Task Scheduling on the PASM Parallel Processing System

DAVID LEE TUOMENOKSA, MEMBER, IEEE, AND HOWARD JAY SIEGEL, SENIOR MEMBER, IEEE

Abstract—**PASM is a proposed large-scale distributed/parallel processing system which can be partitioned into independent SIMD/MIMD machines of various sizes. One design problem for systems such as PASM is task scheduling. The use of multiple FIFO queues for non-preemptive task scheduling is described. Four multiple-queue scheduling algorithms with different placement policies are presented and applied to the PASM parallel processing system. Simulation of a queueing network model is used to compare the performance of the algorithms. Their performance is also considered in the case where there are faulty control units and processors. The multiple-queue scheduling algorithms can be adapted for inclusion in other multiple-SIMD and partitionable SIMD/MIMD systems that use similar types of interconnection networks to those being considered for PASM.**

Index Terms—**Distributed processing, multimicroprocessor systems, multiple-SIMD systems, parallel processing, partitionable SIMD/MIMD systems, PASM, performance evaluation, reconfigurable computer systems, scheduling.**

I. Introduction

As a result of the advances in microcomputer technology, it is now feasible to build large-scale parallel processing systems. Two types of partitionable parallel processing systems are multiple-SIMD (e.g., MAP [14], [15]) and partitionable SIMD/MIMD (e.g., DADO [23], PASM [22], TRAC [20]). A *multiple-SIMD system* can be dynamically reconfigured to operate as one or more independent *SIMD (single instruction stream–multiple data stream) machines* [5], [24] of various sizes. Illiac IV was originally designed to be a multiple-SIMD system [2]. A *partitionable SIMD/MIMD* system can be dynamically reconfigured to operate as one or more independent SIMD/MIMD machines of various sizes. An *SIMD/MIMD machine* can operate as either an SIMD machine or *MIMD (multiple instruction stream–multiple data stream) machine* [5], [24] and can dynamically switch between the SIMD and MIMD modes of operation (e.g., CAIP [10]).

The advantages of a partitionable parallel processing system include: it allows multiple users to be executing tasks simultaneously, it permits the size of a virtual machine to be adjusted to meet the needs of a task, it is fault-tolerant, and it can overlap the execution of different subtasks of an overall task [22]. In this paper, the problem of task scheduling on partitionable parallel processing systems is considered. Tasks must be scheduled so that the processor utilization is maximized while trying to minimize the average user response time. PASM is a partitionable SIMD/MIMD system that is being designed as a tool for studying parallel algorithms for image and speech understanding applications (e.g., image contour extraction [28]) [22]. One of the motivations for this study is to choose a scheduling algorithm to implement on the PASM prototype which is currently being constructed at Purdue University [11]. Two methods for task scheduling have been considered for use with the PASM operating system [29]. The first method is the application of two-dimensional bin packing techniques [26]. With this method the execution time of the task must be known in order to schedule the task. This limitation is unacceptable in the target environment for PASM. The second method, which removes this limitation, makes use of multiple task queues.

The use of multiple queues has been applied to scheduling on uniprocessor systems. In [7], a long range scheduling strategy is described in which a job is inserted into the job queue by the priority which it has accumulated in previous passes through the service facility. The scheduling policy makes use of several queues s_i, each with a given threshold t_i. A job will be put into queue s_i when its accumulated service time exceeds t_i while being less than t_{i+1}. When a job first enters the system it is put into queue s_1. The example discussed in [7] makes use of three queues. A method similar to this has been implemented for process scheduling under the UNIX[1] Time-Sharing System [19].

Coffman and Denning in [3] discuss the use of nonpreemptive priority queues for job scheduling. Jobs are given preferential treatment based on the priorities with which they are associated. Each priority level has a corresponding queue. Although this is similar to the discussion in [7], it differs in that a job has a fixed priority. In a parallel processing system, the fixed priority can be determined by the number of processors a task requires.

Four multiple-queue scheduling algorithms are presented in this paper. When a task is to be scheduled by the multiple-queue algorithms, it is placed in one of the queues depending on a given characteristic of the task. In PASM the queue assignment is determined by the number of processors that the task requires. The performance of the scheduling algorithms is determined via simulation. The simulator, which runs tasks through a queueing network model of PASM, is used to deter-

Manuscript received March 31, 1983; revised August 29, 1984. This work was supported by the Rome Air Development Center under Grant F30602-83-K-0119, by a Purdue University Graduate Fellowship, and by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under Grant AFOSR-73-3581.

D. L. Tuomenoksa was with the PASM Parallel Processing Laboratory, School of Electrical Engineering, Purdue University, West Lafayette, IN 47907. He is now with AT&T Information Systems, Lincroft, NJ 07738.

H. J. Siegel is with the PASM Parallel Processing Laboratory, School of Electrical Engineering, Purdue University, West Lafayette, IN 47907.

[1]UNIX is a trademark of AT&T Bell Laboratories.

mine processor utilization, response time, and other performance measures [27]. It is noted that in cases where it is difficult or impossible to develop an analytical model (such as this), simulation is an accepted alternative for determining system performance.

Section II is an overview of PASM. In Section III the scheduling algorithms are described. The results of the simulation studies are presented and analyzed in Section IV. In Section V the effect of hardware faults on system performance is considered. A modification to the multiple-queue scheduling algorithms that improves system performance in certain cases by providing a limited form of multiprogramming is presented in Section VI. The application of multiple-queue scheduling algorithms to a general model for partitionable parallel processing systems is discussed in Section VII.

II. PASM Overview

PASM, a partitionable SIMD/MIMD machine, is a large-scale dynamically reconfigurable parallel processing system that can be partitioned to operate as many independent SIMD/MIMD machines of various sizes (see Fig. 1) [22]. A prototype has been designed [11] and is currently being constructed. The *System Control Unit* is a conventional machine, such as a PDP-11/70 for the full system, and is responsible for the overall coordination of the activities of the other components of PASM. The *Parallel Computation Unit* (PCU) contains $N = 2^n$ processors ($N = 1024$ for full system, 16 for prototype), N memory modules, and an interconnection network (see Fig. 2). The *PCU processors* are microprocessors that peform the acutal SIMD and MIMD computations. The *PCU memory modules* are used by the PCU processors for data storage in SIMD mode and both data and instruction storage in MIMD mode. A pair of memory units is used for each PCU memory module so that task execution can be overlapped with the loading and unloading of the PCU memory modules [30]. A PCU processor and its associated memory module form a *processing element (PE).* The PE's are physically addressed from 0 to $N - 1$. The *interconnection network* provides a means of communication among the PE's. PASM will use either a cube [1] type or augmented data manipulator [13] type of multistage network. These two types of networks are currently being compared. The *Memory Management System* controls the loading and unloading of the PCU memory modules from the multiple secondary storage devices of the *Memory Storage System.*

The *Micro Controllers (MC's)* are a set of microprocessors that act as the control units for the PE's in SIMD mode and orchestrate the activities of the PE's in MIMD mode (see Fig. 3). There are $Q = 2^q$ MC's ($Q = 16$ or 32 for full system, 4 for prototype), addressed from 0 to $Q - 1$, and each MC controls N/Q PE's. A reconfigurable bus between the MC processors and MC memories can be used to improve MC memory utilization [22]. A PASM *MC-group* is composed of an MC processor, its memory module, and the N/Q PE's that are controlled by the MC. The N/Q PE's connected to MC i are those whose addresses have the value i in their low-order q bit positions. A *virtual machine (VM)* refers to a hardware collection of MC-groups. A *subvirtual machine (sub-VM)* of a given VM is a VM whose MC-groups are all

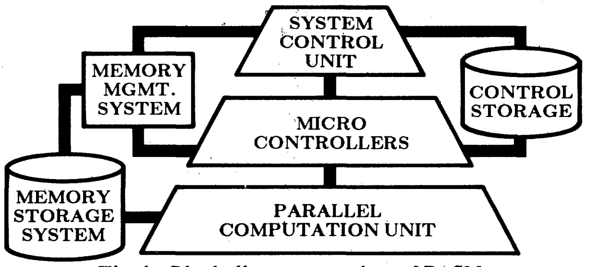


Fig. 1. Block diagram overview of PASM.

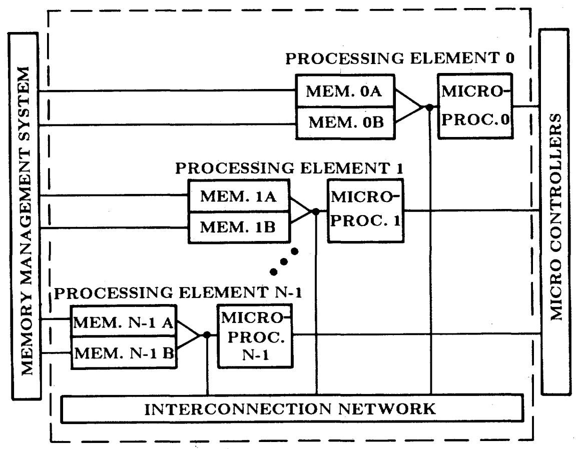


Fig. 2. PASM Parallel Computation Unit (PCU).

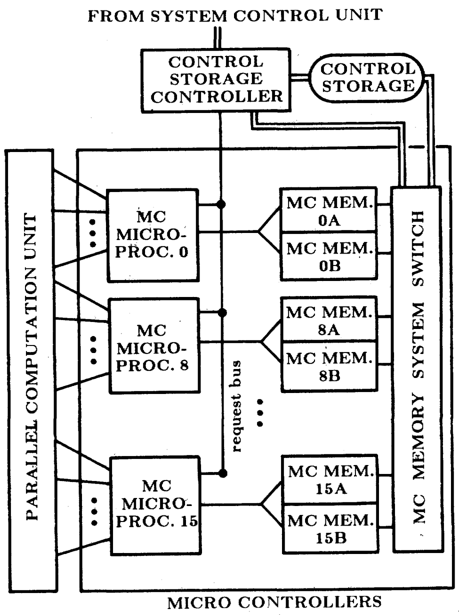


Fig. 3. Organization of the PASM Micro Controllers (MC's).

contained in the given VM. All of the MC-groups within a VM can be assigned to the same task, or different sub-VM's that compose the given VM can be assigned to different tasks. *Control Storage* contains the programs for the MC's and is controlled by the *Control Storage Controller.*

A VM of size RN/Q, where $R = 2^r$ and $0 \leqslant r \leqslant q$, is obtained by combining the efforts of R MC-group(s). According to the partitioning rule for PASM [22], the physical addresses of these MC's must have the same low-order $q - r$ bits so that all of the PE's in the partition have the same low-order $q - r$ physical address bits. For example, for $Q = 16$, an allowable partition of the MC-groups is: 1) MC-group 6 ($R = 1$); 2) MC-group 14 ($R = 1$); 3) MC-groups 2 and 10 ($R = 2$); 4) MC-groups 0, 4, 8, and 12 ($R = 4$); and 5) MC-groups 1, 3, 5, 7, 9, 11, 13, and 15 ($R = 8$). Q is therefore the maximum number of partitions allowable, and N/Q PE's is the size of the smallest partition. Futhermore, the number of MC-groups (and PE's) in a partition must be a power of 2.

The reason for using this particular partitioning rule is because it allows networks like the Extra Stage Cube [1] and the Augmented Data Manipulator [13], which are being considered for PASM, to be partitioned into independent subnetworks [21]. This rule is also valid for other cube type networks, such as the Omega [12], shuffle-exchange [16], and indirect binary n-cube [18] networks, as well as other data manipulator [4] type networks such as the Gamma [17] network [21].

The *designator* of a VM composed of an allowable partition is the smallest physical address of the MC's in the VM. The notation "i/j" refers to the VM of size j MC-groups with designator i. This designator corresponds to the low-order $q - r$ bits of the physical address of each MC in the VM. For the partitions in the above example, the designators are: 6, 14, 2, 0, and 1, respectively.

The approach of permanently assigning a fixed number of PCU processors to each MC has the advantages that the operating system need only schedule Q MC-groups, rather than N PCU processors; it allows the interconnection network to be partitioned into independent subnetworks; and it simplifies the MC/PE interaction, from both a hardware and software point of view, when a VM is being formed. In addition, this fixed assignment scheme is exploited in the design of the Memory Storage System in order to allow the effective use of parallel secondary storage devices [30].

III. Algorithm Descriptions

The *multiple-queue scheduling algorithms* make use of $q + 1$ first-in first-out (FIFO) task queues, $TQ_0, TQ_1, \cdots, TQ_q$, to group tasks which are to be scheduled for execution. In the case of PASM, the tasks are grouped by the number of MC-groups that they require. A task that requires 2^k MC-groups will be put into task queue TQ_k. The multiple-queue algorithms scan the queues beginning with either TQ_0 or TQ_q. In this paper, four multiple-queue scheduling algorithms are presented: the first-fit multiple-queue, the bit-reversed first-fit multiple-queue, the bottom-up best-fit multiple-queue, and the top-down best-fit multiple-queue.

When TQ_0 is examined first, the multiple-queue algorithm selects the first task in TQ_k, where k is the smallest integer such that TQ_k is not empty and a VM of size 2^k MC-groups is available for task execution. This step is repeated until all queues are empty or until there is not an available VM of size 2^k MC-groups where k is the smallest integer such that TQ_k is not empty. When TQ_q is examined first, the multiple-queue algorithm selects the first task in TQ_k, where k is the largest integer such that a VM of size 2^k MC-groups is available for execution. If TQ_k is empty, then the first task from TQ_{k-1} is selected. This process is continued until all available MC-groups have been assigned or until $k = 0$.

The multiple-queue algorithms are *nonpreemptive* scheduling policies, i.e., once a task is executing on a VM it cannot be interrupted. This is a result of the fact that the MC-groups are not multiprogrammed and that each MC processor can only invoke the scheduler when it completes a task. The multiple-queue algorithms are *centralized* scheduling algorithms [9] since the scheduler has complete accurate information regarding the states of all tasks in the system.

The four multiple-queue algorithms differ in the placement policy used to select the VM on which the task is to be placed. The *first-fit multiple-queue (FFMQ)* scheduling algorithm selects the free VM with the smallest designator. The *bit-reversed first-fit multiple-queue (RFMQ)* scheduling algorithm selects the free VM with the smallest bit-reversed designator. For example, consider a PASM with $Q = 16$ MC-groups. There are eight VM's of size two MC-groups, with designators 0, 1, 2, $\cdots$, 7. If VM's with designators 1, 2, and 4 are free, the FFMQ algorithm would select the VM with designator 1. Since the bit-reversed representations of 1, 2, and 4 are $(100)_2$, $(010)_2$, and $(001)_2$, respectively, the RFMQ algorithm would select the VM with designator 4 (i.e., bit-reversed designator $(001)_2$). The RFMQ algorithm is considered since the MC's in PASM are partitioned by low-order physical address bits, i.e., the bit-reversed values of the addresses of the MC's that are combined to form a given VM will be consecutive.

The *best-fit multiple-queue (BFMQ)* scheduling algorithms assign the task to the best VM where the best VM is determined using either the bottom-up or top-down heuristic. With the *bottom-up heuristic* the effect on VM's of the smallest size is considered first, and then to choose between equally good possibilities, the effect on larger VM's is considered. Let the *mate* of a given VM controlled by R MC's be the VM (controlled by R MC's) that is grouped with the given VM to form a VM controlled by $2R$ MC's. For example, for $Q = 16$ and $R = 4$, the mate of VM 1/4 (consisting of MC-groups 1, 5, 9, and 13) is VM 3/4 (consisting of MC-groups 3, 7, 11, and 15). That is, VM 1/4 and VM 3/4 can be combined to form VM 1/8. Since a VM must be paired with its mate to form a larger VM, a free VM with an active (currently being used) mate cannot be used as a part of a larger VM of any size. Hence, higher selection priority is given to free VM's with active mates.

After checking for active mates, if more than one VM has highest selection priority, then from the set of VM's that have highest priority, the VM that has the largest number of active secondary mates is selected. The *secondary mates* of a VM controlled by R MC's are the two VM's each controlled by R MCs that are grouped with the given VM (and its mate) to form a VM controlled by $4R$ MCs. This process is repeated until all levels of mates have been considered or until only one VM has highest selection priority. After all levels have been considered, if more than one VM has highest priority, then the VM with the smallest designator is selected. The bottom-up best-fit placement heuristic is similar to the buddy system which is used for allocating storage in blocks of size 2^k [25].

With the *top-down heuristic* the effect on the largest size of VM is considered first and is then reduced down to smaller sizes of VM's to resolve conflicts. This heuristic finds the VM that is located in the most active part of the system. Initially, the two VM's of size $Q/2$ MC-groups are considered. If only one has a free sub-VM of the desired size, then it is selected. If both VM's of size $Q/2$ MC-groups have a sub-VM of the desired size that is free, then the VM of size $Q/2$ MC-groups with the most active MC-groups is selected. If they have the same number of active MC-groups, then the VM of size $Q/2$ MC-groups with the smallest designator is selected.

Next the two sub-VMs of size $Q/4$ MC-groups of the selected VM (of size $Q/2$ MC-groups) are considered. The VM of size $Q/4$ MC-groups is selected by the same criterion as the VM of size $Q/2$ MC-groups was selected. This reduction process is continued until a VM of the desired size is selected.

In summary, the bottom-up heuristic attempts to optimize the current state of the system. For example, if the mate of a free VM is active, then that VM cannot be part of any larger VM, and hence is an optimal place to assign the task. The top-down heuristic attempts to optimize the future state of the system by grouping active VM's together. This difference is illustrated by the example in Fig. 4. Consider a PASM with 16 MC-groups ($Q = 16$). A task that requires two MC-groups ($R = 2$) is to be assigned. Currently, MC-groups 0, 1, 4, 8, 9, and 12 are active, i.e., VM's 0/2, 1/2, and 4/2 are active. With the bottom-up heuristic (left side of Fig. 4), since there is only one free VM of size two MC-groups with an active mate, it is selected. Hence, the VM 5/2 is selected. Using the top-down heuristic (right side of Fig. 4), both VM's of size 8 MC-groups ($Q/2$) have free VM's of size 2 MC-groups. Since VM 0/8 has four active MC-groups and VM 1/8 has only two, VM 0/8 is selected. Since only one of the sub-VMs (of size 4 MC-groups) of VM 0/8 has a free sub-VM (i.e., VM 2/4), it is selected. Since both sub-VMs of size 2 MC-groups of VM 2/4 are available, VM 2/2 is selected for task assignment since it has the lowest designator. This assignment caused the loss of a VM of size 4 MC-groups (VM 2/4) in addition to the VM of size 2 MC-groups, which is not the optimal selection for the current state of the system. However, the top-down heuristic has assigned the task to the more active half of the system, in an attempt to increase the probability that a VM of size 8 MC-groups (i.e., VM 1/8) will become free. Both the FFMQ and RFMQ algorithms would assign the task to VM 2/2.

In Section IV, the system performance resulting from the use of the BFMQ algorithms is examined. In addition, the FFMQ and RFMQ algorithms are also considered since they have less processing overhead (in the VM selection) than the BFMQ algorithms. Lastly, for comparative purposes the performance of the *first-fit single-queue (FFSQ)* scheduling algorithm is also considered. The FFSQ algorithm uses the first-fit placement policy of the FFMQ algorithm, but only uses a single FIFO task queue.

It is noted that the multiple-queue algorithms allow for overlapped unloading of the output data for the previously executed task (using the double-buffered memory modules) with the execution of the current task. As soon as the execution of a task is completed, the next task(s) can be assigned by the scheduler and loaded into memory (e.g., into the A memory units) by the Memory Storage System. After the system begins to execute the newly assigned task(s), the Memory Storage System can unload the output data (e.g., from the B memory units) for the previously executed task. Since it is not known in advance which task(s) will next be executed by a collection of MC-groups (because it is now known which currently executing task will finish first), the use of the double-buffered memory modules for preloading of programs and input data is not considered in this analysis (for a discussion of preloading see [30]).

IV. Performance Analysis

In this section the results of the simulation studies for the scheduling algorithms using a PASM with 16 MC-groups ($Q = 16$) are discussed. Details of the simulations are given in [27]. The performance of the system was determined using a discrete-event simulation technique [6] with following scheduling variations: A) FFSQ algorithm, B) FFMQ algorithm with TQ_0 first, C) FFMQ algorithm with TQ_4 first, D) RFMQ algorithm with TQ_4 first, E) bottom-up BFMQ algorithm with TQ_4 first, and F) top-down BFMQ algorithm with TQ_4 first.

The input parameters to the simulator consist of the mean task interarrival time, a discrete specification for the task size distribution, the number of faulty MC-groups, and the type of task execution time distribution: uniform or exponential. If a uniform distribution is requested, the range of execution times must be specified, and if an exponential distribution is requested, the mean task execution time must be specified. The distribution function for the task size is specified discretely by listing the probability for each possible task size. If there are faulty MC-groups, the physical addresses of the faulty MC-groups must be given.

Initial studies showed that the FFMQ algorithm with TQ_4 first performed significantly better than the FFMQ algorithm with TQ_0 first (quantitative details are presented below). This substantiated the intuitive notion that scanning TQ_0 first would yield worse performance since scheduling smaller tasks first tends to fragment the collection of MC-groups, preventing larger tasks from running and therefore underutilizing the processors. For this reason, the performance of the RFMQ and BFMQ algorithms with TQ_0 first was not considered.

Performance measures examined in this paper are processor utilization and average response time. The *processor utilization* is the fraction of time that the processors (MC's and PE's) are active during the simulation. The *response time* for a task is the delay between the time when a task arrives at the system and the time when that task completes execution on the system. The response time is being considered since a decrease in response time has the most direct effect on the user. It is noted that system throughput, which is examined in [27], is not considered in this paper since for purpose of this analysis the same conclusions can be drawn from the processor utilization.

"In computer systems, the arrival of individuals at a card reader, the failure of circuits in a central processor, and requests from terminals in a time-sharing system are processes that are essentially Poisson in nature" [6]. Since PASM serves requests from terminals (as does a time-sharing system), task arrivals are modeled with a Poisson process. The performance analysis has been divided into four independent experiments.

Experiment 1

In this experiment the task size distribution was uniform and the task execution time distribution was exponential with

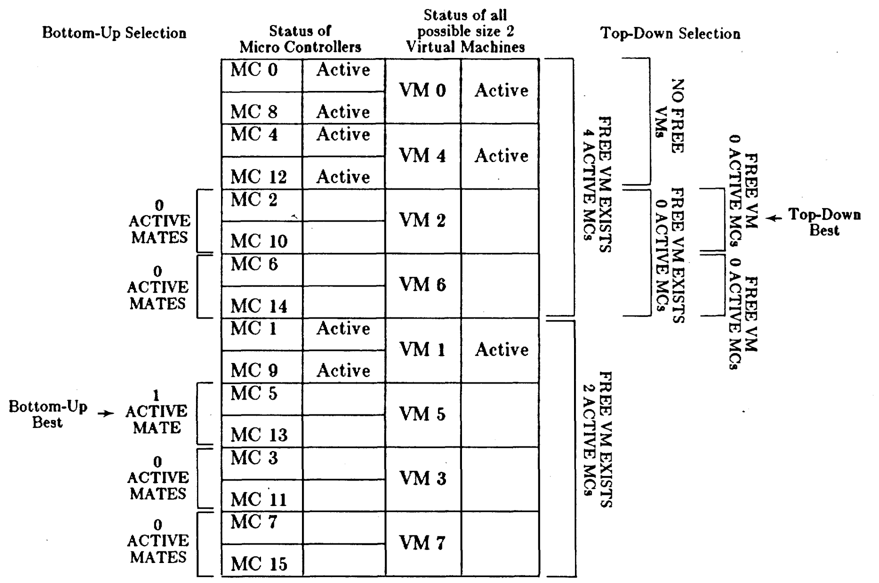


Fig. 4. Description of the selections made by the bottom-up and the top-down best-fit heuristics, with $Q = 16$.

mean of 22 s. The mean task interarrival time was 10 s. This combination of mean interarrival time and mean task execution time was selected since it yields a high level of processor utilization (resulting in a large number of scheduling decisions) and it does not cause the system to become saturated with any of the scheduling algorithms. The simulations were run for 2000 "PASM seconds," and 183 tasks were scheduled by each algorithm. The purpose of this experiment is to determine the distribution of "MC-group choices" made by the different placement policies. Table I indicates the distribution of "MC-group choices"made by the FFMQ, the RFMQ, both BFMQ, and the worst-fit multiple-queue (WFMQ) algorithms, all with TQ_4 first. (The WFMQ algorithm is the same as the bottom-up BFMQ algorithm except that higher selection priority is based on free mates instead of active mates.) A *best* choice occurs when there is no other choice that is better (using the bottom-up heuristic) and there is a choice that is worse. The bottom-up heuristic was arbitrarily chosen over the top-down just for the purpose of having a standard for comparison. A *worst* choice occurs when there is no other choice which is worse, but there is one which is better. When all choices are the same (i.e., no best or worst choice exists) or there is only one choice, the choice is classified as a *same* choice. When both best and worst choices exist, all other choices (which are neigher best or worst) are classified as *other* choices.

From the table it can be seen that in approximately 70 percent of the choices made by the schedulers the placement policy did not matter since all choices were classified as same choices. In 80 percent of the "nonsame" choices, the FFMQ algorithm made worst choices. Considering the "nonsame" choices made by the top-down BFMQ algorithm, 97 percent of the choices were classified as (bottom-up) best. As a result of this, it is expected that which best-fit heuristic is used will not make much difference in system performance.

The two best-fit heuristics usually select the same best VM since the VM with the most active mates tends to be in the

TABLE I
DISTRIBUTION OF "MC-GROUP CHOICES"

	FFMQ	RFMQ	BU-BFMQ	TD-BFMQ	WFMQ
Best	2.73%	18.03%	32.79%	31.15%	0.00%
Worst	23.50%	8.20%	0.00%	0.00%	32.79%
Same	70.50%	71.04%	67.21%	67.76%	67.21%
Other	3.28%	2.73%	0.00%	1.09%	0.00%

most active part of the system and a free VM in the most active part of the system tends to have active mates. When the RFMQ algorithm schedules two tasks of the same size, it will always assign the second task to the mate of the first (if it is free), yielding a bottom-up best choice. As a result, the RFMQ algorithm tends to schedule tasks together in the same part of the system. On the other hand, the FFMQ algorithm tends to schedule tasks away from each other, resulting in more worst choices. For example, in Fig. 4 the RFMQ algorithm would assign two tasks that each require two MC-groups to VM's 2/2 and 6/2 (bit-reversed $(010)_2$ and $(011)_2$), which are close together. The FFMQ algorithm would assign the tasks to VM's 2/2 and 3/2, which are far apart. Hence, the RFMQ algorithm makes more best choices than the FFMQ algorithm.

Experiment 2

In this experiment the task execution time distribution was uniform with minimum and maximum execution times of 1 and 15 s, respectively. The minimum execution time of 1 s is based on the time to do a simple image processing algorithm, such as smoothing [22], on a 300 X 300 pixel (picture element) subimage within each PCU processor. The mean task interarrival time was varied from 2 to 10.5 s. This range of the mean interarrival time was selected since it ranges from the level where the system is saturated with tasks to the level where all scheduling algorithms yield approximately the same performance. The task size distribution was uniform. All simulation runs were for 5000 "PASM seconds," and the number of task arrivals ranged from 468 to 2500. The processor utilization and average response time are given as functions of the mean task interarrival time in Figs. 5 and 6, respectively.

When the mean interarrival time is 9.5 s or more, each of the scheduling algorithms is able to schedule all of the arriving tasks. Since the same set of tasks is scheduled by each scheduling variation, the processor utilization is the same for all scheduling variations (see Fig. 5). As the mean interarrival time decreases below 6 s, the processor utilization for the FFSQ and FFMQ with TQ_0 first scheduling variations is consistently less than the other variations. This occurs since the multiple-queue algorithms with TQ_0 first select the worst task to be scheduled next while the mulitple-queue algorithms with TQ_4 first select the best task to be scheduled next. The *best task* to be scheduled next is the task that requires the most MC-groups since it results in the highest processor utilization. The available task that requires the fewest number of MC-groups is considered the *worst task* since it may be assigned to part of a VM which could have been used by a larger task (which could have yielded higher processor utilization) and causes fragmentation of the available MC-groups in the system. The processor utilization resulting from the use of the FFSQ algorithm is between the processor utilization resulting from the use of the FFMQ algorithm with TQ_0 first and the FFMQ algorithm with TQ_4 first since it selects tasks in the order which they arrive, sometimes selecting the best task and sometimes selecting the worst task to be scheduled.

In this experiment for almost all mean interarrival times, each of the multiple-queue algorithms with TQ_4 first is able to schedule and execute all of the arriving tasks. For a given mean interarrival time, even though the algorithms do not schedule the tasks in the same order, over the entire simulation run the same set of tasks are scheduled and executed. As a result, the processor utilization (i.e., average utilization over each simulation run) is the same for each of the multiple-queue algorithms with TQ_4 first.

As the mean interarrival time is decreased, the average response time for the FFMQ algorithm with TQ_0 first increases much more rapidly than for the FFMQ algorithm with TQ_4 first. Since the FFMQ algorithm with TQ_0 first selects the worst task to schedule next, the processor utilization is lower and, overall, fewer tasks are executed. Hence, arriving tasks must wait longer to be scheduled, resulting in an increase in the average response time.

As the system approaches the saturation point (point at which the system is not able to execute all arriving tasks), the FFSQ algorithm yields shorter average response times than the FFMQ algorithm with TQ_0 first. This results from the fact that the FFSQ algorithm may or may not select the worst task to be scheduled from its single queue, while the FFMQ algorithm with TQ_0 first will always select the worst task.

Although it cannot be seen from the graph, the average response time for the FFMQ algorithm with TQ_4 first is consistently one to two percent longer than the other three multiple-queue algorithms with TQ_4 first. The average response time for the RFMQ algorithm is usually the same or within 0.2 percent (either shorter or longer) of both best-fit scheduling algorithms for all arrival rates. Overall, the

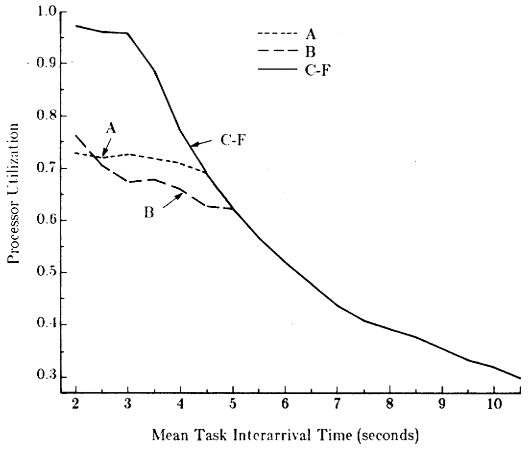


Fig. 5. Processor utilization as a function of the mean task interarrival time for the (A) FFSQ algorithm, (B) FFMQ algorithm with TQ_0 first, (C) FFMQ algorithm with TQ_4 first, (D) RFMQ algorithm with TQ_4 first, (E) bottom-up BFMQ algorithm with TQ_4 first, and (F) top-down BFMQ algorithm with TQ_4 first.

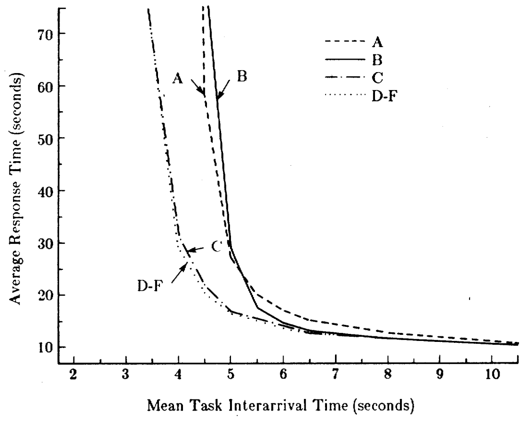


Fig. 6. Average response time (in seconds) as a function of the mean task interarrival time for scheduling variations (A)-(F) as defined in Fig. 5.

bottom-up best-fit policy yields slightly shorter average response times than the top-down. This implies that the bottom-up best-fit heuristic gives a better description of what is "best." The performance of the FFMQ algorithm with TQ_4 differs from the other multiple-queue algorithms since it tends to make a large percentage of worst "MC-group choices," as shown in Experiment 1. On the other hand, the performance of the RFMQ algorithm is closer to that of the BFMQ algorithm since it tends to make the best MC-group choices.

In this experiment the maximum task execution time was 15 s. If the scheduler makes a bad decision in determining to which VM a task should be assigned or about which task should be assigned next, the effect of the decision will last at most 15 s. In Experiment 3, the maximum execution time is varied to determine if the effect of the bad decisions will be increased.

Experiment 3

In this experiment the distribution for the task execution time was uniform with a minimum of 1 s. The maximum task execution time was varied from 12.5 to 125 s. The mean

TABLE II
AVERAGE RESPONSE TIME (IN SECONDS) AS A FUNCTION OF THE MAXIMUM TASK EXECUTION TIME

Maximum Execution Time	FFSQ	FFMQ TQ_0 1st	FFMQ TQ_4 1st	RFMQ	BFMQ B-Up	BFMQ T-Down
12.5	7.108	7.078	7.072	7.066	7.066	7.066
25.0	14.68	14.47	14.44	14.36	14.36	14.36
37.5	24.14	23.08	23.06	22.89	22.89	22.89
50.0	36.59	33.98	33.64	33.37	33.37	33.37
62.5	53.09	46.91	46.55	46.17	46.10	46.14
75.0	76.55	65.28	62.56	61.66	61.66	61.91
87.5	119.1	102.3	87.62	84.31	84.27	84.30
100.0	263.1	310.3	129.0	117.9	117.3	115.9
112.5	1040.	622.7	184.6	178.4	178.3	178.4
125.0	1943.	761.6	363.9	329.2	328.6	328.6

TABLE III
AVERAGE RESPONSE TIME (IN SECONDS) AS A FUNCTION OF THE MEAN TASK EXECUTION TIME

Mean Execution Time	FFSQ	FFMQ TQ_0 1st	FFMQ TQ_4 1st	RFMQ	BFMQ B-Up	BFMQ T-Down
2.5	2.764	2.751	2.754	2.746	2.753	2.746
5.0	6.501	6.191	6.227	6.203	6.226	6.200
7.5	12.23	10.73	10.66	10.56	10.63	10.55
10.0	21.76	16.88	17.01	16.78	17.08	16.78
12.5	42.63	27.96	26.40	25.85	26.41	25.65
15.0	130.3	68.14	41.31	41.05	41.64	41.12
17.5	785.8	298.0	69.52	68.94	68.85	68.67
20.0	1750.	513.8	132.2	131.5	130.3	130.6

TABLE IV
PROCESSOR UTILIZATION AS A FUNCTION OF THE MEAN TASK EXECUTION TIME

Mean Execution Time	FFSQ	FFMQ TQ_0 1st	FFMQ TQ_4 1st	RFMQ	BFMQ B-Up	BFMQ T-Down
2.5	0.101	0.101	0.101	0.101		
5.0	0.203	0.203	0.203	0.203		
7.5	0.304	0.304	0.304	0.304		
10.0	0.405	0.405	0.405	0.405	same as RFMQ	same as RFMQ
12.5	0.505	0.505	0.506	0.506		
15.0	0.604	0.603	0.606	0.606		
17.5	0.648	0.646	0.707	0.707		
20.0	0.660	0.660	0.807	0.808		

TABLE V
AVERAGE RESPONSE TIME (IN SECONDS) AS A FUNCTION OF TASK SIZE

Task Size	FFSQ	FFMQ TQ_0 1st	FFMQ TQ_4 1st	RFMQ	BFMQ B-Up	BFMQ T-Down
1	131.0	19.82	41.21	41.36	41.36	41.24
2	126.7	21.99	36.63	36.79	36.88	36.67
4	127.8	19.68	32.81	32.73	32.55	32.50
8	125.3	29.93	37.25	36.18	37.80	36.47
16	141.3	252.09	58.90	58.44	59.82	58.97

task interarrival time was 30 s, the task size distribution was uniform, and all simulations were run for 30 000 "PASM seconds," resulting in approximately 1000 task arrivals. The average response time is given as a function of the maximum execution time for the six scheduling variations in Table II. For maximum execution times of less than 50 s, the average response time does not vary significantly with changes in the scheduling variation. When the maximum execution time is 125 s, the use of the two BFMQ algorithms results in the same average response time. The RFMQ algorithm results in slightly longer (0.2 percent) average response times than the BFMQ algorithms, while the FFMQ algorithm with TQ_4 first results in significantly longer (10 percent) response times than the RFMQ and BFMQ algorithms.

This difference in performance between the FFMQ algorithm with TQ_4 first and the RFMQ algorithm results from the fact that the RFMQ algorithm makes more best "MC-group choices" than the FFMQ algorithm (as shown in Experiment 1). When the maximum task execution time is 62.5 s or less, the effect of bad "MC-group choices" made by the first-fit placement policy of the FFMQ algorithm with TQ_4 first are not as significant since the processor utilization is low (i.e., there are more processors available) and since the effect of bad decisions does not last as long.

In summary, the longer the task execution time, the greater the difference in performance of the four multiple-queue algorithms with TQ_4 first. This occurs since the effect of the "MC-group choices" made by the scheduling algorithms becomes more significant with longer task execution times.

Experiment 4

In this experiment the task size distribution was again uniform and the task execution time distribution was exponential. The mean execution time was varied from 2.5 to 20 s. The mean task interarrival time was 10 s, and all simulations were run for 20 000 "PASM seconds," resulting in approximately 2000 task arrivals. The average reponse time and processor utilization are given as functions of the mean execution time for the six scheduling variations in Tables III and IV, respectively. The average response time for small mean execution times is the same for all scheduling variations. As the mean execution time is increased, the average response time increases rapidly for the single queue scheduling algorithm. For the same reasons as given in Experiment 2, all four multiple-queue algorithms with TQ_4 first yield similar performance, indicated by similar average response times (see Table III) and the same processor utilization (see Table IV).

Table V gives the average response time for the six scheduling variations as a function of the task size for the case when the mean execution time is 15 s. As would be expected, since the FFSQ algorithm selects tasks for execution in the order that they arrived, tasks of all sizes (number of MC-groups required) receive the same treatment. This is illustrated in Table V in that the average response time varies less (on a percentage basis) between the different task sizes for the FFSQ algorithm than it does for the multiple-queue algorithms. The 16 MC-group tasks have the greatest average response time since they must always wait at the head of the queue for all previous tasks to complete execution before they can be scheduled (i.e., all processors must be free).

As discussed earlier, the FFMQ algorithm with TQ_0 first tends to favor tasks that require one, two, or four MC-groups. This is also illustrated by Table V. Tasks that require 16 MC-groups are only executed when there are no other tasks to be executed and when all MC-groups are free. This results in the average response time for 16 MC-group tasks being an order of magnitude longer than the other size tasks.

For the four multiple-queue algorithms with TQ_4 first, the average response time for tasks that require 16 MC-groups is almost 60 s, while it is less than 42 s for all other tasks since all MC-groups must be free for a 16 MC-group task to be scheduled. Two competing properties effect the average response time. One is, the fewer MC-groups a task requires, the shorter its response time since it must wait less time for a smaller size VM to become available. The other is, with the multiple-queue algorithms with TQ_4 first, the more MC-groups a task requires the shorter its response time since the system attempts to schedule larger tasks first. As a result, tasks that require four MC-groups have the shortest response times since they take advantage of both of these effects. When the FFMQ algorithm with TQ_0 first is used, tasks that require 16 MC-groups cannot take advantage of either of these properties; this explains the extremely long average response times for them.

Summary

The single FIFO queue used by the FFSQ algorithm has a tendency to become blocked by tasks that require a large number of processors. The improved performance of the multiple-queue algorithms results from the elimination of the blockage problem. When the system is heavily loaded, the FFMQ algorithm with TQ_0 first tends to schedule tasks that require one, two, or four MC-groups, while making tasks that require 8 and 16 MC-groups wait; in addition, scheduling the smaller tasks first decreases processor utilization since the free MC-groups tend to become fragmented and cannot be grouped to form larger VM's. The system performance resulting from the use of all four multiple-queue algorithms with TQ_4 first is similar. This occurs since they all select the "best" task (i.e., the task that requires the largest number of MC-groups) when selecting the next task to be scheduled. Selecting the "best" task to schedule tends to limit the impact of the placement policy.

Due to the extra overhead in the "selection processes" used by the BFMQ algorithms for selecting the best VM, the BFMQ algorithms take more time to perform the scheduling operations. It has been determined that the use of the RFMQ algorithm in certain cases results in slightly better system performance than the FFMQ with TQ_4 first. Since the system performances resulting from the use of the RFMQ and both BFMQ algorithms are approximately the same, and the RFMQ algorithm requires less computation time than the BFMQ algorithms, the RFMQ algorithm appears to be the algorithm of choice for PASM when there are no faulty processors. In the next section the relative performance of the algorithms when there are faulty MC-groups is considered.

V. System Faults

In this section the performance of the multiple-queue algorithms when they are faulty MC-groups is considered. In general, an MC-group is faulty if the MC or one of its PE's is faulty. Depending on which MC-group(s) is faulty, the system performance varies. However, the faulty MC-group(s) can be selected by switching boards. From the simulation studies in this section, the best MC-group(s) to be faulty can be selected.

If a VM is assigned to a real-time task with a long execution time, the MC-groups that compose the VM will not be available to execute other tasks for an extended period of time. In the same way that the best MC-groups to be faulty can be selected, the best MC-groups (or VM) to which a real-time task is to be assigned can be selected. If a task is known to have a long execution, the task scheduler will not schedule the task until the optimal VM is available, increasing overall system performance.

Since PASM is not able to execute tasks that require the entire system when there is a faulty MC-group, for simulations in this section tasks require from one to eight MC-groups (recall, $Q = 16$ is assumed). As in Section IV, the effect on the processor utilization and the average response time is considered. The performance of the system is determined using the multiple-queue algorithms with TQ_4 first [i.e., scheduling variations C) through F)]. Two experiments are considered.

Experiment 5

In each simulation run there were 2^i faulty MC-groups, $0 \leqslant i < q$, and the faulty MC-groups compose VM $j/2^i$, $0 \leqslant j < 2^{q-i}$. All possible values of i and j were considered. For case $i = 0$ and $0 \leqslant j < 16$, the faulty MC-group forms VM $j/1$; for case $i = 1$ and $0 \leqslant j < 8$, the two faulty MC-groups form VM $j/2$; and so forth for larger values of i. The task execution time distribution was uniform with minimum and maximum of 1 and 15 s, respectively (as in Experiment 2). The mean task interarrival time was 3.5 s so that the maximum processor utilization would be approximately 50 percent, the maximum allowable processor utilization when there is a faulty VM of size 8 MC-groups. The task size distribution was uniform with minimum and maximum of one and eight MC-groups, respectively. All simulation runs were for 3000 "PASM seconds," resulting in approximately 850 task arrivals. The purposes of this experiment are 1) to determine which scheduling variation is the most fault-tolerant and 2) to determine if the system performance depends on which VM of a given size is faulty.

The average response time for the four multiple-queue algorithms with TQ_4 first are given as functions of the size and designator of the faulty VM in Table VI. For the cases when there is a faulty VM of size one MC-group (i.e., one faulty MC-group in the system), the BFMQ algorithms yield a smaller variance in average response time and have the lowest average response time since they are better able to schedule tasks around the faulty MC-groups. Consider the case when all MC-groups are free, except for the faulty MC-group in VM 1 of size 8 MC's (i.e., one of the MC-groups: 1, 3, 5, $\cdots$, 15 is faulty). If the FFMQ algorithm is used to assign the next task, it will be assigned to VM 0/8 or to a sub-VM of VM 0/8. As a result there are no free VM's of size 8 MC-groups. On the other hand, if the fault had been in VM 0/8, only a task that requires eight MC-groups would be assigned to VM 1/8. Hence, the FFMQ algorithm naturally attempts to pack tasks that require from one to eight MC-groups into VM 0/8. As a result, if the faulty MC-group is in VM 0/8, tasks that require one, two, or four MC-groups will be packed around the faulty MC-groups. The FFMQ algorithm therefore performs worse when the faulty MC-group is a member of VM 1/8 (i.e., an odd numbered MC-group). The RFMQ algorithm has a large variation in average response time for similar reasons as the FFMQ algorithm.

When there is a VM of size 2 MC-groups which is faulty, the average response time for the BFMQ algorithms vary from 12.3 to 12.7 s. The upper bound on the average response time has only increased by 0.2 s while the lower bound has increased by almost 1 s from the one faulty MC-group case. The main cause of the increase is from tasks that require one MC-group. In the one faulty MC-group case, there was an MC-group (the mate of the faulty MC-group) that could only execute tasks that re-

TABLE VI
AVERAGE RESPONSE TIME (IN SECONDS) AS A FUNCTION OF THE FAULTY VM

Faulty VM	FFMQ TQ_4 1st	RFMQ	BFMQ B-Up	BFMQ T-Down
-	10.10	9.48	9.53	9.52
0/1	13.32	11.57	12.40	11.88
1/1	13.90	13.25	11.54	11.94
2/1	13.11	12.43	11.71	11.87
3/1	13.69	14.09	12.50	11.93
4/1	12.24	11.70	12.31	11.88
5/1	13.69	14.14	11.54	11.95
6/1	12.99	12.48	11.61	11.87
7/1	13.47	14.16	12.40	11.93
8/1	12.79	11.57	12.40	11.88
9/1	13.67	13.25	11.54	11.94
10/1	12.88	12.43	11.69	11.87
11/1	13.60	14.09	12.48	11.93
12/1	12.12	11.70	12.31	11.88
13/1	13.42	14.14	11.54	11.95
14/1	12.88	12.48	11.61	11.87
15/1	13.32	14.16	12.40	11.93
0/2	13.68	12.21	12.32	12.46
1/2	14.24	14.52	12.46	12.58
2/2	13.40	12.83	12.55	12.59
3/2	13.95	14.41	12.57	12.69
4/2	13.24	12.21	12.28	12.46
5/2	14.00	14.52	12.31	12.58
6/2	13.22	12.83	12.26	12.59
7/2	13.68	14.41	12.32	12.69
0/4	17.61	16.26	16.78	16.28
1/4	17.89	18.02	17.60	17.17
2/4	18.16	16.26	16.76	16.28
3/4	17.61	18.02	16.78	17.17
0/8	177.51	177.53	176.89	176.89
1/8	177.51	177.53	176.89	176.89

quire one MC-group. This results in a decrease in average response time for tasks that require one MC-group, which in turn affects the overall average response time. On the average, the BFMQ algorithms give the best overall performance when there are two faulty MC-groups. The FFMQ and RFMQ algorithms vary in average response time for similar reasons as they did in the one faulty MC-group cases.

For the two faulty MC-group case, the use of the bottom-up BFMQ algorithm tends to result in shorter average response times than the top-down BFMQ algorithm. The bottom-up heuristic performs better since it tends to assign task that require one and two MC-groups to the VM that has faulty secondary mates or a faulty mate, respectively, leaving the other MC-groups free for larger tasks. The top-down heuristic performs worse since it tries to assign the task to the most active part of the system with the hope that the task(s) executing in the least active part of the system will complete, leaving a free VM of size 8 MC-groups. However, if the faulty VM is in the currently least active part of the system, this will never be true.

When there is a faulty VM of size 4 MC-groups, the top-down BFMQ algorithm yields shorter response times than the bottom-up BFMQ algorithm in three out of four cases. This change from the one and two faulty MC-group cases occurs since a faulty VM of size 4 MC-groups will tend to always be in the most active half of the system. As a result, task that require one, two, and four MC-groups will usually be assigned to the mate of the faulty VM by the top-down BFMQ algorithm. On the other hand, the bottom-up heuristic performs worse since tasks that require one and two MC-groups are less likely to be assigned to the half of the system with the faulty VM (of size 4 MC-groups). As with the previous case there is variation in the average response time for the FFMQ and RFMQ algorithms.

Lastly, when there is a faulty VM of size 8 MC-groups, the performance is the same no matter which of the two VM's of size 8 MC-groups is faulty since this situation is equivalent to having a system with only eight MC-groups (i.e., $Q = 8$), no matter which VM of size 8 MC-groups is faulty. Overall, in the one, two, and four faulty MC-group cases, the BFMQ algorithms have the lowest average response times.

In each of the one, two, and four faulty MC-group cases listed in Table VI, the processor utilization is the same for both BFMQ algorithms (0.52518) while it varies slightly for the FFMQ and the RFMQ algorithms (ranging from 0.52486 to 0.52518). In each of the eight faulty MC-group cases, the processor utilization decreases from the one, two, and four faulty MC-group cases since the maximum possible processor utilization is now 0.5 (50 percent).

From this experiment it appears that the BFMQ algorithms, on the average, yield the shortest response times, and have the most consistent performance; i.e., if a VM of size j is faulty, system performance will not depend greatly on which VM of size j is faulty (see Table VI).

Experiment 6

This experiment is a study of the cases where there are two faulty MC-groups. As in Experiment 5, the task execution time distribution was uniform with minimum and maximum of 1 and 15 s, respectively; the task size distribution was uniform with minimum and maximum of one and eight MC-groups, respectively (recall that no tasks that require sixteen MC-groups can be executed if there is a faulty MC-group in the system); the mean task interarrival time is 3.5 s; and all simulations were run for 3000 "PASM seconds," resulting in approximately 850 task arrivals. All two faulty MC-group cases that are isolated to the same VM of size 8 MC-groups and include MC-group 0 are considered (i.e., MC-group 0 and another even numbered MC-group). The purpose of this experiment is to show how the different two-fault cases affect the average response time for tasks of each size.

The average response time for the four multiple-queue algorithms with TQ_4 first is given as functions of the physical addresses of the faulty MC-groups in Table VII. (Note: the entries in the first and second lines of Table VII are the same as and were generated by the same simulation runs as the entries in the first and second lines of Table VI.) Each pair of faulty MC-groups can be classified in one of the following cases. In case I both faults are in the same VM of size 2 MC-groups; in case II both faults are in the same VM of size 4 MC-groups but not in the same VM of size 2 MC-groups; and in case III both faults are in the same VM of size 8 MC-groups but not in the same VM of size 2 or 4 MC-groups. The average response time for a task that requires 2^i MC-groups is given as a function of the number of MC-groups for one of each of these fault cases in Table VIII.

When there is one faulty MC-group, there is a decrease in the average response time for tasks that require one MC-group. The cause for the decrease is that the mate of a faulty MC-group is only able to execute tasks that require one MC-group. There is an increase in the average response time for tasks that require two, four, and eight MC-groups since the fault prevents the formulation and use of certain VM's. For example, there is an increase in the average response time for tasks that re-

TABLE VII
AVERAGE RESPONSE TIME (IN SECONDS) AS A FUNCTION OF THE PHYSICAL ADDRESSES OF THE FAULTY MC-GROUPS (MCG'S)

Faulty MCGs	Fault Case*	FFMQ TQ_4 1st	RFMQ	BFMQ B-Up	BFMQ T-Down
none	-	10.10	9.48	9.53	9.52
0	-	13.32	11.57	12.40	11.88
0,2	III	46.50	31.18	31.12	43.63
0,4	II	14.90	14.69	14.85	14.90
0,6	III	45.83	31.27	31.12	43.63
0,8	I	13.68	12.21	12.32	12.46
0,10	III	46.10	31.18	31.12	43.63
0,12	II	14.78	14.69	14.85	14.90
0,14	III	45.82	31.27	31.12	43.63

*In case I both faults are in the same VM of size 2 MC-groups; in case II both faults are in the same VM of size 4 MC-groups but not in the same VM of size 2 MC-groups; and in case III both faults are in the same VM of size 8 MC-groups but not in the same VM of size 2 or 4 MC-groups.

TABLE VIII
AVERAGE RESPONSE TIME (IN SECONDS) AS A FUNCTION OF TASK SIZE FOR THE THREE TWO-FAULT CASES

Fault Case*	Task Size	Faulty MCGs	FFMQ TQ_4 1st	RFMQ	BFMQ B-Up	BFMQ T-Down
no	1	-	9.402	8.805	8.827	8.800
no	2	-	8.707	8.518	8.541	8.541
no	4	-	9.222	8.785	8.797	8.705
no	8	-	12.78	11.58	11.73	11.80
one	1	0	8.301	8.510	8.519	8.565
one	2	0	8.716	8.790	8.804	8.777
one	4	0	11.29	9.676	9.724	9.991
one	8	0	23.81	18.54	21.57	19.38
I	1	0,8	9.016	9.566	9.555	9.528
I	2	0,8	9.108	9.087	9.068	9.064
I	4	0,8	11.32	10.05	10.13	9.991
I	8	0,8	24.14	19.39	19.75	20.44
II	1	0,4	8.290	8.289	8.289	8.289
II	2	0,4	12.29	11.76	11.90	11.78
II	4	0,4	12.64	11.07	10.97	11.07
II	8	0,4	25.14	26.35	26.87	27.09
III	1	0,2	8.113	8.118	8.118	8.118
III	2	0,2	8.158	8.087	8.062	7.850
III	4	0,2	101.9	64.34	64.33	94.47
III	8	0,2	63.29	41.46	41.27	59.88

*See Table VII footnote.

quire eight MC-groups that results from the fact that the system can only form one VM of size 8 MC-groups instead of the two VM's that it can form when there are no faulty MC-groups.

In case I only one VM of size 2 MC-groups is faulty, so compared to the no fault case the only tasks that are significantly affected are tasks that require eight MC-groups. For case II, two VMs of size two MC-groups contain faulty MC-groups, so there is an increase in the average response time for tasks that require two MC-groups over case I. There is a decrease for tasks that require one MC-group since the mates of the faulty MC-groups can only execute tasks that require one MC-group. For case III there is an increase in the average response time for tasks that require four MC-groups since the system is only able to create two VMs of size 4 MC-groups instead of the four VM's that it can form when there are no faulty MC-groups and the three VMs that it can form in cases I and II. Furthermore, since the two nonfaulty VM's of size 4 MC-groups form the only VM of size 8 MC-groups, a task that requires four MC-groups cannot be executed while a task that requires eight MC-groups is being executed. This interference also results in an increase in the average response time for tasks that require eight MC-groups. Hence, the average response time where there are two faulty MC-groups is affected by which two MC-groups are faulty. In case III (when the faulty MC-groups are separated), the bottom-up heuristic does better than the top-down since it assigns tasks that require one and two MC-groups close to the faulty MC-groups while the top-down is less likely to since the faulty MC-groups may not be in the most active part of the system. This leaves those MC-groups that do not have faulty mates free to execute larger tasks. In general, the bottom-up BFMQ algorithm tends to yield the best performance (see Table VI).

Summary

When there is a faulty MC-group, the system is still able to execute tasks that require from one to four MC-groups with little increase in response time (see Table VIII). If there is more than one faulty MC-group, provided the faulty MC-groups are isolated to one VM of size 4 (cases I and II), the system can still perform without excessive delays for the user (see Table VII). If the faulty MC-groups are not isolated to the same VM of size 4, it may be possible to switch the boards containing the faulty MC-groups so that the faults can be isolated to the same VM (discussed below). The performance of the two BFMQ algorithms is comparable in all cases, except for case III where the bottom-up BFMQ algorithm yields significantly better performance. The average performance of the RFMQ algorithm is approximately the same as the bottom-up BFMQ algorithm; however, there is a greater variance in the average response time for the RFMQ algorithm (see Table VI). Hence, the advantage of the bottom-up BFMQ algorithm is that its performance is not a function of which MC-group is faulty.

In the previous section it was determined that the RFMQ algorithm was the algorithm of choice for a PASM with no faulty MC-groups. When there are faulty MC-groups, the performance of the RFMQ algorithm depends on which MC-groups are faulty. In certain cases the RFMQ algorithm performs as well or slightly better than the bottom-up BFMQ algorithm (e.g., MC-group 0 is faulty, see Table VI) while in other cases it performs worse (e.g., MC-group 1 is faulty, see Table VI). Therefore, in cases where the unusable MC-groups can be selected (e.g., MC-group 0 instead of MC-group 1), the RFMQ algorithm can perform as well or better than the BFMQ algorithms.

There are two situations when the unusable MC-groups can be selected. The first occurs when there is faulty hardware. Consider the case when there is a faulty PE board in MC-group 1. If a replacement board is not available, it may be possible to switch the faulty PE board with one of the PE boards from MC-group 0, resulting in improved system performance (using RFMQ). The second situation where unusable MC-groups can be selected is when a task is known *a priori* to have a very long execution time. The scheduler could then use this knowledge to assign the task to an optimal VM. If the unusable MC-groups cannot be selected, the bottom-up BFMQ algorithm yields the best performance and is the algorithm of choice for PASM.

VI. ALGORITHM MODIFICATION

The multiple-queue algorithms attempt to assign tasks so that the processor utilization is maximized while trying to minimize the average user response time. Unlike the first-fit shelf scheduling algorithm [26], the multiple-queue scheduling algorithms do not require the user to specify the maximum allowable execution time for the task. The user only needs to specify the number of MC-groups required to execute the task. This is an advantage in the PASM environment since many algorithms will be experimental and a good approximation to the execution time will not be known. In addition, the multiple-queue algorithms are nonpreemptive, i.e., once a task begins to execute on a VM its execution will not be interrupted until it is completed.

Giving tasks exclusive control of VM's for "unlimited" periods of time makes it possible for tasks that require from 1 to $Q/2$ MC-groups (N/Q to N/2 PE's) with long execution times to significantly increase the response time for tasks that require the entire system (N PE's). Consider the following example where the mean task execution time is 25 s. A task requiring one MC-group that has an execution time of 500 s is being executed by the system. During the 500 s period that the one MC-group task is being executed, it is not possible for the system to execute tasks that require the entire system. (However, it is possible for it to execute tasks of all other sizes.) Since 500 s is significantly greater than the mean execution time of 25 s, the response time for tasks that require the entire system may be increased during this period.

This problem can be solved by putting a limit on the amount of time that a task can wait to be scheduled. If the time limit is exceeded, the system stops executing the current task(s) and begins to execute the waiting task(s) that has been loaded into the alternate memory units. This type of context switch on PASM does not require any extra transfers between primary memory and secondary storage since both the executing task(s) and the waiting task(s) can be loaded into the double-buffered memory modules and the system can dynamically switch between the memory units. The limit on the amount of time a task can wait to be scheduled is a function of processor utilization. Since longer response times are acceptable when processor utilization is high, the greater the processor utilization, the greater the limit.

A second problem can occur when the system is heavily loaded. Since the multiple-queue algorithms with TQ_q first select tasks from TQ_0 last, tasks that require one MC-group can experience excessive delays when the system is heavily loaded. This problem can also be eliminated by limiting the time a task can wait in a queue. Hence, the maximum response (or wait) time for any task can be limited. It is noted that with the addition of this feature the multiple-queue algorithms will no longer be nonpreemptive. These modifications, which provide a limited form of multiprogramming, are currently being investigated.

VII. Relation to the General Model

A general model for a partitionable parallel processing system is given in Fig. 7. This model applies to both multiple-SIMD and partitionable SIMD/MIMD systems. The model consists of *Q control units*, *N* PE's, a secondary storage system for the control units, a secondary storage system for the PE's, a switch that is used to connect a control unit to a group of PE's, and a partitionable interconnection network for communication among the PE's. Each control unit and PE contains a

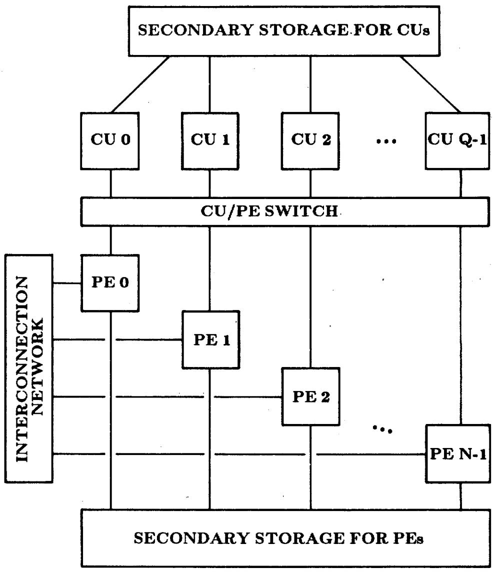


Fig. 7. A general model of a partitionable parallel processing system.

processor and a memory module. Either a separate system control unit (not shown in Fig. 7), e.g., PASM, or a dedicated control unit, e.g., MAP [14], [15], is responsible for overall system coordination. In the case of PASM, the switch is fixed in that a given PE is always connected to the same control unit (MC), and large machines are created by combining the efforts of control units (MC's). In [8], it is assumed that the switch provides all possible interconnections between control units and PE's, e.g., the crossbar type of switch used in MAP. In this case, all of the control units are not always used.

The multiple-queue algorithms can be adapted to the general model provided that the interconnection network can be partitioned into independent subnetworks. In addition, in order to use the best-fit placement policies, system partitioning must be restricted in the following way. Given that PE i is in a VM of size K, where $1 \leq K \leq N$, there is only one way to select the other PE's that form that VM, i.e., given the size of a VM and the address of any PE in that VM uniquely determines the set of PE's that forms the VM. This restriction results from the fact that the best-fit policies are based on the concept of combining nonoverlapping VM's together to form larger VM's. The following cube and data manipulator type networks can be partitioned so that they meet these restrictions: Augmented Data Manipulator [13], Extra Stage Cube [1], Gamma [17], Omega [14], indirect binary n-cube [18], and the shuffle-exchange [16] (see [21]). Systems with a binary-tree type network (topology) can also be partitioned so that these restrictions can be met (e.g., DADO [23]). If the tasks, which are to be scheduled, are to be grouped by task size, a task queue must be provided for each possible task size. Since all of the above networks are partitioned by powers of 2, this limits the number of task sizes to the powers of 2. It is noted that a cube type network can be partitioned in more than one way. The multiple-queue algorithms can be used provided that the cube network is always partitioned in a consistent way (e.g., always by the high-order bits). An alternative structure is to position the interconnection network between the processors and the memories instead of using it to connect the

PE's. The multiple-queue algorithm can also be adapted for use with this type of arrangement.

VIII. Conclusion

It has been shown that the multiple-queue algorithms yield significantly better performance than the use of a single first-in first-out queue. Since all multiple-queue algorithms (with TQ_q first) are able to select the best task (the task that yields the highest processor utilization) that is next to be scheduled for execution (through the use of the multiple queues), the best-fit placement policies of the BFMQ algorithms do not yield a significant improvement in performance over the first-fit and bit-reversed first-fit policies of the FFMQ and RFMQ algorithms, respectively. Since the partitioning rules for PASM are based on the low-order bits, the RFMQ algorithm performs better than the FFMQ algorithm. Hence, when there are no faulty MC-groups, the RFMQ algorithm is the best algorithm to use for PASM. However, when there are faulty MC-groups, the bottom-up BFMQ algorithm is the best algorithm to use for PASM. Therefore, when there are no faulty MC-groups detected by the system, the scheduler will use the bit-reversed first-fit placement policy, and when there are faulty MC-groups detected, scheduler will use the bottom-up best-fit placement policy. As discussed in Section VII, this task scheduling scheme can be adapted for scheduling processing elements on other multiple-SIMD and partitionable SIMD/MIMD systems.

Acknowledgment

The authors gratefully acknowledge the suggestions made by the referees. A preliminary version of portions of the material in this paper was presented at the 3rd International Conference on Distributed Computing Systems, October 1982.

References

[1] G. B. Adams, III and H. J. Siegel, "The extra stage cube: A fault-tolerant interconnection network for supersystems," *IEEE Trans. Comput.*, vol. C-31, pp. 443–454, May 1982.

[2] G. Barnes, *et al.*, "The Illiac IV computer," *IEEE Trans. Comput.*, vol. C-17, pp. 746–757, Aug. 1968.

[3] E. G. Coffman, Jr. and P. J. Denning, *Operating Systems Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

[4] T. Feng, "Data manipulating functions in parallel processors and their implementations," *IEEE Trans. Comput.*, vol. C-23, pp. 309–318, Mar. 1974.

[5] M. J. Flynn, "Very high-speed computer systems," *Proc. IEEE*, vol. 54, pp. 1901–1909, Dec. 1966.

[6] S. H. Fuller, "Performance evaluation," in *Introduction to Computer Architecture*, 2nd ed., H. S. Stone, Ed. Chicago, IL: Science Research Associates, 1980, pp. 527–590.

[7] A. N. Habermann, *Introduction to Operating System Design.* Chicago, IL: Science Research Associates, 1976.

[8] K. Hwang and L. M. Ni, "Resource optimization of a parallel computer for multiple vector processing," *IEEE Trans. Comput.*, vol. C-29, pp. 831–836, Sept. 1980.

[9] R. Y. Kain, A. A. Raie, and M. G. Gouda, "Multiple processor scheduling policies," in *Proc. 1st Int. Conf. Distributed Computing Systems*, Oct. 1979, pp. 660–668.

[10] J. Keng and K. S. Fu, "A special purpose architecture for image processing," in *Proc. 1978 IEEE Comput. Soc. Conf. Pattern Recognition Image Processing*, June 1978, pp. 287–290.

[11] J. T. Kuehn, H. J. Siegel, and P. D. Hallenbeck, "Design and simulation of an MC68000-based multimicroprocessor system," in *Proc. 1982 Int. Conf. Parallel Processing*, Aug. 1982, pp. 353–362.

[12] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. Comput.*, vol. C-24, pp. 1145–1155, Dec. 1975.

[13] R. J. McMillen and H. J. Siegel, "Routing schemes for the augmented data manipulator network in an MIMD system," *IEEE Trans. Comput.*, vol. C-31, pp. 1202–1214, Dec. 1982.

[14] G. J. Nutt, "Microprocessor implementation of a parallel processor," in *Proc. 4th Symp. Computer Architecture*, Mar. 1977, pp. 147–152.

[15] —, "A parallel processor operating system comparison," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 467–475, Nov. 1977.

[16] D. S. Parker, "Notes on shuffle/exchange-type switching networks," *IEEE Trans. Comput.*, vol. C-29, pp. 213–222, Mar. 1980.

[17] D. S. Parker and C. S. Raghavendra, "The gamma network: A multiprocessor interconnection network with redundant paths," in *Proc. 9th Symp. Comput. Architecture*, Apr. 1982, pp. 73–80.

[18] M. Pease, "The indirect binary n-cube microprocessor array," *IEEE Trans. Comput.*, vol. C-26, pp. 458–473, May 1977.

[19] D. M. Ritchie and K. Thompson, "The UNIX time-sharing system," *Bell Syst. Tech. J.*, vol. 57, pp. 1905–1929, July 1978.

[20] M. C. Sejnowski, E. T. Upchurch, R. N. Kapur, D. P. S. Charlu, and G. J. Lipovski, "An overview of the Texas reconfigurable array computer," in *Proc. AFIPS 1980 Nat. Comput. Conf.*, May 1980, pp. 631–641.

[21] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies.* Lexington, MA: Lexington Books, 1984.

[22] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Comput.*, vol. C-30, pp. 934–947, Dec. 1981.

[23] S. J. Stolfo and D. P. Miranker, "DADO: A parallel processor for expert systems," in *Proc. 1984 Int. Conf. Parallel Processing*, Aug. 1984, pp. 74–82.

[24] H. S. Stone, "Parallel computers," in *Introduction to Computer Architecture,* 2nd ed., H. S. Stone, Ed. Chicago, IL: Science Research Associates, 1980, pp. 363–425.

[25] D. C. Tsichritzis and P. A. Bernstein, *Operating Systems.* New York: Academic, 1974.

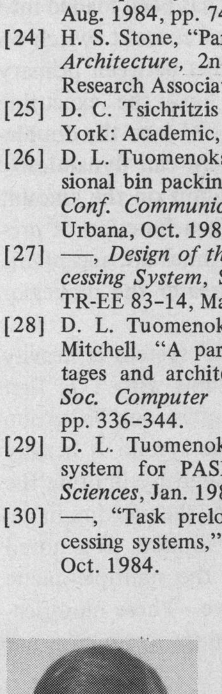
[26] D. L. Tuomenoksa and H. J. Siegel, "Application of two-dimensional bin packing to task scheduling in PASM," in *Proc. Allerton Conf. Communication, Control, and Computing*, Univ. Illinois, Urbana, Oct. 1981, p. 542.

[27] —, *Design of the Operating System for the PASM Parallel Processing System*, School Elec. Eng., Purdue Univ., Lafayette, IN, TR-EE 83-14, May 1983.

[28] D. L. Tuomenoksa, G. B. Adams, III, H. J. Siegel, and O. R. Mitchell, "A parallel algorithm for contour extraction: Advantages and architectural implications," in *Proc. IEEE Comput. Soc. Computer Vision Pattern Recognition 1983*, June 1983, pp. 336–344.

[29] D. L. Tuomenoksa and H. J. Siegel, "A distributed operating system for PASM," in *Proc. 17th Hawaii Int. Conf. System Sciences*, Jan. 1984, vol. 1, pp. 69–77.

[30] —, "Task preloading schemes for reconfigurable parallel processing systems," *IEEE Trans. Comput.*, vol. C-33, pp. 895–905, Oct. 1984.



David Lee Tuomenoksa (S'79–M'83) was born in Morristown, NJ, on October 14, 1958. He received the B.S. degree in 1980 and the Ph.D. degree in 1983, both in electrical engineering from Purdue University, West Lafayette, IN.

As a graduate student, he was a Research Assistant for the School of Electrical Engineering, Purdue University. He was also awarded a Purdue University Fellowship for Graduate Study. In June 1983 he joined AT&T Information Systems where he is currently a Member of the Technical Staff of the Integrated Systems Laboratory, Lincroft, NJ. His research interests include computer architecture, operating systems for distributed/ parallel computer organizations, image processing, and system reliability.

Dr. Tuomenoksa is a member of the Association for Computing Machinery and Eta Kappa Nu, Tau Beta Pi, and Phi Kappa Phi.

Howard Jay Siegel (M'77-SM'82) was born in New Jersey on January 16, 1950. He received the S.B. degree in electrical engineering and the S.B. degree in management from the Massachusetts Institute of Technology, Cambridge, in 1972; the M.A. and M.S.E. degrees in 1974, and the Ph.D. degree in 1977, all in electrical engineering and computer science from Princeton University, Princeton, NJ.

In 1976 Dr. Siegel joined the School of Electrical Engineering, Purdue University, West Lafayette, IN, where he is currently an Associate Professor. His research interests include parallel/distributed processing, multimicroprocessor systems, and image and speech understanding.

Dr. Siegel is currently the Chairman of the ACM SIGARCH (Special Interest Group on Computer Architecture). His previous activities include serving as Program Co-Chairperson of the 1983 International Conference on Parallel Processing, as the General Chariman of the Third International Conference on Distributed Computing Systems (1982), as an IEEE Computer Society Distinguished Visitor, as Chairman of the IEEE Computer Society TCCA (Technical Committee on Computer Architecture), and a guest editor of the IEEE TRANSACTIONS ON COMPUTERS. He is a member of Eta Kappa Nu and Sigma Xi.

Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory

DAVID M. WEISS AND VICTOR R. BASILI, MEMBER, IEEE

Abstract—An effective data collection methodology for evaluating software development methodologies was applied to five different software development projects. Results and data from three of the projects are presented. Goals of the data collection included characterizing changes, errors, projects, and programmers, identifying effective error detection and correction techniques, and investigating ripple effects.

The data collected consisted of changes (including error corrections) made to the software after code was written and baselined, but before testing began. Data collection and validation were concurrent with software development. Changes reported were verified by interviews with programmers. Analysis of the data showed patterns that were used in satisfying the goals of the data collection. Some of the results are summarized in the following.

1) Error corrections aside, the most frequent type of change was an unplanned design modification.

2) The most common type of error was one made in the design or implementation of a single component of the system. Incorrect requirements, misunderstandings of functional specifications, interfaces, support software and hardware, and languages and compilers were generally not significant sources of errors.

3) Despite a significant number of requirements changes imposed on some projects, there was no corresponding increase in frequency of requirements misunderstandings.

4) More than 75 percent of all changes took a day or less to make.

5) Changes tended to be nonlocalized with respect to individual components but localized with respect to subsystems.

6) Relatively few changes resulted in errors. Relatively few errors required more than one attempt at correction.

7) Most errors were detected by executing the program. The cause of most errors was found by reading code. Support facilities and techniques such as traces, dumps, cross-reference and attribute listings, and program proving were rarely used.

Manuscript received December 13, 1982; revised April 25, 1984. This work was supported in part by the National Aeronautics and Space Administration under Grant NSG-5123 to the University of Maryland.

V. R. Basili is with the Department of Computer Science, University of Maryland, College Park, MD 20742.

D. M. Weiss is with the Naval Research Laboratory, Washington, DC 20375.

Index Terms—Software change analysis, software change data, software errors, software measurement.

I. Introduction

In previous and companion papers [1]-[4] we have discussed how to obtain valid data that may be used to evaluate software development methodologies in a production environment. Briefly, the methodology consists of the following five elements.

1) Identify goals. The goals of the data collection effort are defined before any data collection begins. We often relate them to how well the goals for a product or process are met.

2) Determine questions of interest from the goals. Specific questions, derived from the goals, are used to sharpen the goals and define the data to be collected. Answering the questions derived from each goal satisfies the goal.

3) Develop a data collection form. The data collection form used is tailored to the product or process being studied and to the questions of interest.

4) Develop data collection procedures. Data collection is easiest when the data collection procedures are part of normal configuration control procedures.

5) Validate and analyze the data. Reviews and analyses of the data are concurrent with software development. Validation includes examining completed data collection forms for completeness and consistency. Where necessary, interviews with the person(s) supplying the data are conducted.

The purpose of this paper is to present the results from such an evaluation. The data presented here were collected as part of the studies conducted by NASA's Software Engineering Laboratory [5]. In this section we present an overview of the SEL and the projects analyzed for this paper. Section II describes the application of the methodology described in the