natural workloads. Modeling these applications with a single synthetic workload can yield a good feeling for the performance of a set of natural workloads.

5. Easily migrated to different systems. Generally the same workload model can be used on several systems. Thus if we model the same workload on several computer systems it is much easier to make direct comparisons between systems. Figure 4-1 illustrates this concept. In this figure, if workload W is a natural workload it is sometimes called a *benchmark*.

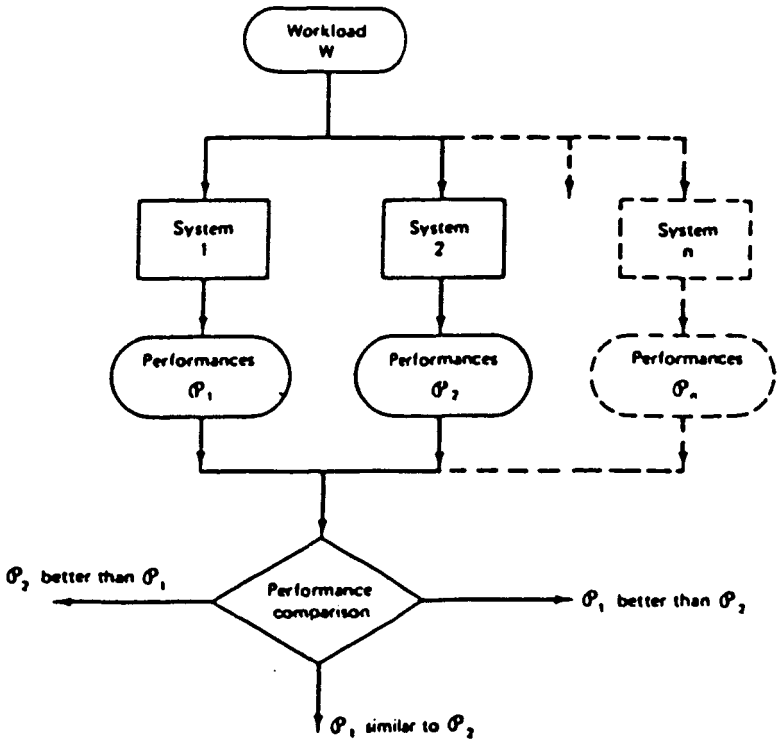

Figure 4-1: General scheme of performance comparisons among n systems [Ferrari 78]

Of course there are disadvantages to using synthetic workloads:

1. The system must be dedicated while using the synthetic workload. With natural workloads data can be collected while useful work is being done.

2. The synthetic workload is only an approximation of a natural workload.

4.3. Motivations

An additional motivation for designing a synthetic workload for FTMP is to simplify the experimentation environment (see Figure 2-5). Prior to the use of the synthetic workload, experiments were performed by creating a program on an IBM 4341 followed by compilation, assembly and linkage of the task. An absolute load module was then downloaded to the support VAX and then to FTMP for execution. The entire experimental cycle usually took up to two hours assuming the experiment was designed correctly. Analysis was limited to a few parameters in each experiment. To analyze data from the experiment the user must provide a data collection program or modify an existing data collection program. The original FTMP baseline experiments were conducted in this manner. In order to master the experimental loop, the user had to learn about the internal structure of FTMP, including the setting up of task tables, the CTA interface program between FTMP and the VAX, and the VAX/VMS command language. Because of the time it took to develop experiments, there was substantial motivation to simplify the experiment loop, even possibly taking the IBM 4341 -- the major bottleneck -- completely out of the experimental loop.

A synthetic workload relieves the user of these details as well as providing a mechanism for further simplifying experimental preparation. Synthetic workload experiments would be run by varying parameters in the model. The parameters of the synthetic workload must correspond to meaningful variables; otherwise analogies to real workloads would be meaningless. There is, of course, a fine line between representativeness and ease of use.

The next section discusses a realtime workload model. This is followed by the details of the implementation of that model on FTMP and the program support for the implementation. Finally, several workload experiments are compared to equivalent baseline experiments to *calibrate* (i.e. test the representativeness of) the synthetic workload.

4.4. A Realtime Workload Model

The goal of any model is to find a simple representation of a system that is not too far removed from the natural system. If the model is too complex, deriving conclusions from parameter changes will be difficult. Conversely, too simplistic a model would not adequately describe system behavior.

There are several factors that must be considered when developing a realtime workload model. First is

the task structure of realtime workloads. A task is a single thread of execution. With a realtime system, a task is run at regular intervals, which defines the iteration rate of that task. Not all tasks need to be run at the same iteration rate (i.e. a display terminal does not need to be updated nearly as often as the airplane flap control). Thus a realtime task model should allow for multiple iteration rates. Control systems demand task completion within the interval defined by the task iteration rate, which is referred to as a *hard deadline*. This implies that any implementation of a workload model must collect data from several task iterations to check if deadlines, and thus iteration rates, are adhered to. A realtime workload model was presented in [Clune 84]. The following discussion is an overview of that workload model.

For our model, tasks are assumed to be execution entities sharing a common memory. Each task has the form:

- read sensor data
- read interprocess communication (IPC) data
- do work (computations) on the data
- write IPC data
- write sensor data

On FTMP, a task is represented by the program in Figure 4-2. In this case the loops represent data read in (P and Q), operated on (T), and written out (R and S), with A=B+C considered the typical instruction. The communication mechanism between processes on FTMP is main memory. Thus both sensor and IPC exchanges are done through memory reads and writes. The value of the realtime clock is stored after each iteration for later timing analysis.

```
Task_i();
Begin
  Read(P_i, Q_i, T_i, R_i, S_i);
  Store(Time);
   For X=1 to P_i do
       Read Sensor Input (read memory);
  Store(Time);
   For X=1 to Q_i do
       Read IPC Input (read memory);
  Store(Time);
   For X=1 to T_i do
       Execute Instruction (A = B + C);
  Store(Time);
   For X=1 to R_i do
       Write Sensor Output (write memory);
  Store(Time);
   For X=1 to S_i do
       Write IPC Output (write memory);
  Store(Time);
End;
```

Figure 4-2: Representation of a Synthetic Workload Task

The above task model is sufficient to implement a synthetic workload on FTMP. However, if we want to more closely approximate a realtime system, a higher level structure is required.

The next abstraction level above the task is the function. A workload can consist of any number of functions, each of which is composed of one or more tasks. The parameters at the function level are:

- the number of tasks
- frequency of execution of this function. All tasks within the function will have this iteration rate.
- percentage of total system instructions used by the function
- percentage of total sensor I/O used by the function
- percentage of total IPC I/O used by the function

Tasks are grouped into a function because of parametric similarities (i.e. perform approximately the same number of operations and have the same execution rate), rather than functional similarities.

Finally, we define the system level of the model which gives the structure and capability of the overall realtime workload. Parameters at this level are:

- number of instructions (thousands of operations per second)
- total amount of sensor I/O (words per second)
- total amount of IPC (words per second)
- number of functions
- percentage of sensor I/O that is input
- percentage of IPC I/O that is input

Figure 4-3 illustrates the workload model for a realtime system.

A program, called the *workload calculator*, takes system and functional level parameters and calculates iteration numbers that can be used to implement a synthetic workload. This program, developed in [Clune 84], is discussed in Section 4.5.1.

4.5. Implementation of the Synthetic Workload on FTMP

The goal of the synthetic workload implementation is for a user to be able to use the workload with minimal knowledge of the underlying system. The user should only need to know the workload model. In addition, the workload should have an easy to use interface. Initially, the discussion of the synthetic workload implementation will focus on the user interface. This will be followed by a discussion of the details of the actual synthetic workload implementation on FTMP.

4.5.1. User Interfaces

To the user there are three parts to the synthetic workload: the *workload calculator*, the *workload generator*, and the *workload data analyzer*. Each of these programs is invoked at different times in the developing and running of a workload experiment. The following is a discussion of these three programs.

Workload Calculator:

The workload calculator was developed and implemented in [Clune 84]. This program converts parameters from the workload model into iteration numbers in a workload task on FTMP. This program inputs system and functional level parameters and calculates iteration numbers that are used by the synthetic workload generator. The

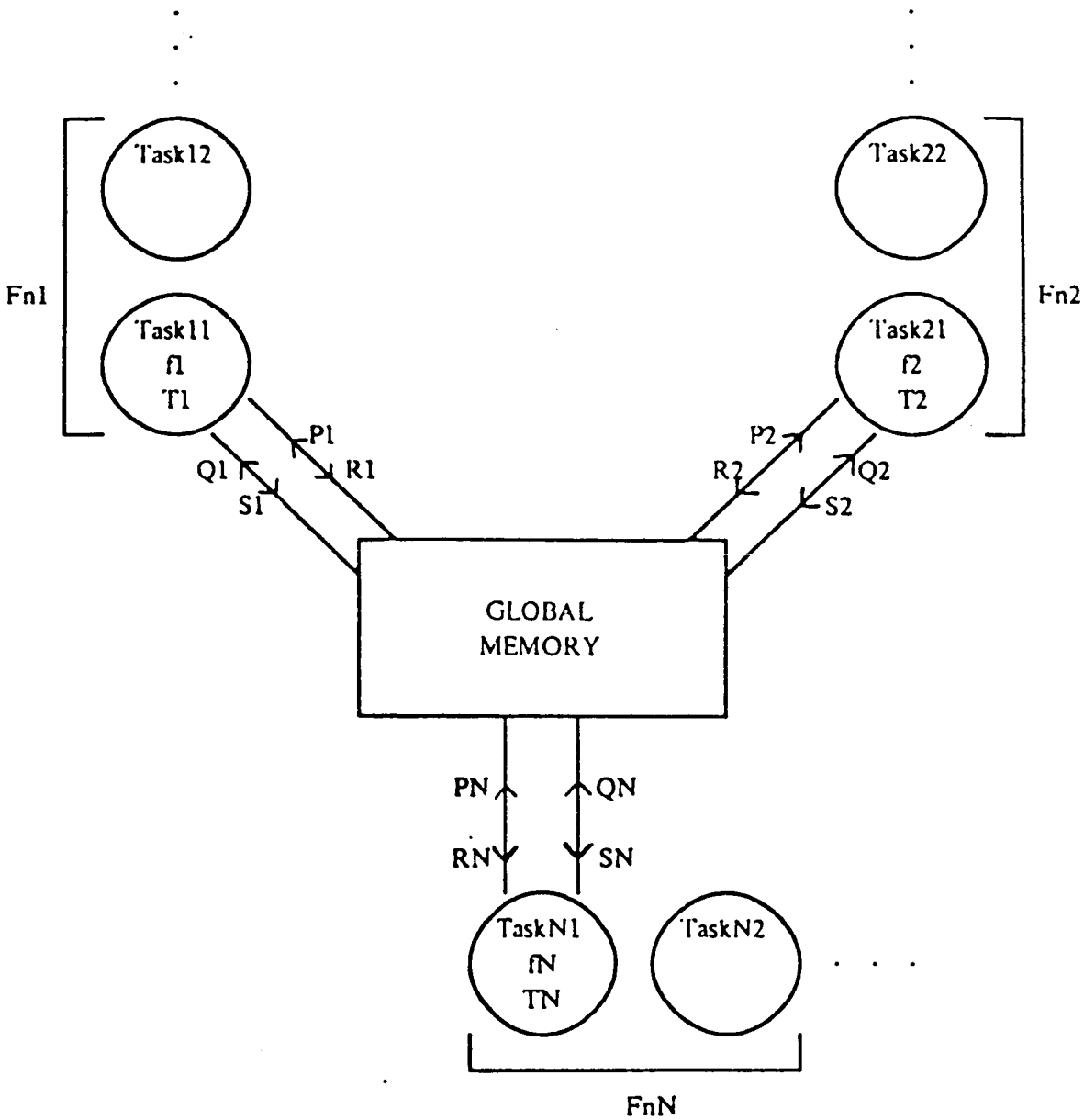

Figure 4-3: Workload Model [Clune 84]

system level parameters directly correspond to those parameters presented in the model. These parameters include total instruction KOPs, total sensor I/O, and total IPC rate. Functional level parameters also correspond to those presented in the model. Examples of functional level inputs include the number of tasks per function, the function's iteration rate and the percent of the total system instructions, the total sensor I/O, and the total IPC I/O used but each function. This program outputs loop iteration values for insertion into the synthetic workload tasks (Figure 4-2). The workload calculator can specify workloads for any control computer that implements the same workload model.

Workload Generator:
This program is the interface between the user and FTMP. The major motivation for the program is to separate the details of the workload model from the details of installing task level parameters into the FTMP synthetic workload. This program uses iteration values supplied by the user (e.g. those supplied by the workload calculator) and deposits them into synthetic workload tasks on FTMP by setting up a command file. When run, this command file enters CTA, the interface between FTMP and the VAX, and selectively writes to FTMP's memory to set up the workload. The command file also sets up the number of tasks to run in each rate group (again defined by the calculator), plus configures FTMP for one, two or three processor triads. The workload generator creates a second command file for collecting timer data from FTMP. The user is again quizzed on which timer values to save and the number of iterations to observe. These timer dumps are later analyzed by the third component of the workload, the *data analyzer*.

Data Analyzer: This program works in conjunction with the workload generator to analyze data dumps and make histograms of differences between timer values. The user is quizzed on which timer values to compare and put into histograms.

Figure 4-4 illustrates the relationship of the above programs. Each program is user oriented, quizzing the user about system configuration, workload structure, and timer values desired. Presently, the user is responsible for filling in the link between the workload calculator and the workload generator.

The steps to running an experiment with the synthetic workload are:

1. Load FTMP with the synthetic workload (need only be done once).

2. Use the *workload calculator* to describe the application workload you wish to test. Iteration values are stored in a file called RESULT.DAT.

3. Run the *workload generator* using data from Step 2 as parameters into the workload model. The workload generator will create two command files: one to configure the the synthetic workload on FTMP and a second to collect data from the workload.

4. Run the first command file to configure FTMP.

5. Run the second command file, storing the data in an output file. Run this command file several times until you have the desired amount of data.

6. Run the *data analyzer* using an output file from Step 5 as input. The data analyzer outputs

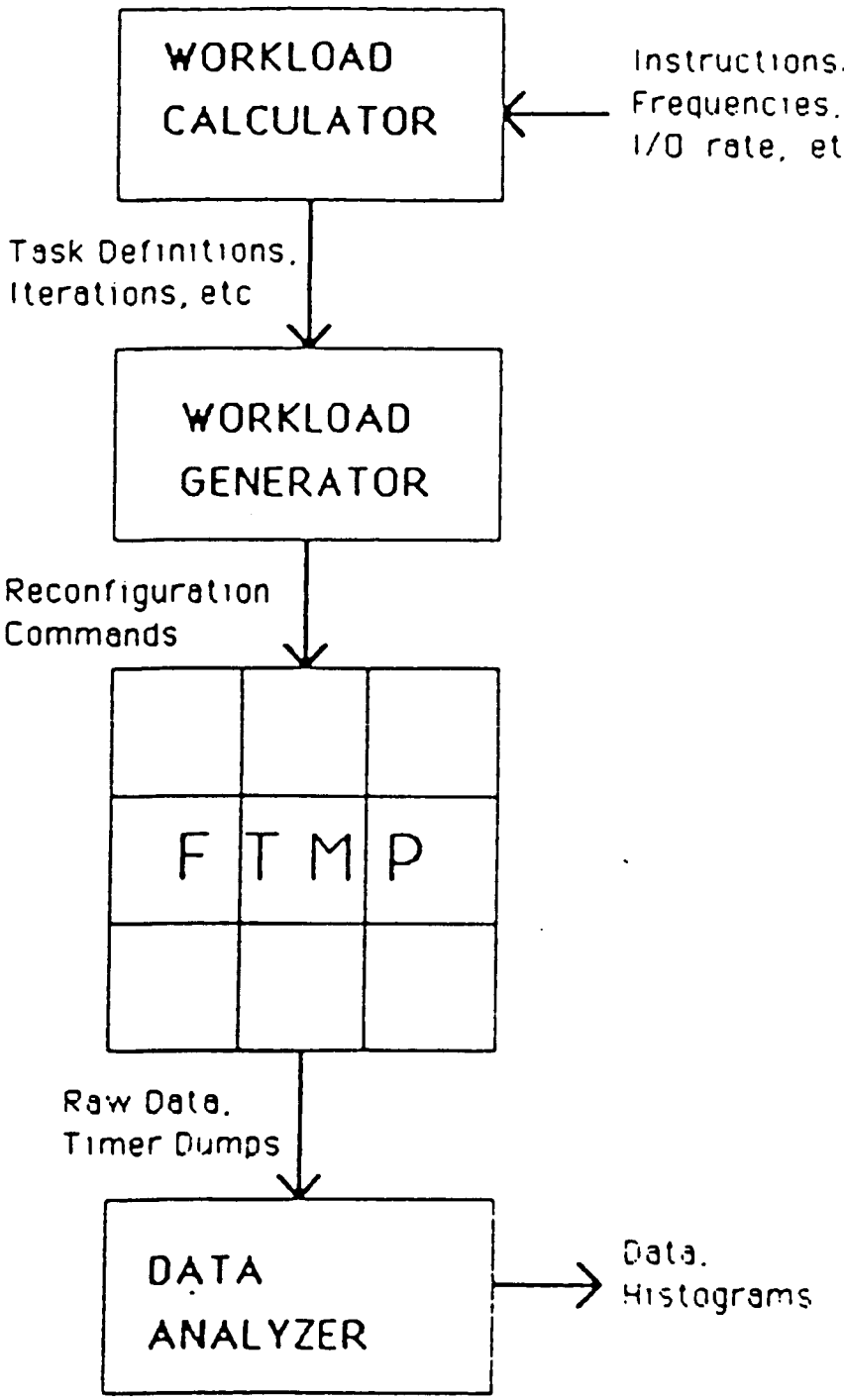

Figure 4-4: FTMP Synthetic Workload Environment

the data in a readable form and create histograms of that data.

7. Repeat Steps 2 through 6 for each workload experiment.

Once FTMP is initially loaded with the synthetic workload, the elapsed time from running the *workload calculator* to output histograms is about 10 minutes. Occasionally, a hardware interface to FTMP may stall, in which case the experiment loop can be significantly longer.

4.5.2. Implementation: FTMP Tasks and Workload Considerations

The model for a realtime workload task was presented in Figure 4-2. In this task model the values for the loop iterations are read in from a special area in memory set up by the workload generator before the workload starts. Timer values are written back to memory at the end of the task.

FTMP has three task rate groups. For initial implementation, there are three workload tasks for each rate group. Three per group is not a hard limit since there is room in the task tables to potentially expand to 15 tasks per rate group (except for the R1 rate group -- there are 6 special tasks thus limiting this rate group to 9 workload tasks). The major limit on the number of workload tasks in FTMP is memory storage for timer values. The number of tasks that actually run in each rate group is set up by the workload generator.

Data collection is done in cycles. A collection cycle starts when the data collection command file (created by the workload generator) enables tasks to execute. For a period of time workload tasks write timer values to memory. These values are then retrieved from FTMP's memory by the command file for later analysis. Once this is done tasks are enabled again to start another data collection cycle. The saved data is essentially a snapshot of the computer over a defined execution period.

To encompass all workload tasks, a collection cycle must include at least one full execution frame of the lowest frequency rate tasks (R1). Thus, a collection cycle begins at an R1 frame boundary, called a *major* frame. A major frame encompasses four R3 frames and eight R4 frames. An additional R4 task collection was added, making nine R4 collection frames, to record boundary cases such as missed deadlines. To monitor when to start collection cycles an additional R4 task is present. This task monitors when a major frame is ready to begin and sets all the workload tasks to start collecting data. It then removes itself from the R4 task list so as not to interfere with workload tasks while the workload is executing. A cycle is begun by externally linking in the special R4 task. All of these details of data collection is transparent to the user since they are set up by a data collection command file created by the workload generator.

The workload has to take into consideration several special tasks running on FTMP. These tasks are:

1. A R3 task (R31) called "TIME" which updates TIME.NOW, the current time, in memory by

checking RT.CLOCK (the realtime clock) and BASE.TIME. This is considered essential to the computer performance and is always linked in.

2. The R1 "DISPLAY" task which updates FTMP's display terminal on the status of the system. This is considered non-essential and can be taken out if the user so chooses (i.e. if a workload task already models a system display).

3. Two R1 tasks "READALL" and "SCC" which are the fault-tolerant tasks of FTMP. These two tasks can be considered essential in a fault-tolerant computer such as FTMP for fault recovery and reconfiguration. However, during fault-free execution they only perform self-tests. Therefore, the user has an option to take either of these tasks out of the task structure, which is useful should the user want to investigate the overhead of fault-tolerant tasks.

The workload generator will ask the user which special tasks to include in the workload and links them in accordingly.

Each task has an associated Task Control Block (TCB) which contains information on that task. Task Control Blocks are in a linked list common data structure in global memory. Processor triads select tasks from this structure when they need a new task to execute. Figure 2-3, presented earlier, illustrates the TCB data structure and the position of workload and other tasks in that structure. The final three R1 tasks, IDLE1, IDLE2 and IDLE3, are special tasks to record idle time in a major frame on each of the processor triads. After a processor has completed an R1 task it will select an idle task and hold that task until other processors have finished their R1 tasks and select an idle task.

Finally, the FTMP R1 task dispatcher can assign R1 tasks to a specific processor if possible. A special field in the TCB of the task determines which processor (1, 2, or 3) to run the task on with 0 specifying any processor. "SCC" modifies this field so it can progressively run a battery of self-tests on different processors. Execution of SCC effects TCB ordering since the dispatcher will postpone execution of this task until the requested processor becomes available by moving this task down the task list.

4.5.3. Calibration

The final step to synthetic workload implementation is *calibration*. Calibration determines the correctness of the workload model. The best calibration experiments are, of course, direct comparisons to natural workloads. However, comparisons to dedicated FTMP experiments is acceptable since the goal of calibration is to show that the workload can produce similar results.

The calibration experiments chosen for FTMP's synthetic workload are baseline experiments previously conducted without the workload generator in [Clune 84]. These experiments provide an opportunity for comparison. The experiment are:

1. A task switching time experiment. This finds the overhead associated with starting a new task once a task finishes. This time is found by comparing timer values recorded at the end of the first task and the beginning of the second task respectively. Figure 4-5 illustrates task

switching overhead.

2. The task startup experiments measures the overhead of starting a task on a processor. This time is found by comparing timer values taken at the beginning of tasks running on separate processors. Figure 4-6 illustrates task startup overhead.

Figures 4-7 though 4-10 are the results of four experiments: task switching time, dedicated experiment; task switching time, workload experiment; task startup overhead, dedicated experiment; and task startup, workload experiment.

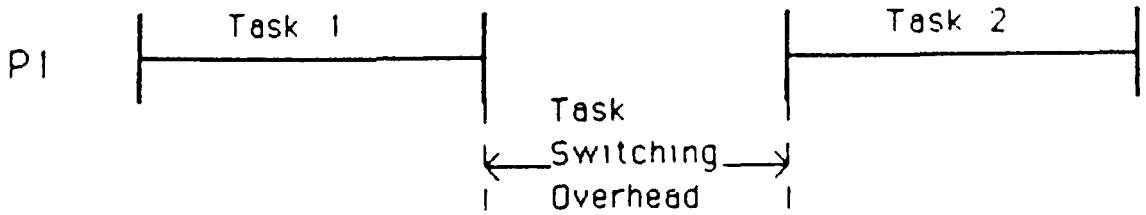

Figure 4-5: Task Switching Overhead

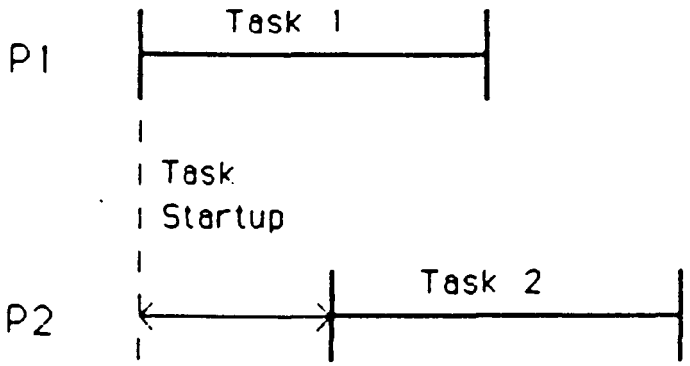

Figure 4-6: Task Startup Overhead

Initial comparison is encouraging; both baseline and workload experiments have similar shapes. Both task startup experiments reveal similar dual peak curves with fringe data points. In the baseline experiment, these lone data points revealed that the dispatcher was occasionally late starting a task. The synthetic workload exhibits the same behavior.

Closer inspection of the data reveals that the workload curves of task switching overhead and task startup time are displaced 4 and 1.88 clock ticks (1 and .47 mSec) respectively from their baseline experiment counterparts. Thus, overhead exists in the workload that is not present in the baseline experiments. The source of this overhead is obvious upon inspection of the AED source code of the baseline experiment task (Figure 4-11) and a workload task (Figure 4-12). The baseline experiment was designed to measure beginning and end task times. Thus, time is read immediately upon entering and just before exiting the task. In contrast, the workload contains both task entry overhead (statements

```
  clock              data-
  ticks     time     points
--------  ---------  ----
12 ticks (3.00 mSec) [242] ***********************************
13 ticks (3.25 mSec) [298] *******************************************

Average: 12.55 ± 0.042 Ticks (540 data points)
          3.13 ± 0.011 mSec
```

Figure 4-7: Baseline Experiment: Task Switching Overhead

```
  clock              data-
  ticks     time     points
--------  ---------  ----
16 ticks (4.00 mSec) [122] ***************************************************
17 ticks (4.25 mSec) [ 67] *****************************

Average: 16.35 ± 0.068 Ticks (189 data points)
          4.09 ± 0.017 mSec
```

Figure 4-8: Workload Experiment: Task Switching Overhead

S1-S4) and task end overhead to save results (statements E1-E4). Because the synthetic workload is an application level tool, overhead is put outside the inner loops. The workload can still be used for timing intertask events if we take into account this overhead.

By summing the execution times of statements *S1* through *S4* in the workload we can find the workload task initialization overhead. Execution times of the RD primitive are from a separate experiment (Appendix I). Execution time of arithmetic operations are taken from [Clune 84]. Execution time of the "IF" statement is neglected since global memory RD time is substantially larger.

Statement #	Instruction	Execution Time (mSec)
S1	RD [1 word]	0.138
S2	IF (EXEC4 GEQ 0)...	0.0 (for simplifying calculations)
S3		0.0
S4	RD [5 words]	0.150
		0.288 mSec (Ave.)

Similarly, the workload end overhead is:

```
  clock              data-
  ticks     time     points
--------  ---------  ----
4  ticks (1.00 mSec) [ 24] ***
5  ticks (1.25 mSec) [298] ****************************************
6  ticks (1.50 mSec) [ 48] ******
7  ticks (1.75 mSec) [  2] *
8  ticks (2.00 mSec) [ 29] ****
9  ticks (2.25 mSec) [328] ********************************************
10 ticks (2.50 mSec) [  9] *
11 ticks (2.75 mSec) [  0]
12 ticks (3.00 mSec) [  0]
13 ticks (3.25 mSec) [  1] *
14 ticks (3.50 mSec) [  0]
15 ticks (3.75 mSec) [  0]
16 ticks (4.00 mSec) [  0]
17 ticks (4.25 mSec) [  0]
18 ticks (4.50 mSec) [  1] *
19 ticks (4.75 mSec) [  0]
20-30 ticks          [  0]
31 ticks (7.75 mSec) [  0]
32 ticks (8.00 mSec) [  3] *
33 ticks (8.25 mSec) [  0]
34 ticks (8.50 mSec) [  1] *


Average:  7.15 ± 0.198 Ticks (744 data points)
          1.79 ± 0.014 mSec
```

Figure 4-9: Baseline Experiment: Task Startup Time

Statement #	Instruction	Execution Time (mSec)
E1	WRT [12 words]	0.190
	EXEC4*6	0.063
E2	WRT [1 word]	0.164
	3*EXEC4	0.063
E3	EXEC4=EXEC4+1	0.058
E4	WRT [1 word]	0.164
		0.702 mSec (Ave.)

In the synthetic workload, calculation of task switching must consider task ending overhead of the first task, and task initialization overhead of the second task. Finally, 0.164 mSec is added since the baseline experiment must write a timer value to memory (E1) at the end of the task. Taking these into account, we get

$$4.09 \text{ mS} - 0.288 \text{ mS} - 0.702 \text{ mS} + 0.164 \text{ mS} = 3.26 \text{ mS (Ave.)}$$

a value within 5 percent of the baseline experiment's value.

Similarly, overhead should be deducted from the task startup time experiment. Since this experiment compares the first timer values of two workload tasks, task initialization overhead for both tasks should

```
  clock              data-
  ticks     time     points
--------  ---------  ----
 5 ticks (1.25 mSec) [ 18] ******
 6 ticks (1.50 mSec) [ 95] ********************************
 7 ticks (1.75 mSec) [ 21] *******
 8 ticks (2.00 mSec) [  0]
 9 ticks (2.25 mSec) [  2] *
10 ticks (2.50 mSec) [ 51] ****************
11 ticks (2.75 mSec) [108] ************************************
12 ticks (3.00 mSec) [  0]
13 ticks (3.25 mSec) [  1] *
14 ticks (3.50 mSec) [  1] *
15 ticks (3.75 mSec) [  0]
16 ticks (4.00 mSec) [  0]
17 ticks (4.25 mSec) [  0]
18 ticks (4.50 mSec) [  0]
19 ticks (4.75 mSec) [  0]
20 ticks (5.00 mSec) [  0]
21 ticks (5.25 mSec) [  1] *
22 ticks (5.50 mSec) [  3] *
23 ticks (5.75 mSec) [  0]
24 ticks (6.00 mSec) [  1] *
25 ticks (6.25 mSec) [  2] *
26 ticks (6.50 mSec) [  2] *

Average:  9.03 ± 0.391 Ticks  (306 data points)
          2.26 ± 0.098 mSecs
```

Figure 4-10: Workload Experiment: Task Startup Time

be deducted. The actual startup time becomes:

2.26 mSec - 2*0.288 mSec = 1.68 mSec (Ave.)

a value within 10 percent to the baseline experiment's value.

The following table summarizes the above results:

Experiment	Baseline Experiment Times	Workload Experiment Times
Task Switching time	3.13 mSec (Ave.)	4.09 mSec
Minus workload overhead	--	3.26 mSec
Task Startup time	1.79 mSec	2.26 mSec
Minus Workload overhead	--	1.68 mSec

Although these experiments are not application level calibration experiments, they do show that the synthetic workload is a valid tool for making baseline experiments, as long as workload overhead is considered in any intertask measurements. If measurements are intratask, the overhead is much smaller

```
CMU.TEST1 BEGIN
    ...

   DEFINE PROCEDURE TIMETEST1 TOBE
      BEGIN
         LONG HOLD,HOLD1;
         INTEGER EXEC,RTCNUM,I;
         INTEGER A;

         HREAD(RT.CLOCK,HOLD,2);
         RD(CMU.EXEC,EXEC,1);
         IF EXEC LEQ 14
            THEN BEGIN
              RD(CMU.RTCNUM,RTCNUM,1);
               FOR I=1 STEP 1 UNTIL RTCNUM
                   DO BEGIN
                      A=1;
                   END;
               A = EXEC * 6;
               WRT(CMU.TIME(A),HOLD,2);
               HREAD(RT.CLOCK,HOLD1,2);
               WRT(CMU.TIME(A+1),HOLD1,2);          E1
            END;
         RESUME(0);
      END;
   END FINI;
```

Figure 4-11: Baseline Experiment Task (AED)

since the clock read time (HREAD) is the only overhead. In conclusion, the workload is a useful tool for performing experiments on FTMP.

```
CMU.TEST BEGIN
  ...
 DEFINE PROCEDURE WRKLOADR41 TOBE
   BEGIN
     INTEGER X, Y, Z;       ... NON-STACK LOCALS //
     OWN INTEGER A;
     OWN INTEGER LOCAL, EXEC4;
     OWN LONG ARRAY HOLD(OUT.VALUES); ...HOLDS TIMER VALUES //
     OWN INTEGER ARRAY R41.INPUT(5);  ...INPUT PARAMETERS   //
     INTEGER P; P $=$ R41.INPUT(0);
     INTEGER Q; Q $=$ R41.INPUT(1);
     INTEGER T; T $=$ R41.INPUT(2);
     INTEGER R; R $=$ R41.INPUT(3);
     INTEGER S; S $=$ R41.INPUT(4);

       RD(CMU.EXEC(0),EXEC4,1);                                  S1
       IF (EXEC4 GEQ 0) AND (EXEC4 LES 9) THEN                   S2
         BEGIN                                                   S3
           RD (R4.INPUT(0),R41.INPUT,5);                         S4
           HREAD(RT.CLOCK,HOLD(0),2);
           FOR A=1 STEP 1 UNTIL P DO
              RD(CMU.GLOBAL,LOCAL,1);
           HREAD(RT.CLOCK,HOLD(1),2);
           FOR A=1 STEP 1 UNTIL Q DO
              RD(CMU.GLOBAL,LOCAL,1);
           HREAD(RT.CLOCK,HOLD(2),2);
           FOR A=1 STEP 1 UNTIL T DO
             X=Y+Z;
           HREAD(RT.CLOCK,HOLD(3),2);
           FOR A=1 STEP 1 UNTIL R DO
             WRT(CMU.GLOBAL,LOCAL,1);
           HREAD(RT.CLOCK,HOLD(4),2);
           FOR A=1 STEP 1 UNTIL S DO
             WRT(CMU.GLOBAL,LOCAL,1);
           HREAD(RT.CLOCK,HOLD(5),2);
           WRT(R41.OUTPUT(EXEC4*6),HOLD,12);                     E1
           WRT(R4.ID(3*EXEC4),TRIAD.ID,1);                       E2
           EXEC4 = EXEC4 + 1;                                    E3
           WRT (CMU.EXEC(0),EXEC4,1);                            E4
         END;    ... IF (EXEC4 GEQ 0) AND .. //
      RESUME(0);
    END;

END FINI;
```

Figure 4-12: Synthetic Workload Task (AED)

5. Future Work

Although much work has been done defining the experimental methodology and using it to validate FTMP, there is still work to be done. First, the methodology should be verified through application to another system. In particular, the Software Implemented Fault-Tolerant (SIFT) computer at AIRLAB should have the validation steps applied to it. This computer has constraints similar to FTMP's and would be an excellent candidate for the validation procedure.

On FTMP, a few remaining baseline experiments should be performed. These include:

- Measure the time to transfer varying blocks of data from global to local memory, varying parameters much more than was done in the brief RD/WRT experiments described in Appendix I.
- Measure instruction execution time in pairs to see if the result is equivalent to the sum of the execution times when the instructions were measured singly.
- Investigate overhead and variation in application software due to the fault-tolerant mechanisms of FTMP.
- Find the nominal length of R3 and R1 tasks on FTMP.
- Find context swap time. This time is defined as the amount of time it takes to start up an R3 task once the dispatcher finishes with R4 tasks.

The later three experiments can probably be performed with the synthetic workload.

The potential of the synthetic workload has only been superficially demonstrated. The workload should be used for performance tests and comparisons, along with application level baseline experiments. Only through use will its power be demonstrated.

Also, the present synthetic workload is a minimal implementation that was used to investigate feasibility. Presently, there are only three tasks per rate group. The R4 and R3 rate groups each have room for ten more tasks in their task structure, while the R1 rate group has room for seven more tasks. The only limiting factor is the amount of global memory available on FTMP to hold timer dumps. More compact timer dumps could possibly resolve this problem. Any enhancements will require changing the workload generator and data analyzer.

Finally, in the future it will be desirable to contrast performance versus reliability of fault-tolerant computers. One idea is to integrate the synthetic workload -- a performance measurement tool -- with the fault-injection experiments.

6. Conclusion

This project outlined and refined an experimental methodology for validating the multiprocessor avionics computer, FTMP. The methodology emphasizes a building block approach in which tests are performed starting at the instruction level, progressing through the operating system level and finally up to application level validation. At each level baseline experiments, which test a single phenomenon, were performed. These were followed by more sophisticated experiments which test interactions between several baseline phenomenon. Finally, the concept of a generalized application level experiment tool, called the synthetic workload, was developed.

Previous research had developed an outline of the methodology and tested it through the application level. This research refined that methodology with additional baseline tests. In addition, the synthetic workload was implemented as an application level tool. The synthetic workload was then calibrated with a baseline experiment to demonstrate the workload's representativeness.

Although the technique was developed specifically for FTMP the origin of the technique dates back to earlier work on multiprocessors at C-MU. Thus, the methods used here should be applicable to other computer systems. Tests on another system will supply information on the robustness of the technique along with supplying meaningful comparisons between systems.

By no means is the methodology complete. Using the synthetic workload for experiments will undoubtedly reveal deficiencies in the original methodology. But the existence of this tool will greatly improve productivity, allowing researchers to run more experiments and further refine the methodology. In general, the methodology has proven to be a sound approach to validating computer systems.

I. Test of Select RD/WRT Primitives

On FTMP, most program tasks access the shared system memory with the following bus service routines:

- RD(*sys.adr,cache.adr,num*). This routines transfers *num* number of words from system memory address *sys.adr* to cache address *cache.adr*.
- WRT(*sys.adr,cache.adr,num*). This procedure is the same as RD except, of course, the direction of transfer is reversed.

We wish to find the time these procedures use to access system memory with varying transfer sizes. In particular, we are interested in the sizes that are used in the workload. The following instructions were tested:

1. RD(sys,cache,**1**)
2. WRT(sys,cache,**1**)
3. RD(sys,cache,**5**)
4. WRT(sys,cache,**12**)

Instructions 1 and 2 were each executed in a loop 100 times along with the instruction 'A=1;'. The other two instructions were executed in a similer loop 50 times[2]. To find loop overhead, a loop just containing an 'A=1;' instruction was executed both 50 and 100 times. This is the 'NULL' loop[3]. Times to execute instructions can be found by subtracting loop overhead from the instruction loop, leaving only instruction execution time.

The results of the measurements were as follows:

Instruction	clock ticks per loop(Ave.) /loop count	μSec per instruction w/ overhead	μSec per instruction w/o overhead	Number of data points
1) Null	15.7/100	39.3 ± 0.019	0.0	340
2) RD (x,y,num=1)	70.8/100	177.0 ± 0.025	137.7 ± 0.044	220
3) WRT(x,y,num=1)	69.1/100	172.8 ± 0.023	133.5 ± 0.042	260
4) Null	8.3/50	41.5 ± 0.027	0.0	500
5) RD (x,y,num=5)	38.2/50	191.0 ± 0.025	149.5 ± 0.052	500
6) WRT(x,y,num=12)	46.0/50	230.0 ± 0.018	188.5 ± 0.045	300

The first column is the raw data in clock ticks (1 clock tick = .25 mSec). The next column is the time to execute a single instruction including loop overhead. The third column adjusts the time from the second by subtracting overhead.

---

[2]The loop count was reduced to 50 for these calls since many large block transfers could take more time than an R4 process is allowed

[3]A loop must contain at least one instruction; otherwise the compiler will not accept it. This is why 'A=1' is used as a substitute for a 'NULL' loop

II. Example of Workload Use

This appendix contains an example of the running of the workload *generator* and *data analyzer*. An example of the running of the workload *calculator* is not presented since that program is discussed in [Clune 84]. This example starts with the very first step of the user providing information to the workload generator followed by the loading of FTMP with the synthetic workload. Then, using the two command files produced by the generator, the FTMP synthetic workload is configured and data collection is run. Output from the data collection is redirected into a file which is used as input to the workload data analyzer.

The workload *generator* basically queries the user on how he/she wants the synthetic workload configured. Input parameters to tasks correspond directly to workload parameters in Figure 4-2. The workload generator will also ask if the user wants the special R1 tasks (SCC, READALL, and DISPLAY) included in the workload. Finally, this program will inquire about data collection including what values and how many iterations the user wants from the workload collection.

The workload *data analyzer* is more complicated. This program reads in timer values produced by the collection file generated by the generator and quizzes the user on which timer values to compare. The initial part of the analyzer is file management. The program skips comments and tables in the data file to find the start of the workload data. It then quizzes the user on where he/she wants output sent. Should there be a break due to garbage data, a new collection set, or incomplete data (i.e. CTA stalled in the middle of a collection and had to be restarted), this program will skip to the next major frame of data and return to the file management prompt.

Next, the Analyzer gets from the user timer values to compare. The format for specifying timer values is:

```
<task name> <timer no>
where <task name> ::= READAL, SCC, IDLE[123], R[431][123]
      <timer no>  ::= 0-5 for Rxx tasks.
                      6+ for timer value in another collection frame.
                      0-1 for READALL, SCC and IDLE task.
```

Figure II-1 illustrates the workload tasks and timer numbers. For Rxx tasks, the user can specify a number greater than 5 to refer to a timer value in another collection frame, e.g. 6 corresponds to the 0th timer value in the task iteration immediately after the current iteration. Thus, to find the time between running of task R41 we would compare **R41 5**, the last timer value in task R41, to **R41 6**, the first timer value in the next R41 iteration. This is feasible since the timer values for all iterations of a task in a major frame are stored in a continuous array. The analyzer will try to collect as many data points as possible in a major frame.

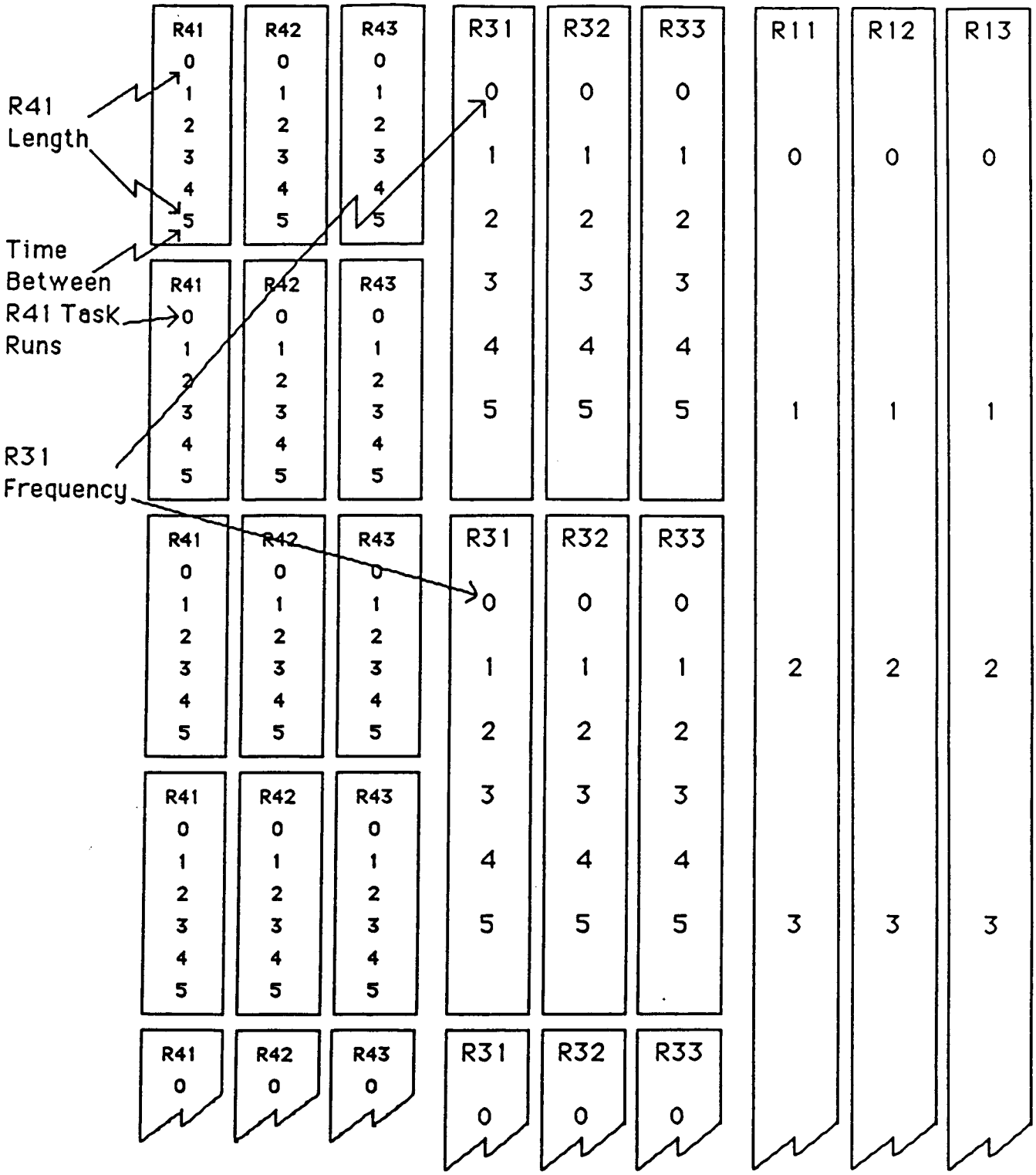

Figure II-1: Illustration of Workload Tasks

It is recommended that the reader look at the steps for running the workload presented in Section 4.5.1 while reading through this example. Figure II-2 illustrates the running of the workload. '.COM' files contain CTA commands for loading FTMP with the synthetic workload (2TRIAD.COM), configuring the workload (CONFIG.COM), and collecting data from the workload (COLLECT.COM). WRKLD.CAP is the absolute load module of the synthetic workload. WRKLD.LOG is an output log of workload data produced through the collection command file (COLLECT.COM). WRKINFO.TXT is an internal file that communicates workload information from the workload generator to the data analyzer.

Throughout this appendix the user response will be in bold font while *italicized* phrases are guiding comments. Space constraints require that the example be minimal. Therefore, data collection is for eight major frames of data. This is much less than would be included in a normal experiment.

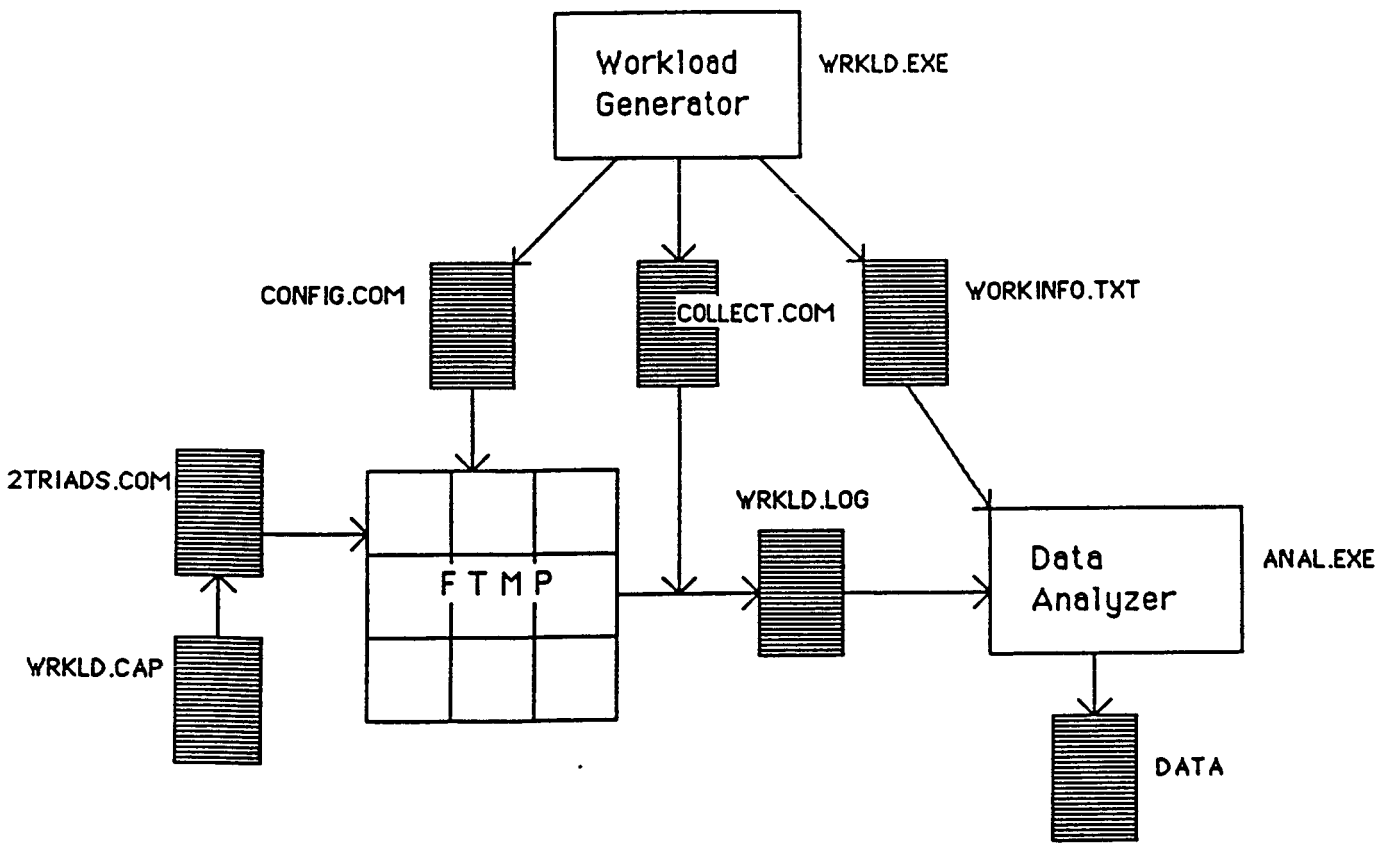

Figure II-2: Running the FTMP Workload

```
$ RUN WRKLD
        Input file [STDIN]: <CR>
        Output file [STDOUT]: CONFIG.COM
        No. of R1 tasks: 0
        No. of R3 tasks: 1
        Task R31:
        Time limit in ticks (1 tick=0.25 msec) [48 ticks]: <CR>
        Input parameters [? or (P Q T R S)]: 0 0 0 0 0
```

```
No. of R4 tasks: 2
Task R41:
Time limit in ticks (1 tick=0.25 msec) [24 ticks]: <CR>
Input parameters [? or (P Q T R S)]: 0 0 0 0 0
Task R42:
Time limit in ticks (1 tick=0.25 msec) [24 ticks]: <CR>
Input parameters [? or (P Q T R S)]: 0 0 0 0 0
How many processor triads (1, 2, or 3)? 2
Do you want SCC linked in [Y]? <CR>
Do you want DISPLAY linked in [Y]? <CR>
Do you want READALL linked in [Y]? <CR>
Data for collection
Do you want the data collection loop in a separate file? [n] y
Output file [STDOUT]: COLLECT.COM
Wait time between collections [6 secs]: <CR>
There are 2 R4 tasks.
      How many of these tasks do you want data from? [ALL] <CR>
There are 1 R3 tasks.
      How many of these tasks do you want data from? [ALL] <CR>
Do you want the ID table dumped? [YES] <CR>
Do you want IDLE, SCC, and READALL values dumped? [YES] <CR>
Loop iterations [25]: 8
```

```
$ @2TRIADS.COM          Load FTMP with the synthetic workload.
Output from loading...
Bit set
.     THIS PROGRAM STARTS UP 2 PROCESSOR AND MEMORY TRIADS.
.
.     MEMBERS OF TRIAD1 ARE LRU'S 0, 1 AND 2.
.
.     MEMBERS OF TRIAD2 ARE LRU'S 3, 4 AND 5.
.
.     THE MASTER IS LRU "A".
.
.     COOP.CAP LOADED IN MASTER
.
.     MASTER ISSUING BUS ENABLE/SELECT COMMANDS.
.
.   CLEARING SYSTEM MEMORY TO 0
.
.     BEGINNING LOAD OF EXEC MEMORY IMAGE
.
.     SYSTEM MEMORY LOAD COMPLETE
.
.     LRU'S 6,7,8,9,A,B ARE MARKED FAILED.
.
.     TRIAD.ID.TABLE, MRR.TABLE SHOULD BE ALTERED TO CHANGE
.     THIS CONFIGURATION.
.
.     SLOP IS SET TO 40 PER CENT OF R4 PERIOD.
.
.     STARTING 2 TRIADS
.
.     MASTER MAKING FINAL BUS ASSIGNMENTS
```

```
.
.
.   SYSTEM STARTED IN MULTIPROCESSOR MODE.
.
.   CONFIGURATION TABLES ARE LOCATED AS FOLLOWS:
.
.
.           TABLE              LOCATION  LENGTH
.
.  BUS INMUX SELECT CODE    0  20     12
.
.  C BUS ASSIGNMENTS        0  20     12
.
.  P, R AND T BUS ASSGN     0  38     12
.
.  MEMORY STATUS            0  44     12
.
.  PROCESSOR STATUS         0  50     12
.
.  ERROR LATCHES            1  00     48
.
. INITIATING TRANSFER OF CLOCK FROM MASTER
.
Bit is reset
.    DISCONNECTED FROM C BUS 1
.    DISCONNECTED FROM C BUS 2
.    DISCONNECTED FROM C BUS 3
.    DISCONNECTED FROM C BUS 4
.    DISCONNECTED FROM C BUS 5
.
$ @CONFIG
```

Output from configuring...

```
.
. Linking in DISPLAY .
.
.
.   Preparing R1 tasks   .
.
.     0 R1 tasks .
.
.
.   Preparing R3 tasks   .
.
.     1 R3 Tasks   .
.
.
.   Preparing R4 tasks   .
.
.     2 R4 Tasks   .
.
.
.   Bringing up 2 Processors
.
```

```
.   Repairing 0-2 .
.
.
.   Failing 3-8 .
.
.
.  Bringing up Processors 3-5 .
.
.
.  Linking in IDLE and (optionally)
.         SCC, DISPLAY and READALL
.
```

```
$ @COLLECT /OUTPUT:WRKLD.LOG
```
All output going a file
```
$ RUN ANAL
```
Send Output to the terminal
```
  Input file [STDIN]:  wrkld.log
.
.  STARTING COLLECTION .
.
.
. TABLES OF INTEREST  LRU assignment table and
.                     table of workload input
        0020 0020 0016 0016 0016 0015 0015 0015 ! 0000 0050   2 processor
                                               0020 ! 0000 0058   triads
        0000 0000 0000 0000 0000 0000 0000 0000 ! 000F 0000
        0000 0000 0000 0000 0000 0000 0000 0000 ! 000F 0008
        0000 0000 0000 0000 0000 0000 0000 0000 ! 000F 0010
        0000 0000 0000 0000 0000 0000 0000 0000 ! 000F 0018
        0000 0000 0000 0000 0000 0000 0000 0000 ! 000F 0020
                    0000 0000 0000 0000 0000 ! 000F 0028
        000A 042F 000A 042E 000A 042D 000A 042D ! 0010 0000
        %Start of new data.
        Where do you want new data (S,#,N,L,?): N
        New output file [STDOUT]: <CR>
EATING DATA...
```
For this running of the workload we will collect data to measure four things:

- *The R41 task length. This is calculated by subtracting the first timer value in task R41 (**R41 0**) from the last timer value in that task (**R41 5**).*
- *The time for the second processor to start its R4 task after the first processor started its R4 task. This "task startup" time is found by comparing timer values taken at the beginning of tasks R41 and R42 (**R41 0** and **R42 0**).*
- *The effective rate of an R3 task. This is done by comparing time at the beginning of each iteration of the first R3 task (**R31 0** to **R31 6**). There are four R3 task iterations per major frame of data. Thus, **three** values can be collected in a major frame.*
- *SCC startup time. This is a measure of the time for SCC to start after the first R4 task starts. It is found by comparing the first timer value in SCC (**SCC 0**) with the first timer*

*reading (**R41 0**) of the first iteration of task R41.*

```
There are:
        2 R4 tasks, 2 are dumped.
        1 R3 tasks, 1 are dumped.
        0 R1 tasks, 0 are dumped.
        The task ID table was dumped.
        SCC, READALL and IDLE task values were dumped.
        Data point dump 1.  Please list highest rate group first.
                First timer value (cmd,Q,H,?) [?] > R41 0
                Second timer value > R41 5
                Name of this data dump:  Task R41 length
        Data point dump 2.  Please list highest rate group first.
                First timer value (cmd,Q,H,?) [?] > R41 0
                Second timer value > R42 0
                Name of this data dump:  Task Startup time
        Data point dump 3.  Please list highest rate group first.
                First timer value (cmd,Q,H,?) [?] > R31 0
                Second timer value > R31 6
                Timer number for 2nd task crosses a frame boundary.
                How many collections do you want per dump group? [?] > <CR>
                        Normal collection values are: 9 (R4) and 4 (R3).
                        Use a number that is less than default or
                            you'll go out of bounds on the data structure.

                How many collections do you want per dump group? [?] > 3
                Name of this data dump:  R3 task rate
        Data point dump 4.  Please list highest rate group first.
                First timer value (cmd,Q,H,?) [?] > R41 0
                Second timer value > SCC 0
                Which R4 task iteration do you want? [0-8] 0
                Name of this data dump:  scc startup time
        Data point dump 5.  Please list highest rate group first.
                First timer value (cmd,Q,H,?) [?] > Q
    4     10    403    378
    4      6
    4     10    635
    4      6
    4     12    380
    4      6
    4     11
    4      7
    4     11
    4     11    396     89
    4      6
    4     11    638
    4      6
    4     14    380
    4      6
    4     10
    4      7
    5     11
    4     11    638    294
    3      6
```

```
4      12     389
4       6
5      11     642
4       6
4      11
5       7
4      10
4      11     638     443
4       6
4      10     381
5       7
3      11     641
4       6
4      10
4       7
5      11
5      11     640     283
3       6
5      11     380
4       6
5      11     638
4       6
4      10
4       7
4      11
5      11     639      84
4       6
4      11     380
4       6
4      11     642
4       6
5      11
4       7
4      11
4      10     637      57
3       6
5      11     382
4       6
4      11     643
4       6
5      11
4       7
4      10
5      11     638     298
4       6
4      16     381
4       6
4      10     642
4       6
4      11
4       7
4      10

     >>Task R41 length.
```

```
        AVERAGE = 4.125000  (72 Data points)
        VAR = 0.223592 (ST. DEV. = 0.472855)
        MAX = 5 MIN = 3
Print histogram of Task R41 length [Y]? <CR>
    3 (  4) ****
    4 ( 55) *******************************************************
    5 ( 13) *************

        >>Task Startup time.
        AVERAGE = 8.888889  (72 Data points)
        VAR = 6.269163 (ST. DEV. = 2.503830)
        MAX = 16        MIN = 6
Print histogram of Task Startup time [Y]? <CR>
    6 ( 23) ***********************
    7 (  9) *********
    8 (  0)
    9 (  0)
   10 ( 11) ***********
   11 ( 25) *************************
   12 (  2) **
   13 (  0)
   14 (  1) *
   15 (  0)
   16 (  1) *

        >>R3 task rate.
        AVERAGE = 533.458333  (24 Data points)
        VAR = 12412.259040 (ST. DEV. = 128.110339)
        MAX = 643       MIN = 380          The spread is too large to print
Print histogram of R3 task rate [Y]? no

        >>scc startup time.
        AVERAGE = 240.750000  (8 Data points)
        VAR = 21286.214844 (ST. DEV. = 145.897960)
        MAX = 443       MIN = 57
Print histogram of scc startup time [Y]? no
Merge any of the data sets? no


$
```

References

[Clune 84] Ed Clune.
Analysis of the Fault-Free Behavior of the FTMP Muliprocessor System: Baseline Measurements and Synthetic Workload Development.
Master's thesis, Carnegie-Mellon University, 1984.

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

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

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

[Draper 83d] *Development and Evaluation of a FTMP Computer, Vol IV, FTMP Executive Summary*
Charles Stark Draper Laboratories, 1983.

[Feather 84] Frank Feather, Carlos Liceaga.
FTMP Programmer's Manual
2nd edition, 1984.

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

[Hopkins 78] Hopkins, A.L., et.al.
FTMP - A Highly Reliable Multiprocessor.
IEEE Trans. on Computers , October, 1978.

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

(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.

[Singh 81] Ajay Singh.
Pegasus: A Controllable, Interactive, Workload Generator for Multiprocessors.
Master's thesis, Carnegie-Mellon University, 1981.

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

[Wensley 78] Wensley, J.H., et.al.
SIFT: A Computer for Aircraft Control.
IEEE Trans. on Computers , October, 1978.